СЕРИЯ

ПИТЕР®

UNIX internals:
the new frontiers

Uresh Vahalia

Prentice Hall PTR
Upper Saddle River, New Jersey 07458
www.phptr.com

КЛАССИКА COMPUTER SCIENCE

Ю. ВАХАЛИЯ

UNIX ИЗНУТРИ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2003

ББК 32.973-018.2
УДК 681.3.066
В22

В22 **UNIX изнутри** / Ю. Вахалия. — СПб.: Питер, 2003. — 844 с.: ил. — (Серия «Классика computer science»)

ISBN 5-94723-013-5

Эта книга показывает ядро UNIX с точки зрения разработчика систем. Для каждого компонента ядра приводится описание архитектуры и внутреннего устройства, практической реализации в каждом из описываемых вариантов операционной системы, а также преимуществ и недостатков альтернативных вариантов рассматриваемого компонента. Вы увидите описание основных коммерческих и научных реализаций операционной системы

Книга не рассчитана на начинающих и содержит знания о таких концептуальных вещах, как ядро системы, процессы или виртуальная память Она может быть использована как профессиональное руководство или как пособие для изучения UNIX в высших учебных заведениях. Уровень изложения материала достаточен для изложения в качестве основного или дополнительного курса лекций по операционным системам.

ББК 32 973-018 2
УДК 681.3.066

ISBN 0-13-101908-2 (англ)
ISBN 5-94723-013-5

Краткое содержание

Содержание

Эта книга посвящается Бхинне (Bhinna), память о которой навсегда останется в моем сердце, Рохану (Rohan) за его веселость и энтузиазм, а также Аркане (Archana), за ее любовь и поддержку.

От редактора английского издания

П. Х. Салюс (P. H. Salus), ведущий редактор Computing Systems

На сегодняшний день версий UNIX существует больше, чем производителей мороженого. Несмотря на подталкивание частью консорциума X/Open и его членами, единая спецификация UNIX все более удаляется от нас. Однако это и не есть главная цель. С тех пор как компания Interactive Systems представила первую коммерческую систему UNIX, а компания Whitesmiths создала первый клон UNIX, пользователи были поставлены перед фактом появления самых разнообразных вариантов системы, разработанных под разные платформы.

Система UNIX была создана в 1969 году. Не прошло и десяти лет, как ее версии начали множиться. Когда UNIX исполнилось 20 лет, уже существовали крупные консорциумы (Open Software Foundation и UNIX International) и большое количество различных реализаций ОС. Два основных потока развития системы исходили из AT&T (сейчас Novell) и Калифорнийского университета в Беркли. Описания этих вариантов UNIX, созданные Морисом Бахом (Maurice Bach) [1] и Сэмом Леффлером (Sam Leffler), Кирком МакКьюзиком (Kirk McKusick), Майком Кэрельсом (Mike Karels) и Джоном Квотерманом (John Quarterman) [2], можно легко найти.

Ни одна книга ранее не предлагала описание реализаций операционной системы UNIX с интересной для студентов точки зрения. Эту задачу выполнил Юреш Вахалия (Uresh Vahalia). Он сделал то, чего до него не создавал ни один автор, подробно обрисовал внутреннее устройство систем SVR4, 4.4BSD и Mach. Более того, книга содержит тщательно продуманное изложение компонентов систем Solaris и SunOS, Digital UNIX и HP-UX.

Он сделал прекрасное описание систем, свободное от предпочтений какого-либо одного варианта UNIX, часто не скрываемых другими авторами. Несмотря на то что уже создаются различные реализации относительно новых систем, таких как Linux, и даже варианты Berkeley значительно отличаются между собой, такие книги, как эта, показывают внутреннее устройство UNIX и заложенные в нее принципы, ставшие причиной популярности системы в экспоненциальной зависимости.

12 июня 1972 года Кен Томпсон (Ken Thompson) и Денис Ритчи (Dennis Ritchie) представили вторую редакцию своего руководства *UNIX Programmer's Manual*. В предисловии авторы заметили: «Количество инсталляций UNIX достигло десяти — более, чем мы ожидали». Они даже не могли предположить, что действительно произойдет с их системой в будущем.

Я рассмотрел появление и историю развития систем в книге [3], но Вахалия дал нам действительно оригинальный и исчерпывающий взгляд на сравнительную анатомию систем.

Ссылки:

1. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
2. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3 BSD UNIX Operating System», Addison-Wesley, Reading, MA, 1989.
3. Salus, P. H., «A Quarter Century of UNIX», Addison-Wesley, Reading, MA, 1994.
4. Thompson, K., and Ritchie, D. M., «UNIX Programmer's Manual», Second Edition, Bell Telephone Laboratories, Murray Hill, NJ, 1972.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Предисловие

Со времени своего появления (начало 70-х годов) система UNIX претерпела значительные изменения: начав свое развитие с небольшой экспериментальной операционной системы, распространяемой бесплатно Bell Laboratories, на сегодняшний день UNIX представляет собой целое семейство ее преемников. За эти годы она вобрала в себя огромное количество предложений от представителей науки и промышленности, прошла через множество битв за стандартизацию и авторские права и превратилась в стабильную целостную операционную систему. Существует несколько реализаций UNIX, предназначенных как для коммерческого, так и для научного использования, которые имеют определенные сходства между собой, достаточные для причисления их к одному и тому же типу операционных систем. Программист, изучивший один из клонов UNIX, может также производительно работать с другими аппаратными платформами и версиями операционной системы без необходимости переучивания.

Разные возможности самых различных реализаций ОС UNIX описаны в сотнях изданных книг. Хотя многие из этих трудов показывают системы со стороны пользователя, рассказывая, например, о командной оболочке или программном интерфейсе, лишь малая часть книг посвящена внутреннему устройству UNIX. Изучение архитектуры UNIX подразумевает описание ядра, являющегося «сердцем» каждой операционной системы. На сегодняшний день все существующие книги по UNIX описывают только какую-либо одну реализацию системы. Например, книга М. Дж. Баха «The Design of the UNIX Operating System» [1] является наиболее ярким описанием ядра System V Release 2 (SVR2), книга С. Дж. Леффлера и др. «The Design and Implementation of the 4.3 BSD UNIX Operating System» [4] представляет собой подробное описание системы 4.3BSD от лица ее создателей, внутреннее устройство ОС System V Release 4.0 (SVR4) раскрывают на страницах своей книги «The Magic Garden Explained — The Internals of UNIX System V Release 4» Б. Гудхарт и Дж. Кокс [3].

Изложение материала

Эта книга показывает ядро UNIX с точки зрения разработчика систем. Вы увидите описание основных коммерческих и научных реализаций операционной системы. Для каждого компонента ядра приводится описание архитек-

туры и внутреннего устройства, практической реализации в каждой из описываемых версий операционной системы, а также преимуществ и недостатков альтернативных вариантов рассматриваемого компонента. Такой сравнительный подход придает книге отличительную особенность и дает возможность читателю рассматривать систему с критической точки зрения. При изучении операционной системы важно знать не только сильные, но и слабые ее стороны. Это возможно только при проведении анализа альтернативных вариантов.

Реализации UNIX

В этой книге большое внимание уделяется системе SVR4.2, однако здесь вы можете найти подробное описание 4.4BSD, Solaris 2.x, Mach и Digital UNIX. Более того, на страницах книги рассказывается и о самых интересных возможностях других вариантов UNIX, в том числе разработок, до сих пор не реализованных в коммерческих версиях ОС, проводится анализ развития UNIX, начиная от середины 80-х и заканчивая серединой 90-х годов. Для цельности повествования в материал книги включено краткое описание основных возможностей и реализаций системы UNIX. Если это необходимо, описание включает в себя исторический контекст, ведется, начиная со стандартных функций, проведения анализа недостатков и ограничений и заканчивая представлением последних разработок.

Для кого предназначена эта книга

Предлагаемая книга может быть использована как профессиональное руководство для изучения в высших учебных заведениях. Уровень изложения материала достаточен для изложения в качестве основного или дополнительного курса лекций по операционным системам. Книга не рассчитана на начинающих и содержит знания о таких концептуальных вещах, как ядро системы, процессы или виртуальная память. В конце каждой главы приводится набор вопросов, разработанных для стимулирования дальнейшего самостоятельного изучения и поиска дополнительного материала, а также для более глубокого изучения внутреннего устройства систем. Ответы на многие вопросы остаются открытыми, а для некоторых из них нужно изучение дополнительной литературы. Каждая глава завершается исчерпывающим списком материалов, которые могут быть использованы студентами для более подробного ознакомления с описываемой тематикой.

Книга также является профессиональным руководством для разработчиков операционных систем, программных приложений и для системных администраторов. Разработчики систем могут использовать ее для изучения архитектуры ядра существующих ОС, сравнения преимуществ и недостатков

различных реализаций систем, а также использовать изложенный материал для создания следующих поколений операционных систем. Программисты могут применить полученные знания внутреннего устройства систем для написания более эффективных приложений, максимально задействующих полезные возможности UNIX. Рассказ о разнообразных параметрах и описание функционирования систем при использовании их определенных комбинаций поможет системным администраторам в настройке и наладке обслуживаемых ими операционных систем.

Как организована эта книга

Первая глава («Введение») описывает эволюцию систем UNIX и анализирует факторы, ставшие причиной основных изменений, произошедших в системе. В главах 2–7 изложено функционирование подсистем. В частности, глава 2 рассказывает о возможностях традиционных систем UNIX (SVR3, 4.3BSD и более ранних реализаций), в то время как на страницах глав 3–7 вы познакомитесь с возможностями современных ОС, таких как SVR4, 4.4BSD, Solaris 2.x и Digital UNIX. В третьей главе описаны потоки и их реализация в ядре системы и программах пользователя. В главе 4 говорится о сигналах, управлении процессами и обработке сессий входа в систему. Глава 5 посвящена диспетчеру UNIX и постоянно растущей поддержке приложений, работающих в режиме реального времени. Из материала главы 6 вы узнаете о взаимодействии процессов (IPC), а также о возможностях системы под названием System V IPC. Здесь же описана архитектура системы Mach, использующей IPC как основу построения ядра. Глава 7 расскажет о синхронизации выполнения процессов, используемой в современных одно- и многопроцессорных системах.

Следующие четыре главы книги посвящены файловым системам. Глава 8 описывает интерфейс файловой системы с точки зрения ее пользователя, а также рассказывает о механизме vnode/vfs, определяющем взаимодействие между ядром и файловой системой. В главе 9 рассматриваются подробности реализаций различных файловых систем, в том числе оригинальной файловой системы ОС System V (s5fs), Berkley Fast File System (FFS), а также других, более редких специализированных файловых систем, использующих наилучшие возможности vnode/vfs. Глава 10 представляет большое количество распределенных файловых систем: Network File System (NFS) компании Sun Microsystems, Remote File Sharing (RFS) компании AT&T, Andrew File System, разработанной в университете Карнеги—Меллона, и Distributed File System (DFS), созданной корпорацией Transarc Corporation. Глава 11 содержит рассказ о расширенных файловых системах, использующих ведение журнала с целью достижения более высокого уровня работоспособности и производительности, а также о новой интегрированной среде файловой системы, построенной на наращиваемых уровнях vnode.

Главы 12–15 описывают управление памятью. В главе 12 рассказано о выделении памяти ядром и приведены некоторые интересные алгоритмы выделения памяти. В главе 13 представлено понятие виртуальной памяти, некоторые особенности ее использования проиллюстрированы на примере системы 4.3BSD. Глава 14 посвящена описанию построения виртуальной памяти в SVR4 и Solaris, глава 15 расскажет о моделях памяти в системах Mach и 4.4BSD. В этом разделе вы также увидите анализ эффективности таких аппаратных возможностей, как буфер ассоциативной трансляции и кэш-память.

Последние две главы книги затрагивают подсистему ввода-вывода. Глава 16 описывает работу драйверов устройств, взаимодействие между ядром и подсистемой ввода-вывода, а также интерфейс драйверов устройств системы SVR4. В главе 17 приведено описание STREAMS, используемых для написания сетевых протоколов, а также сетевых драйверов или драйверов терминалов.

Некоторые обозначения, принятые в книге

Все системные вызовы, библиотеки подпрограмм, а также команды оболочки выделены специальным моноширинным шрифтом (например, `fork`, `fopen`, `ls -l`). Имена внутренних функций ядра, переменных и примеры кода также оформлены моноширинным шрифтом (например, `ufs_lookup()`). Новые термины выделяются курсивом. Имена файлов и каталогов также выделяются шрифтом (например, `/etc/passwd`). На рисунках сплошными линиями показаны прямые указатели, в то время как прерывистые линии указывают на то, что взаимосвязь между их начальной и конечной точкой является только косвенной.

Несмотря на все усилия автора возможно существование в книге некоторого количества ошибок. Присылайте все поправки, комментарии и предложения на адрес электронной почты автора `vahalia@acm.org`.

Благодарности

В создании книги участвовало много людей. В первую очередь я хочу поблагодарить моего сына Рохана и мою жену Аркану, чье терпение и любовь сделали написание этой книги возможным, ведь самым сложным для меня оказалось просиживание вечеров и выходных над ее созданием вместо того, чтобы провести это время со своей семьей. Они разделили вместе со мной этот тяжелый труд и постоянно поддерживали меня. Я также хочу поблагодарить моих родителей за их любовь и поддержку.

Далее я хотел бы поблагодарить моего друга Субода Бапата (Subodh Bapat), который дал мне уверенность в осуществлении этого проекта. Именно он помог мне сконцентрировать внимание на проекте и потратил большое количество своего времени на советы, консультации и поддержку. Я должен

также поблагодарить его за предоставление доступа к инструментам, шаблонам и макросам, использующимся в этой книге, за его труд «Объектно-ориентированные сети» [2], за тщательную обработку предварительных версий моего труда и за консультации по стилю изложения материала.

Для улучшения этой книги было потрачено время и использован опыт не одного рецензента. Книга имела несколько предварительных вариантов, в ходе изучения которых было получено большое количество комментариев и предложений. Я хочу поблагодарить Питера Салюса (Peter Salus) за его постоянную поддержку и консультирование, а также Бенсона Маргулиса, Терри Ламберта (Terry Lambert), Марка Эллиса (Mark Ellis) и Вильяма Балли (William Bully) за помощь в создании содержания и организации этой книги. Я также хочу поблагодарить Кейт Бостик (Keith Bostic), Еви Немет (Evi Nemeth), Пэт Парсегян (Pat Parseghian), Стивена Раго (Steven Rago), Марго Сельцер (Margo Seltser), Ричарда Стивенса (Richard Stevens) и Льва Вэйзблита (Lev Vaitzblit), написавших рецензии на отдельные части книги.

Я хочу поблагодарить моего менеджера Перси Цельника (Percy Tzelnic) за поддержку и понимание. Также я хочу выразить признательность своему издателю Алану Апту (Alan Apt) не только за появление этой книги, но и за помощь на каждом этапе ее создания, а также остальному коллективу издательств Prentice-Hall и Spectrum Publisher Services и особенно Ширли Макгуайр (Shirley McGuire), Сондре Чавес (Sondra Chavez) и Келли Риччи (Kelly Ricci) за их помощь и поддержку.

Дополнительная литература

1. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
2. Bapath, S. G., «Object-Oriented Networks», Prentice-Hall, 1994.
3. Goodheart B., Cox J., «The Magic Garden Explained — The Internals of UNIX System V Release 4», An Open System Design, Prentice-Hall, 1994.
4. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3 BSD UNIX Operating System», Addison-Wesley, Reading, MA, 1989.

Глава 1
Введение

1.1. Введение

В 1994 году компьютерное сообщество отметило двадцатипятилетие с момента появления операционной системы UNIX. После своего рождения в 1969 году система была перенесена на множество различных аппаратных платформ, появилось большое количество ее реализаций, созданных коммерческими компаниями, учебными заведениями и научно-исследовательскими организациями. Система UNIX начала свое развитие с небольшого набора программ и за годы переросла в гибкую ОС, использующуюся для работы огромного количества программных сред и приложений. На сегодняшний день существуют варианты UNIX для самых различных аппаратных платформ, начиная от небольших встроенных процессоров, рабочих станций и настольных систем и заканчивая высокопроизводительными многопроцессорными системами, объединяющими большое количество пользователей.

Операционная система UNIX — это среда выполнения и системные службы, под управлением которых функционируют входящие в набор ОС пользовательские программы, утилиты и библиотеки. Эта книга рассказывает о разработке и реализации самой системы, не описывая программы и утилиты, выполняющиеся под ее управлением. Система UNIX начала свою жизнь в недрах Bell Telephone Laboratories (BTL), которая и ответственна за все ее ранние реализации. Сначала система получила распространение среди нескольких компаний и учебных заведений. Именно этот факт повлиял на дальнейшее развитие различных реализаций UNIX. Все они поддерживали только набор внутренних интерфейсов, приложений и возможностей, обычно ожидаемых от стандартной «системы UNIX». Внутреннее устройство каждой было различным, отличаясь семантикой интерфейсов и набором предоставляемых «дополнительных» возможностей. В этой книге уделяется особое внимание описанию основополагающих реализаций UNIX, таких как *System V Release 4* (*SVR4*) компании Novell, *Berkeley Software Distribution* (*4xBSD*) Калифорнийского университета и *Mach* университета Карнеги—Меллона. Здесь также обсуждается большое количество коммерческих ва-

риантов системы, таких как *SunOS* и *Solaris* компании Sun Microsystems, *Digital UNIX* компании Digital Equipment Corporation и *HP-UX* корпорации Hewlett-Packard Corporation.

Эта глава посвящена описанию развития систем UNIX. Сначала вы увидите краткий рассказ о рождении, становлении и принятии системы компьютерным сообществом. Затем будут описаны факторы, повлиявшие на ее эволюцию, и, наконец, будут отмечены возможные направления дальнейшего развития UNIX.

1.1.1. Краткая история

Перед тем как начать изучение операционной системы UNIX, полезно немного узнать о ее истории и эволюции. В следующих разделах мы проследим развитие UNIX от зарождения ее в недрах AT&T до нынешнего современного, немного хаотичного состояния в виде набора операционных систем, реализованных под различные платформы, различными авторами и существующих в самых различных вариантах. Более полное изложение истории развития UNIX можно найти в других публикациях, например в книге Питера Салюса (Peter Salus) «A Quarter Century of UNIX» [1]. Эта глава расскажет только об основных событиях, произошедших в истории системы UNIX.

1.1.2. Начало

В конце 60-х годов Bell Telephone Laboratories совместно с компанией General Electric и Массачусетским технологическим институтом организовали совместный проект, целью которого была разработка операционной системы под названием *Multics* [2]. Проект был аннулирован в марте 1969, но некоторые разработчики из BTL продолжили поиск интересных идей для последующей реализации. Один из участников проекта, Кен Томпсон (Kenneth Thompson), написал игровую программу под названием «Космическое путешествие» («Space Travel») и нашел для ее запуска малоиспользуемый в те годы компьютер *PDP-7* (созданный Digital Equipment Corporation). Однако этот компьютер не имел собственной среды разработки программ, поэтому Томпсон перенес свою программу на другую машину, Honeywell 635, работавшую под управлением ОС GECOS, и затем создал перфоленту со своей программой для PDP-7.

С целью совершенствования разработки «Космического путешествия» Томпсон совместно с Денисом Ритчи (Dennis Ritchie) начал разработку рабочей среды для PDP-7. Первым созданным компонентом стала простая файловая система, дальнейшее развитие которой впоследствии появилось в первых версиях UNIX и сейчас известно как *System V file system* (*s5fs*). Чуть позже ими были добавлены подсистема обработки, простой командный

интерпретатор под названием shell (который позже развился в Bourne shell, [4]), а также небольшой набор утилит. Система стала самодостаточной и не требовала больше среды GECOS. Авторы назвали свою систему *UNIX* в честь проекта Multics[1].

В следующем году Томпсон, Ритчи и Йозеф Оссанна (Joseph Ossanna) добились того, что BTL приобрела машину Digital *PDP-11* для обработки документации в отделе патентов. Затем они экспортировали UNIX на эту машину и добавили несколько утилит обработки текста, в том числе редактор ed и инструмент отображения текста runoff. Томпсон также разработал новый язык программирования *B* (усовершенствовав тем самым язык *BPCL*, [5]) и написал на нем несколько первых ассемблеров и утилит. Язык B был интерпретируемым, вследствие чего обладал низкой производительностью. Позже Ритчи усовершенствовал свою разработку, назвав результат *C*. Язык C поддерживал типы и структуры данных. Успех языка C являлся основой успешного развития системы UNIX.

ОС UNIX становилась все более популярной внутри BTL. В ноябре 1971 года Ритчи и Томпсон под руководством Дуга Мак-Илроя (Doug McIlroy) опубликовали первую редакцию «Руководства для программиста UNIX». В дальнейшем появилось ровно 10 изданий этого руководства — по количеству версий систем UNIX, созданных в лабораториях BTL.

Первые несколько реализаций ОС использовались только внутри BTL. Третья версия, вышедшая в феврале 1973, включала в себя компилятор языка C под названием cc. В том же году система была переписана на языке C, в результате чего в ноябре того же года появилась версия 4. Это в высокой степени повлияло на будущий успех системы UNIX. Томпсон и Ритчи создали первую работу о UNIX под названием «The UNIX Timesharing System» [6], которая была представлена на симпозиуме по операционным системам (*ACM Symposium on Operating Systems, SOSP*) в октябре 1973 года и опубликована в июле 1974 года в *Communications of ACM*[2]. Эта работа стала первой публикацией, возвестившей миру о UNIX.

---

[1] Питер Салюс в своей книге рассказывает, что этимология слова UNIX обязана своим происхождением шуткам коллег-хакеров. Multics была многопользовательской системой, а первая UNIX работала всего с двумя пользователями. Латинский корень «много» заменили на «один» («единственный»). Получилось — UNICS (Uniplexed Information and Computing Service). Название понравилось, поскольку напоминало об участии сотрудников Bell Labs в разработке Multics. Позже UNICS было изменено на UNIX. Том ван Влек, лично зная всех персонажей, «из первых рук» уточняет (http://www.multicians.org/unix.html), что название UNIX предложил Брайан Карниган как «Multics without balls». История имела продолжение. Когда хакерская коалиция распалась, Ричард Столлмен решил создать систему, «совместимую с UNIX, чтобы она была переносимой и чтобы пользователи UNIX могли бы легко на нее перейти. Аббревиатура GNU была выбрана для нее в соответствии с хакерской традицией как рекурсивный акроним выражения «GNUTs Not UNIX » («GNU — это не UNIX »)». — *Прим. ред.*

[2] Позже эта работа была подвергнута изменению и переиздана в виде книги [7].

1.1.3. Распространение

В 1956 году в результате антимонопольной судебной тяжбы Министерства юстиции США против компаний AT&T и Western Electric Company корпорация AT&T вынуждена была подписать согласительный документ с правительством. Это соглашение запрещало компании производить какое-либо оборудование, не относящееся к телефонам или телеграфу, а также вести дела в областях, отличных от «общих служб доставки сигнала».

В результате компания AT&T не могла заниматься продажей продукции, относящейся к вычислительной технике. С другой стороны, конференция SOSP показала наличие большого спроса на программное обеспечение UNIX. Корпорация AT&T распространяла свою систему в высших учебных заведениях для использования в образовательных и научных целях под простыми лицензионными условиями, что не противоречило подписанному соглашению. Компания не рекламировала созданную систему и не продвигала ее на рынке сбыта, а также не вела поддержку ее реализаций. Одним из первых лицензию на использование UNIX от компании AT&T получил университет Беркли (Berkeley) в Калифорнии. Это произошло в декабре 1973 года.

Такие условия дали возможность системе UNIX довольно быстро распространиться по всему миру. О широкой географии использования ОС говорит тот факт, что к 1975 году система была установлена в таких учебных заведениях, как Еврейский университет в Иерусалиме, университет Нового Южного Уэльса (Австралия) и университет Торонто (Канада). Первым переносом системы на новую аппаратную платформу стала ее реализация для машины *Interdata*, полностью выполненная университетом Уолонгонга (Wollongong) самостоятельно в 1976 году. Годом позже аналогичный процесс был осуществлен Ритчи и Стивом Джонсоном в BTL.

Седьмая версия операционной системы UNIX, выпущенная в январе 1979 года, являлась первой настоящей переносимой ОС, что послужило большим толчком в ее дальнейшем развитии. Эта версия изначально работала на PDP-11 и Interdata 8/32. Она, с одной стороны, была устойчивее и намного функциональнее своей предшественницы, версии 6, однако, с другой стороны, работала значительно медленнее. Существовало несколько лицензий ОС, позволяющих увеличить ее производительность при использовании в различных областях. Компания AT&T позже вставила многие из этих разработок в последующие версии UNIX. Кооперация между разработчиками и пользователями системы (которая, к сожалению, перестала быть возможной после коммерческого успеха ОС) была ключевым фактором, обусловившим быстрый рост и увеличение популярности UNIX.

Вскоре система UNIX была импортирована на другие аппаратные платформы. Корпорация Microsoft совместно с Santa Cruz Operation (SCO) перенесла систему на компьютеры под управлением процессора Intel 80x86, в результате чего появилась ОС *XENIX* — один из первых коммерческих вариантов UNIX. В 1978 году компания Digital представила свой новый 32-разрядный компьютер VAX-11 и предложила группе разработчиков отделения BTL в Холмделе,

штат Нью-Джерси, перенести UNIX на эту машину. Так возник первый вариант системы UNIX для 32-разрядного компьютера, который был назван *UNIX/32V*. Копия этой системы была передана Калифорнийскому университету, где была переработана и стала основой ОС 3BSD, появившейся в 1979 году.

1.1.4. BSD

Калифорнийский университет в Беркли получил одну из первых лицензий на операционную систему UNIX в декабре 1974 года. За несколько лет группа выпускников университета, в состав которой входили Билл Джой (Bill Joy) и Чак Хэлей (Chuck Haley), разработала несколько утилит для этой системы, в том числе редактор ex (позже сопровождавшийся vi) и компилятор языка Паскаль. Все созданные приложения были собраны в единый пакет под названием *Berkeley Software Distribution* (*BSD*) и продавались весной 1978 года по цене $50 за одну лицензию. Первые версии BSD (2BSD появилась в конце 1978 года) состояли из дополнительных приложений и утилит, сама же операционная система тогда еще не подвергалась изменению или передаче. Одной из первых разработок Джоя стала оболочка C shell [7], имевшая такие возможности, как управление заданиями и ведение истории команд, отсутствовавшие в то время в оболочке Bourne.

В 1978 году университет Беркли приобрел машину VAX-11/780 и операционную систему UNIX/32V, портирование которой на этот компьютер осуществила группа BTL в Холмделе, штат Нью-Джерси. Компьютер VAX имел 32-разрядную архитектуру и мог использовать до 4 Гбайт адресного пространства, однако имел всего лишь 2 Мбайт физической памяти. Примерно в то же время Озалп Бабаоглу (Ozalp Babaoglu) разработал систему страничной виртуальной памяти для VAX и добавил ее в ОС UNIX. В результате в конце 1979 года появилась новая версия ОС, 3BSD, которая стала первой операционной системой, созданной университетом Беркли.

После появления системы виртуальной памяти агентство DARPA (Defence Advanced Research Projects Agency) начало финансирование разработки систем UNIX в Беркли. Одной из главных задач, стоявших перед DARPA, являлась интеграция в создаваемой системе набора протоколов *TCP/IP* (Transmission Control Protocol/Internet Protocol). При финансовой поддержке агентства DARPA Berkeley выпустил несколько вариантов системы BSD, объединенных под общим названием *4BSD*: *4.0BSD* в 1980 году, *4.1BSD* в 1981[1], *4.2BSD* в 1983, *4.3BSD* в 1986 и *4.4BSD* в 1993.

Команде Беркли принадлежало авторство большого количества важных технических усовершенствований системы. Кроме уже упомянутых новшеств (виртуальной памяти и интеграции протоколов TCP/IP), в системе BSD UNIX была представлена файловая система *Fast File System* (*FFS*), надежная реализация сигналов и технология сокетов. В 4.4BSD оригинальная разработка виртуальной памяти была заменена новой версией, базирующей-

[1] Эта версия системы, в свою очередь, имела три различных варианта: 4.1a, 4.1b и 4.1c.

ся на Mach (см. раздел 1.1.7), а также были добавлены другие возможности, например файловая система с ведением журнала.

Работа над системой UNIX производилась группой CSRG (Computer Science Research Group). Однако после выпуска 4.4BSD группа приняла решение закрыть проект и завершить разработку систем UNIX. Наиболее важными причинами этого решения были:

- ✦ уменьшение финансирования и выделения грантов;
- ✦ возможности, представленные в системе BSD, к тому времени уже были реализованы во многих коммерческих проектах;
- ✦ операционная система становилась слишком большой и сложной для разработки и поддержки силами небольшой группы программистов.

Для продвижения и продажи 4.4BSD как коммерческого продукта была создана компания Berkeley Software Design, Inc (BSDI). К тому времени почти весь код оригинальной системы UNIX был заменен разработчиками из Беркли, поэтому компания BSDI утверждала, что созданная ею версия системы *BSD/386* полностью свободна от лицензионных ограничений AT&T. Однако подразделение AT&T, UNIX System Laboratories, занимавшееся разработками UNIX, все-таки подало иск против BSDI и управляющего совета Калифорнийского университета. Компания обвиняла их в нарушении авторских прав, невыполнении условий соглашения, в незаконном перехвате коммерческих секретов, а также выступала резко против использования BSDI телефонного номера 1-800-ITS-UNIX для продажи исходных кодов своей системы. Университет подал ответный иск, вследствие чего продажи BSD/386 были приостановлены. В результате 4 февраля 1994 года обе стороны договорились между собой вне здания суда и отменили иски друг к другу, после чего компания BSDI анонсировала новый продукт, *4.4BSD-lite*, продаваемый без исходных кодов примерно по $1000 за пакет[1].

---

[1] Разработчики BSD UNIX были вынуждены вести юридическую битву с AT&T, что замедлило темпы развития системы, а успех SVR5 и альянса единой UNIX во главе с SCO к 2000 году окончательно выбил почву из-под ног университета Беркли и приверженцев коммерциализации этой ветви развития UNIX. Спецификация 4.4BSD используется во многих коммерческих ОС, но суммарный объем продаж исчезающе мал, прежде всего, по причине сильнейшей конкуренции со стороны бесплатных систем FreeBSD (www.freebsd.org), OpenBSD (www.openbsd.org) и NetBSD (www.netbsd.org). Последние версии FreeBSD (в основе которой лежит 4.4BSD-lite) ни в чем не уступают коммерческим разработкам. FreeBSD 4.7 (октябрь 2002) улучшена относительно версии 4.6, поддерживает архитектуры Intel и ALPHA и может быть установлена из Сети с одного из множества анонимных ftp-серверов. FreeBSD 5.0 (январь 2003) явилась результатом трехлетнего труда и работает на 64-разрядных архитектурах Intel и SPARC. В ее файловой системе UFS2, следующем поколении UFS, преодолен терабайтный барьер на размер файла. Функции резервного копирования реализуются на основе Background filesystem checking (bgfsck) и моментальных снимков. Модель безопасности включает экспериментальный сервис Mandatory Access Controls (MAC) и предоставляет более гибкие возможности администраторам. Работа в многопроцессорных системах обеспечивается высоким уровнем грануляции ядра. Такие новшества, как инфраструктура устройств памяти GEOM и файловая система виртуальных устройств DEVFS, облегчают управление устройствами памяти. — *Прим. ред.*

1.1.5. System V

Вернемся снова к истории AT&T. Судебные битвы корпорации с министерством достигли кульминации в 1982 году, с выходом разграничивающего постановления. Во исполнение его предписаний корпорация Western Electric распалась на части, AT&T лишилась своих региональных отделений, которые стали основой «детей Белла» («Baby Bells»), Bell Telephone Laboratories была отделена и переименована в AT&T Bell Laboratories. Также корпорация AT&T отныне получила возможность заниматься бизнесом в компьютерных отраслях.

Работа над системой UNIX по-прежнему велась группой разработчиков из BTL, но создание внешних реализаций системы постепенно перешло к UNIX Support Group, затем к UNIX Development Group и, наконец, к AT&T Information Systems. Этими группами были созданы *System III* в 1983 году, *System V* в 1983, *System V Release 2* (*SVR2*) в 1983 и *System V Release 3* (*SVR3*) в 1987. Корпорация AT&T агрессивно продвигала System V на рынок операционных систем. На основе этой ОС было разработано несколько различных коммерческих вариантов UNIX.

В System V были впервые представлены многие новые возможности и средства. Например, сегментная реализация виртуальной памяти отличалась от варианта, предлагаемого BSD. В SVR3 появилось средство взаимодействия процессов (поддерживающее совместное использование памяти, семафоров и очередей сообщений), удаленное разделение файлов, общие библиотеки, а также поддержка STREAMS для драйверов устройств и сетевых протоколов. Последняя версия операционной системы System V Release 4 (*SVR4*) будет подробнее описана в разделе 1.1.10[1].

1.1.6. Коммерциализация

Увеличивающаяся популярность UNIX вызвала интерес к этой ОС со стороны нескольких компаний, которые приняли решение начать выпуск и продажу своих собственных вариантов системы. Производители брали за основу одну из базовых реализаций UNIX от AT&T или Беркли, переносили вы-

[1] В марте 2000 года компания SCO представила новую ОС на основе System V Release 5 (SVR5), усовершенствованной SVR4.2, которая включила в себя более производительную сетевую подсистему — улучшенные механизмы синхронизации процессов, планирования и управления памятью, поддержку до 32 процессоров в сервере и технологий NUMA и I2O, файлов размером до 1 Тбайт, 76 800 Тбайт внешней памяти, разбиваемой до 512 логических томов. SVR5 работает с 64-гигабайтной RAM (прямая адресация к 4 Гбайт, остальная — в режиме расширенной физической адресации (PAE)). Технология Multi-path I/O позволяет использовать несколько контроллеров ввода-вывода и удвоение дисков. Поддержана 64-разрядная журнальная отказоустойчивая файловая система VxFS компании Veritas, 64-разрядый API и системные вызовы. Дополнительно: графические Java-консоли администрирования (SCOadmin), программная поддержка RAID, кластеризация, файл-серверы и серверы печати, совместимые с Windows, и многое другое. — *Прим. ред.*

бранную систему на свои машины и добавляли в ОС свои *дополнительные* возможности. Первой компанией, начавшей в 1977 году продажу собственного коммерческого варианта UNIX, стала Interactive Systems. Продукт назывался *IS/1* и работал на PDP-11.

В 1982 году Билл Джой покинул Беркли и стал одним из создателей корпорации Sun Microsystems, выпустившей свой вариант UNIX на основе 4.2BSD и назвавшей его *SunOS* (позже компанией была разработана еще одна версия системы, *Solaris*, базирующаяся на 4.3BSD). Компании Microsoft и SCO совместно создали систему *XENIX*. Позже SCO перенесла SVR3 на платформу 80386 и выпустила эту систему под названием SCO UNIX. В 80-х годах появилось множество различных коммерческих вариантов системы UNIX, в том числе *AIX* от IBM, *HP-UX* от Hewlett-Packard Corporation и *ULTRIX* (выпущенной вслед за *DEC OSF/1*, переименованной позже в *Digital UNIX*) от компании Digital[1].

В коммерческих вариантах UNIX были представлены многие новые возможности, некоторые из которых позже были встроены и в базовые системы. В SunOS была реализована сетевая файловая система Network File System (NFS), интерфейс vnode/vfs, поддерживающий работу с различными типами файловых систем, а также новая архитектура виртуальной памяти, адаптированная позже в SVR4. Система AIX одна из самых первых начала поддержи-

[1] Выходя за «рамки повествования книги», следует заметить, что на сегодняшний день роли на рынке коммерческих систем UNIX достаточно четко определились. Интерес к коммерческим вариантам UNIX усилился в связи с успехом Linux. Радикальные перемены произошли в 1997–2000 годах, когда компания SCO представила System V Release 5, являющуюся усовершенствованной версией SVR4.2, и на ее базе выпустила ОС UnixWare 7. В настоящее время SCO оккупировала 80% рынка ОС для серверов UNIX на платформе Intel. UnixWare 7 наследовала лучшее из UnixWare 2 и SCO OpenServer 5. UnixWare 7 поддерживает процессоры Pentium, 64-разрядный Merced и 64-разрядную RISC-архитектуру (PowerPC). SCO организовала альянс с IBM и Sequent (IBM собиралась принести в жертву AIX во благо стандартного диалекта UNIX для платформ IA-32/64, поскольку компьютеры с процессорами Intel составляют половину рынка систем UNIX), Intel и др. по разработке единой UNIX, условно названной Monterey. Результатом должно было стать объединение UnixWare, элементов AIX и Sequent Dynix. Принять новую ОС в качестве стандарта согласились Compaq, Siemens Nixdorf, Hyundai, ICL HPS, Fujitsu/ICL. Ряд производителей объявили о разработке версий своих продуктов конкретно под UnixWare, в том числе Informix, Netscape, Sybase и Oracle. Увы, в 2001 году IBM отказалась от участия в проекте, и он был прекращен. Теперь, спустя два года, SCO пытается отсудить $1 млрд за то, что IBM якобы использовала конфиденциальные данные, полученные от SCO, в собственных продуктах (AIX 5). А пока суд да дело, Hewlett-Packard и Sun Microsystems пошли своей дорогой, продвигая собственные решения, HP/UX и Solaris. Впрочем, эти компании не делают погоды, поскольку рынок сбыта Solaris узок, внимание компании к платформе Intel пониженное (более 1 млн зарегистрированных пользователей), поэтому доля Sun на платформе Intel оценивается не более чем в 10%. В результате ее маркетинговые акции а-ля «бесплатная Solaris» заканчиваются возвращением в русло коммерции. Ей также требуется перенос всех возможностей со SPARC на IA-64 (на процессорах Intel поддержан только 32-разрядный режим). HP вообще не имеет версии для Intel, что отодвигает сроки ее активного внедрения на рынок. А после нужно будет решать проблему отсутствия приложений. — *Прим. ред.*

вать файловую систему с ведением журналов (Journaling File System, JFS) для UNIX. ULTRIX стала одной из первых систем UNIX с поддержкой многопроцессорной архитектуры.

1.1.7. Mach

Одной из главных причин популярности системы UNIX являлись ее простота и небольшой размер, сочетающиеся со множеством полезных утилит. Но после того как система стала поддерживать все большее количество возможностей, ее ядро постепенно становилось большим, сложным и громоздким. Многие специалисты пришли к мнению, что развитие UNIX постепенно уходит от предполагаемого пути, приведшего когда-то систему к успеху.

В середине 80-х годов разработчики из университета Карнеги—Меллона [8] в Питтсбурге приступили к созданию новой операционной системы под названием *Mach*. Их целью была разработка *микроядра*, включающего небольшой набор утилит, служащих для реализации остальных системных функций пользовательского уровня. Архитектура Mach должна была поддерживать интерфейс программирования UNIX, работать как на однопроцессорных, так и на многопроцессорных системах, а также подходить для распределенных сред. Разработчики надеялись, что, создав новую систему «с нуля», они могут избежать множества проблем, имевшихся в текущих вариантах UNIX.

Одним из первых приближений к реализации задуманных планов стало создание микроядра, в котором были выделены некоторые основные функции, в то время как большинство возможностей системы исходило от набора внешних по отношению к ядру процессов, называемых *серверами*. Система Mach также имела еще одно существенное преимущество: она никак не зависела от лицензий AT&T, что сделало ее привлекательной для многих производителей. Самой популярной версией системы стала *Mach 2.5*, и многие коммерческие ОС, такие как *OSF/1* или *NextStep*, были созданы на ее основе. Ранние версии системы имели монолитные ядра с поддержкой интерфейса 4.4BSD UNIX на высоком уровне. Первой реализацией идеи микроядра стала система Mach 3.0.

1.1.8. Стандарты

Распространенность различных реализаций UNIX привела к появлению проблем совместимости. Несмотря на то что все существующие варианты на первый взгляд «похожи на UNIX», на самом деле они имеют существенные различия между собой. Существование отличий было заложено изначально, за счет наличия двух веток развития UNIX, «официальной» системы AT&T System V и альтернативного варианта BSD, создаваемого в Беркли. Появление коммерческих вариантов UNIX еще более усложнило проблему.

Системы System V и 4BSD существенно отличались. Они имели различные несовместимые между собой файловые системы, реализации поддержки сетей и архитектуры виртуальной памяти. Некоторые различия были обусловлены дизайном ядра систем, но большинство из них находились на уровне программирования интерфейса. Это привело к невозможности создания сложных приложений, работающих без внесения каких-либо изменений в обеих операционных системах.

Все коммерческие варианты UNIX строились на основе либо System V, либо BSD, к которым производители добавляли дополнительные возможности. Именно эти добавления часто оказывались непереносимыми на иные платформы. В результате создатели приложений тратили огромное количество времени и усилий, для того чтобы их программы нормально функционировали в различных реализациях UNIX.

Для решения проблемы необходимо было разработать некий стандартный набор интерфейсов, чем и занялись несколько групп энтузиастов. В результате миру предстало множество стандартов, столь же многочисленных и отличающихся друг от друга, как и существовавшие в те времена варианты UNIX. Однако большинство производителей признала только несколько из созданных стандартов, в том числе «*Определение интерфейса System V*» (System V Interface Definition, SVID) компании AT&T, спецификации организации IEEE под названием *POSIX* и «*Руководство по переносу X/Open*» (X/Open Portability Guide) консорциума X/Open.

В каждом из стандартов описывалось взаимодействие между программами и операционной системой и не затрагивался вопрос реализации самого интерфейса взаимодействия. В них определялись наборы функций и подробно приводились их конструкции. Совместимые системы должны удовлетворять требованиям, изложенным в стандартах, однако реализация необходимых функций могла быть произведена как на уровне ядра, так и на уровне библиотек пользователя.

Стандарты также определяли поднабор функций, предлагаемых большинством систем UNIX. Теоретически, если пользователь будет использовать при написании приложения только те функции, которые входят в этот набор, то созданное приложение будет переносимо на любую систему, совместимую со стандартами. Это заставляло разработчиков программ использовать дополнительные возможности конкретного варианта системы, а также производить оптимизацию своих программ под конкретную аппаратную платформу или операционную систему только в том случае, если их исходные коды легко переносимы.

Стандарт SVID представляет собой подробную спецификацию программного интерфейса System V. Корпорация AT&T выпустила три версии стандарта — *SVID1*, *SVID2* и *SVID3*, описывающие соответственно ОС SVR2, SVR3 и SVR4 [10]. AT&T предоставила возможность производителям систем называть свои продукты System V только в том случае, если они отвечают

требованиям SVID. Корпорация также выпустила пакет *System V Verification Suite* (*SVVS*), который проверял операционные системы на соответствие SVID.

В 1986 году организация IEEE поручила специальному комитету разработать и опубликовать стандарты на среды операционных систем. Для их обозначения было придумано название POSIX (Portable Operating System based on UNIX, что переводится как «Переносимые операционные системы, основанные на UNIX»). Эти документы описывали компоненты ядра систем SVR3 и 4.3BSD. Стандарт *POSIX1003.1*, более известный как *POSIX.1*, был опубликован в 1990 году [11]. Многие производители приняли этот стандарт, так как он не ограничивался каким-то одним вариантом системы UNIX.

X/Open — это международный консорциум производителей компьютерной техники и программного обеспечения. Он был сформирован в 1984 году. Его целью являлась не только разработка новых стандартов, но и создание открытой среды *Common Applications Environment* (Общей программной среды, CAE), базирующейся на **уже** существующих стандартах. Консорциум опубликовал семитомный труд «X/Open Portability Guide» (XPG), последнее (четвертое) издание которого вышло в 1993 году [12]. Материал руководства был основан на стандарте POSIX.1, расширял его и описывал многие дополнительные области, такие как интернационализация, оконные интерфейсы и обработка данных.

1.1.10. OSF и UI

В 1987 году корпорация AT&T, осознавая непринятие общественностью ее лицензионной политики, принимает решение о закупке 20% акций Sun Microsystems. AT&T и Sun решают заняться совместной разработкой SVR4, следующей версии операционной ОС AT&T System V UNIX. Корпорация Sun объявляет, что будущая операционная система станет базироваться на SVR4, в отличие от SunOS, основу которой составляла ранее система System V.

Эти заявления вызвали бурную реакцию со стороны других производителей систем, которые поняли, что созданное объединение даст корпорации Sun огромное преимущество перед остальными производителями. В ответ группа компаний, в которую входили Digital, HP, IBM, Apollo и другие, объявила в 1988 году о создании объединения *Open Software Foundation* (*OSF*). OSF финансировалась компаниями-основателями. Основной задачей организации стала разработка операционной системы, пользовательской и распределенной вычислительной среды, не зависящей от ограничений, накладываемых лицензионными соглашениями AT&T. OSF распространила среди своих членов *Request for Technology* (Запрос на технологии, RFT) и затем выбрала из полученных предложений самые лучшие независимо от того, на кого из производителей работал их автор.

В ответ корпорации AT&T и Sun совместно с другими производителями систем, основанных на System V, в срочном порядке основали свою органи-

зацию, названную *UNIX International* (*UI*). Ее основной целью было продвижение системы SVR4 на рынке, а также выбор дальнейшего направления развития UNIX System V. В 1990 году организация UI выпустила труд под названием *UNIX System V Road Map*, в котором были выделены основные направления будущего развития системы UNIX.

В 1989 году OSF представила графический пользовательский интерфейс *Motif*, положительно встреченный многими пользователями. Позже организация выпустила первую версию своей операционной системы под названием *OSF/1*. Первая версия OSF/1 базировалась на Mach 2.5, имела совместимость с 4.3BSD и обладала некоторыми возможностями IBM AIX. Представленная система имела множество дополнительных возможностей, не поддерживаемых в SVR4, таких как полная поддержка многопроцессорных систем, динамическая загрузка и монтирование томов. В планах членов организации UI была дальнейшая разработка коммерческих операционных систем, базирующихся на OSF/1.

Объединения OSF и UI начинали с весьма высоких целей, но все равно очень быстро столкнулись с общими, не зависящими от них проблемами. Экономический спад начала 90-х, экспансия Microsoft Windows на рынок операционных систем резко уменьшили рост UNIX-систем. Организация UI ушла из компьютерного бизнеса в 1993 году, а объединение OSF было вынуждено расстаться с большинством амбициозных планов (в том числе и планов по созданию Распределенной среды управления, Distributed Management Environment). Одной из основных систем, основанных на OSF, стала DEC OSF/1, созданная компанией Digital в 1993 году. Позже компания приняла решение удалить из этой системы многие возможности, отличающие ее от своей ОС, и в 1995 году изменила имя системы на Digital UNIX.

1.1.10. SVR4 и ее дальнейшее развитие

В 1989 году вышла первая версия совместно разработанной корпорациями AT&T и Sun системы System V Release 4 (SVR4). Эта система объединила в себе возможности SVR3, 4BSD, SunOS и XENIX. В SVR4 также были добавлены новые функции, такие как изменение состава классов в режиме реального времени, командный интерпретатор *Korn shell* и новые возможности подсистемы STREAMS. В следующем году AT&T основала компанию UNIX Systems Laboratories для разработки и продажи систем UNIX.

В 1991 году компания Novell, Inc., создатель сетевой операционной системы NetWare для персональных компьютеров, приобрела часть акций USL и основала совместное предприятие под названием Univel. Целью новой компании стало создание версии SVR4 для настольных систем, интегрированной с ОС NetWare. Такая система была разработана в конце 1992 года и получила название *UnixWare*. После этого было выпущено еще несколько вариантов системы SVR4. Последний вариант, *SVR4.2/ES/MP*, предлагает пользователям расширенную защиту и поддержу многопроцессорных систем.

В 1993 году корпорация AT&T полностью передала USL компании Novell. В следующем году Novell получила права на торговую марку UNIX и подтверждение совместимости своих операционных систем с X/Open. В 1994 году корпорация Sun Microsystems выкупила права на использование кодов SVR4 у Novell, что освободило ее от проблем, связанных с возможным нарушением лицензионных прав и совместимости со стандартами. Система Sun, основанная на SVR4, получила название *Solaris*. Ее последняя версия — это *Solaris 2.5*. Система поддерживает многие дополнительные возможности, такие как собственное многопоточное ядро и поддержка многопроцессорных систем[1].

1.2. Причины изменений системы

Система UNIX сильно преобразилась за годы своего существования. Начав с небольшой операционной среды, использовавшейся группой людей в единственной лаборатории, на сегодняшний день система UNIX стала одной из

[1] ОС Solaris была перенесена на платформу Intel с платформы SPARC. Последние ее версии представляют собой мощные и масштабируемые системы для рабочих станций, младших серверов, корпоративных серверов и суперсерверов. Многие решения Sun не имеют аналогов, технологии компании всегда были «на гребне» (та же Java), а ее системы отличаются высокой надежностью. Но поскольку Sun приходится вести войну на всех фронтах, в первую очередь против Linux и Microsoft, она вынуждена поспевать за конкурентами. В результате производительность Solaris 8 оказывается ниже таковой для Solaris 7, надежность страдает, а проблемам интернационализации (в том числе поддержке русского языка) уделяется недостаточное внимание. За последние 4 года вышли версии для Intel 7 (или Solaris 2.7, SunOS 5.7), 8 и 9 (начало 2003). Solaris 7 работает на одно- и многопроцессорных системах, поддерживает до 4 Гбайт RAM и файловые системы до 1 Тбайт. Файловая система UFS с расширениями от Sun и 64-разрядной адресацией позволяет протоколировать события (по образу журналов). Недостатки низкой производительности NFS сглажены за счет кэширования информации непосредственно на локальном диске (файловая система CacheFS). Solaris 8 Sun противопоставила Windows 2000 Datacenter, а Solaris 9 — Microsoft Windows 2000 Server. Самая главная новация в Solaris 8 — Live Upgrade («горячее» обновление). Она позволяет администраторам устанавливать исправления в ядро и аппаратные средства на серверах SPARC без их перезагрузки. Встроенная кластеризация до 4-х процессоров (для SPARC) позволяет увеличить производительность вебсервера. Solaris 8 поддерживает стандарт IPv6 и спецификацию IPSec (для IPv4), стандарт Mobile IP для мобильных пользователей. Sun объявила о поддержке сервера каталогов LDAP и об отказе от службы NIS, вошедшей в общее пользование с ее же подачи. LDAP интегрирован в Solaris 9, что повышает ее производительность в сравнении с Solaris 8 в 5 раз. В Solaris 8 встроены две графические оболочки — CDE (Common Desktop Environment) и Open Windows, в Solaris 9 — графический интерфейс пользователя Web Start и консоль администрирования Solaris Management Console (SMC). Последняя версия — Solaris 9 — включила в себя множество новых возможностей. Это: протокол сетевой аутентификации Kerberos 5, протокол Secure Shell для безопасного установления связи с UNIX-машинами, поддержка RAID-массивов, расщепление и удвоение дисков, новая организация нитей, повышающая производительность многопроцессорных систем, защита от переполнения буфера — «бич» безопасности для любых систем, основанная на блокировке выполнения стековых операций определенными приложениями. Веб-сервисы, Java-сервер приложений, выполняющий программы е-бизнеса, средства управления предприятиями на основе интернет-технологий WBEM направлены на интеграцию множества компьютеров в один огромный вычислительный комплекс. — *Прим. ред.*

основных операционных систем, предлагаемой в самых различных вариантах многими поставщиками. В настоящее время UNIX используется на самых различных системах, начиная от небольших встроенных контроллеров и заканчивая огромными мэйнфреймами и массивно-параллельными системами. Под управлением UNIX работают самые разнообразные приложения: в офисах UNIX используется как настольная ОС; в финансовых областях с ее помощью обрабатываются крупные базы данных, а научные лаборатории применяют эту систему для сложных математических вычислений.

Изменения и рост UNIX обусловлены, в первую очередь, появлением новых задач, стоявших перед системой. Хотя на данный момент UNIX является целостной операционной системой, это не значит, что она будет оставаться неизменной в дальнейшем. Причиной постоянно происходящих изменений никак нельзя назвать изначально неверный дизайн ОС. Напротив, простота добавления в UNIX новых возможностей по мере развития технологий убедительно доказывает обратное. Не имея точного представления о цели, формах и задачах будущей системы, ее создатели начали работу с построения простых, расширяемых базовых средств, собирая и анализируя предложения по усовершенствованию системы отовсюду, от научных учреждений, коммерческих предприятий и простых пользователей.

Проведем анализ основных факторов, повлиявших на рост и совершенствование системы. В этом разделе вы увидите не только описание самих факторов, но и соображения автора относительно предполагаемых трансформаций системы UNIX в будущем.

1.2.1. Функциональные возможности

Главной причиной, подталкивающей к изменениям, является необходимость добавления новых возможностей в систему. Изначально новые функциональные средства появлялись в UNIX только как пользовательские инструменты и утилиты. Позже, когда UNIX превратилась во вполне развитую систему, разработчики стали добавлять многие дополнительные возможности прямо в ядро ОС.

Многие из новых функций системы создавались для поддержки более сложных программ. Одним из примеров таких нововведений является набор Interprocess Communication (IPC) в ОС System V, в состав которого входят поддержка разделения памяти, семафоры и очереди сообщений. Все эти возможности позволили процессам взаимодействовать, используя совместно данные, обмениваясь сообщениями и синхронизируя свои действия. Большинство современных систем UNIX также имеют несколько уровней поддержки для создания многопоточных приложений.

Возможности IPC и технологии потоков существенно помогают в разработке сложных приложений, например тех, что основаны на модели «клиент-сервер». В таких программах обычно серверная часть находится в режиме

постоянного ожидания запроса от клиентов. Если такой запрос приходит, сервер обрабатывает его и снова переходит в режим ожидания следующего. Если сервер имеет возможность обслуживания сразу несколько клиентских запросов, в этом случае предпочтительно вести их обработку параллельно. С применением технологии IPC сервер может использовать отдельный процесс для обработки каждого запроса, в то время как все выполняемые процессы могут совместно использовать одни и те же данные. Многопоточная система позволяет реализовать сервер как один процесс, имеющий несколько параллельно функционирующих потоков, использующих общее адресное пространство.

Возможно, наиболее «видимой» частью любой операционной системы является файловая система. В ОС UNIX также было добавлено множество новых возможностей, в том числе поддержка файлов *FIFO* (First In, First Out), символьных связей, а также файлов, имеющих размеры большие, чем раздел диска. Современные системы поддерживают защиту файлов, списки прав доступа, а также ограничения доступа к дискам для каждого пользователя.

1.2.2. Сетевая поддержка

За годы развития UNIX максимальным изменениям была подвергнута часть ядра, являющаяся сетевой подсистемой. Ранние версии ОС работали отдельно друг от друга и не имели возможности соединения с другими машинами. Однако распространение компьютерных сетей поставило перед разработчиками проблему необходимости их поддержки в системе UNIX. Первым занялся решением этой проблемы университет Беркли. Организация DARPA профинансировала проект встраивания поддержки TCP/IP в 4BSD. На сегодняшний день системы UNIX поддерживают большое количество сетевых интерфейсов (таких как Ethernet, FDDI и ATM), протоколов (TCP/IP, UDP/IP[1], SNA[2] и других) и средств (например, сокетов и STREAMS).

Появление возможности соединения с другими компьютерами во многих отношениях повлияло на операционную систему. Пользователи ОС сразу же захотели совместно использовать файлы, расположенные на соединенных между собой машинах, а также запускать приложения на удаленных узлах. Для удовлетворения этих требований развитие системы UNIX велось в трех направлениях:

- Разрабатывались распределенные файловые системы, позволявшие вести прозрачный доступ к файлам на удаленных узлах. Наиболее удачными из созданных систем оказались *Network File System* (*NFS*) корпорации Sun Microsystems, *Andrew File System* (*AFS*) университета Карнеги—Меллона и *Distributed File System* (*DFS*) корпорации Transarc.

[1] User Datagram Protocol/Internet Protocol, протокол пользовательских дейтаграмм/протокол Интернета.

[2] System Network Architecture (системная сетевая архитектура) компании IBM.

- Создавалось большое количество распределенных служб, позволяющих совместно использовать информацию по сети. Эти службы представляют собой обычные пользовательские программы, основанные на модели «клиент-сервер» и использующие удаленные вызовы процедур для активации действий на других компьютерах. Примерами таких программ являются *Network Information Service* (Сетевой информационный сервис, NIS) и *Distributed Computing Environment* (Распределенная вычислительная среда, DCE).
- Появлялись распределенные операционные системы, такие как *Mach*, *Chorus* и *Sprite*, имеющие различную степень совместимости с UNIX и продвигаемые на рынок как базовые технологии для построения будущих распределенных ОС.

1.2.3. Производительность

Постоянной движущей силой, заставляющей вносить изменения в системы, является увеличение их производительности. Конкурирующие между собой поставщики операционных систем тратят огромные усилия на демонстрацию того, что именно их ОС производительнее, чем другие. Почти все внутренние подсистемы претерпели большие изменения, выполненные с целью увеличения производительности систем.

В начале 90-х годов университет Беркли представил файловую систему Fast File System, благодаря интеллектуальным правилам размещения блоков на диске увеличивающую производительность системы. Позже появились более быстродействующие файловые системы, использующие внешнее размещение или технологии поддержки журналов. Увеличение вычислительных мощностей также стало основной причиной разработок в области коммуникаций между процессами, работы с памятью и многопоточных процессов. Когда для работы многих приложений оказалось недостаточно одного процессора, производители разработали многопроцессорные системы под управлением UNIX, некоторые из которых имеют сотни процессоров.

1.2.4. Изменение аппаратных платформ

Операционная система должна «шагать в ногу» с современными аппаратными технологиями. Часто это означает необходимость ее переноса на новые и более высокопроизводительные процессоры. После того как ядро UNIX было почти полностью переписано на языке C, решение этой задачи стало относительно легким. В прошлом разработчики тратили огромные усилия на отделение от общего кода программы участков, написанных для конкретного аппаратного оборудования, и включение их в отдельные модули. После проведения этих действий для переноса системы требовалось переписывать заново только отдельные модули. Обычно такие модули отвечали за обработку пре-

рываний, трансляцию виртуальных адресов, переключение контекстов и драйверы устройств.

В большинстве случаев для выживания операционной системы на рынке необходим ее постоянный перенос на вновь появляющееся оборудование. Наиболее очевидной возможностью, которую обязательно желают видеть в UNIX, является поддержка многопроцессорных систем. Ядро традиционного варианта UNIX разрабатывалось для работы на одном процессоре и не имело возможности защиты структур данных при параллельном доступе к ним одновременно нескольких процессоров. Однако позже сразу несколько производителей разработали многопроцессорные реализации системы UNIX. Большинство их них использовали традиционное ядро UNIX и добавляли собственные элементы защиты, обеспечивающие безопасность общих структур данных. Это средство называется параллелизацией. Малая часть производителей занялась построением собственного ядра на иных основах.

При более глубоком рассмотрении проблемы видно, что неравномерность развития различных аппаратных технологий оказала глубокое влияние на разработку операционных систем. С тех времен, как была создана первая версия UNIX для PDP-7, средняя скорость процессоров увеличилась примерно в 100 раз[1]. Объемы памяти и дискового пространства, выделяемого для одного пользователя, возросли более чем в 20 раз. С другой стороны, скорость доступа к памяти и дискам увеличилась всего лишь в 2 раза.

В 70-х годах производительность систем UNIX ограничивалась скоростью работы процессора и размером памяти. Но вскоре в ядре системы стали использоваться такие технологии, как свопинг и, чуть позже, страничная организация памяти (технология, позволявшая выполнять большое количество процессов на малых объемах памяти). По мере развития вычислительной техники скорость доступа к памяти и процессору стала играть меньшую роль, а сама система занималась в большей степени вводом-выводом, основное время занимаясь переносом страниц между дисками и оперативной памятью. Это являлось важной причиной появления новых разработок в области файловых систем, хранения информации и архитектур виртуальной памяти, целью которых была оптимизация дисковых процедур. Именно для этого были разработаны *Redundant Arrays of Inexpensive Disks* (массивы недорогих дисков с избыточностью, RAID) и происходило быстрое распространение структурированных файловых систем.

1.2.5. Улучшение качества

Все преимущества функциональности и скорости работы можно запросто свести на нет, если в систему заложены ошибки. Разработчики вносили мно-

[1] Частота процессора Intel 4004 в первом миникомпьютере (1971 г.) составляла 1 МГц. Соответственно на 2002 год приходится говорить о росте производительности в 1000 раз. — *Прим. ред.*

жество изменений в дизайн систем, чтобы сделать их более устойчивыми, стараясь добиться увеличения надежности программного обеспечения.

Изначальный механизм оповещения был ненадежен и неэффективен по многим причинам. Однако позже его реализация была пересмотрена (сначала разработчиками из Беркли, затем — AT&T), и в результате появилась новая, более устойчивая система оповещения, получившая название *надежных сигналов*.

Система BSD, точно так же, как и System V, не была застрахована от возможных отказов. В системах UNIX перед записью на диск данные хранятся некоторое время в памяти. Следовательно, существует потенциальная возможность их потери в случае отказа, а также нарушения целостности файловой системы. В различных вариантах UNIX предлагается стандартная утилита fsck(8), проверяющая и восстанавливающая поврежденные файловые системы. Эта операция занимает ощутимое количество времени и может длиться десятки минут на крупных серверах, имеющих диски больших объемов. Многие современные системы UNIX поддерживают файловые системы, использующие технологию поддержки журналов, что увеличивает доступность и стабильность системы и устраняет потребность в применении fsck.

1.2.6. Глобальные изменения

За последние три десятилетия произошли огромные изменения в принципах использования компьютеров. В 70-х годах вычислительная система представляла большой централизованный компьютер размером с комнату, поддерживающий работу пользователей, которые подключались к нему при помощи терминалов. Применялись системы разделения времени, в которых центральный компьютер разделял процессорные ресурсы среди пользователей. Терминалы представляли собой простые устройства, умеющие чуть больше, чем вывод данных в текстовом режиме.

В 80-х годах началась эра *рабочих станций*, оборудованных высокоскоростными графическими дисплеями, имеющими возможность вывода информации в нескольких окнах, в каждом из которых выполняется оболочка UNIX. Рабочие станции идеальны для интерактивного использования и имеют достаточную вычислительную мощность для работы с обычными пользовательскими приложениями. На рабочих станциях обычно в один момент времени работает один пользователь, но они могут поддерживать многих пользователей. Появление высокоскоростных сетей позволило рабочим станциям соединяться между собой, а также с другими компьютерами.

Позже появилась новая модель вычислений, называемая *клиент-серверными вычислениями*, в которой один или более мощных централизованных компьютеров, называемых *серверами*, предоставляют различные службы индивидуальным рабочим станциям, или *клиентам*. Для хранения файлов пользователей стали применяться *файловые серверы*. *Серверы приложений* — это

компьютеры, оснащенные одним или несколькими мощными процессорами, на которых пользователи могут выполнять задачи, требующие большого объема вычислений (например, решение математических уравнений). На *серверах баз данных* выполняется специальная программа, обрабатывающая запросы к базам данных, поступающие от клиентов. Обычно серверы представляют собой мощные высокопроизводительные машины с быстродействующими процессорами и большими объемами оперативной и дисковой памяти. Клиентские рабочие станции имеют меньшую производительность, размеры памяти и объемы дисков, но они, как правило, оснащаются высококачественными мониторами и предоставляют пользователю большие интерактивные возможности.

Постепенно рабочие станции становились все более производительными, различия между клиентами и серверами все больше стирались. Более того, централизованное выполнение необходимых служб на небольшом количестве серверов приводило к перегрузкам сетей и самих серверов. Результатом стало изменение подхода к построению компьютерных систем и появление новой технологии *распределенных вычислений*. При использовании этой модели компьютеры совместно предоставляют какую-либо сетевую службу. Каждый узел сети может иметь собственную файловую систему и предоставлять доступ к ней для других узлов. В результате узел функционирует как сервер по отношению к локальным файлам и как клиент по отношению к файлам, хранящимся на других узлах. Такой подход уменьшает перегрузки сети, а также количество сбоев в ее работе.

Система UNIX была адаптирована для различных моделей вычислений. Например, ранние варианты системы поддерживали только локальную файловую систему. Поддержка сетевых протоколов привела к разработке распределенных файловых систем. Некоторые из них, такие как AFS, требовали специализированных серверов. Позже файловые системы были переработаны и стали распределенными, что дало возможность каждому компьютеру в сети выступать в роли клиента и сервера.

1.2.7. Поддержка различных приложений

Система UNIX изначально разрабатывалась для применения в простых средах разделения времени, таких как исследовательские лаборатории или университеты. Системы позволяли некоторому количеству пользователей выполнять несложные программы обработки и редактирования текстов и математических вычислений. После того как UNIX обрела популярность, ею стали пользоваться для более широкого спектра приложений. К началу 90-х годов UNIX использовалась в физических и космических лабораториях, мультимедийных рабочих станциях для обработки звука и видео, а также во встроенных контроллерах, использующихся в критически важных системах.

Каждое приложение обычно имеет различные требования к системе, что подхлестнуло разработчиков систем изменять их в соответствии с этими требованиями. Мультимедийные и встроенные приложения требуют гарантий доступности ресурсов и определенной скорости отклика на запросы. Научные приложения требуют одновременной работы нескольких процессоров. Для реализации этих требований в некоторых системах UNIX появились возможности работы в режиме реального времени, такие как процессы с фиксированным приоритетом, разграничение использования процессоров и возможность сохранения данных в памяти.

1.2.8. Чем меньше, тем лучше

Одним из преимуществ оригинального дизайна системы UNIX был ее небольшой размер, простота и ограниченный набор основных функций. Изначальным подходом к системам было предоставление простых инструментов, которые можно гибко комбинировать между собой, используя такие инструменты, как конвейер (pipe). Однако ядро традиционного варианта системы было достаточно монолитным, следовательно, его расширение представляло собой непростую задачу. Чем больше функций добавлялось в ядро, тем оно все больше разрасталось и усложнялось, все дальше уходя от его первоначального размера, не превышающего 100 Кбайт, постепенно достигнув объема в несколько мегабайтов. Объемы памяти компьютеров стали увеличиваться, вместе с тем поставщики систем и их пользователи игнорировали этот факт. Но именно он сделал UNIX менее пригодной для использования на небольших персональных компьютерах и портативных системах.

Многие стали понимать, что такое изменение системы не предвещает ничего хорошего, так как она становится слишком большой, перенасыщенной и неорганизованной. Были потрачены большие усилия на переработку системы или на написание нового варианта, который основывался бы на оригинальной философии UNIX, но имел бы большую расширяемость и модульность. Наиболее удачной реализацией такой системы стала Mach, которая выступила основой для последующих коммерческих ОС, примерами которых являются OSF/1 и NextStep. Система Mach впоследствии использовала архитектуру микроядра (см. раздел 1.1.7), в которой небольшое ядро предоставляет средство для выполнения программ, а серверные задачи на пользовательской уровне предоставляют все остальные функции.

Не все попытки контроля размера ядра имели успех. К сожалению, микроядра не могли иметь производительность, сравнимую со скоростью работы традиционного монолитного ядра, в первую очередь, по причине издержек, накладываемых передачей сообщений. Менее амбициозные проекты, такие как модульность, ядра со страничной поддержкой и динамическая загрузка, имели больший успех, так как позволяли загружать или выгружать из памяти компоненты ядра по мере надобности.

1.2.9. Гибкость системы

В 70-х и начале 80-х годов ядро систем UNIX было недостаточно гибким. Ядро поддерживало только один тип файловой системы, набор алгоритмов планирования, а также формат выполняемых файлов (рис. 1.1). ОС имела некоторую степень гибкости только по отношению к переключателям символьных и блочных устройств, позволяя различным типам устройств иметь доступ к системе через общий интерфейс. Развитие распределенных систем в середине 80-х годов стало очевидной причиной для того, чтобы UNIX стала поддерживать как удаленные, так и локальные файловые системы. А такие возможности, как разделяемые библиотеки, потребовали поддержки различных форматов выполняемых файлов. Система UNIX стала поддерживать эти форматы, оставив традиционный a.out для совместимости. Одновременное сосуществование мультимедийных приложений и программ, функционирующих в режиме реального времени, потребовало поддержки планирования для различных типов приложений.

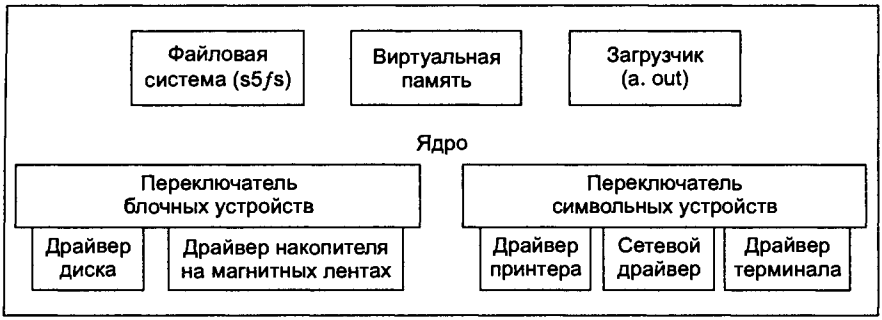


Рис. 1.1. Традиционное ядро UNIX

В итоге UNIX превращалась в более гибкую операционную систему, которая бы поддерживала несколько различных методов выполнения одной и той же задачи. Это потребовало разработки многих более гибких структур, таких как интерфейс vnode/vfs, системный вызов exec, планирование процессов, а также сегментную архитектуру памяти. Современное ядро UNIX похоже на систему, показанную на рис. 1.2. В каждом из внешних кругов представлен интерфейс, который может быть реализован различными способами.

1.3. Оглянемся назад, посмотрим вперед

UNIX заметно изменилась после своего первого представления. Несмотря на достаточно скромный «выход в свет» она завоевала большую популярность. Во второй половине 80-х годов операционную систему UNIX выбрали многие коммерческие организации, учебные заведения и научно-исследователь-

ские лаборатории. Чаще всего результатом приобретения того или иного варианта системы был осознанный выбор. Позже позиции UNIX были потеснены корпорацией Microsoft, предлагающей операционные системы семейства Windows. Постепенно ОС UNIX стала проигрывать в битве за рынок настольных систем. В этом разделе мы проведем анализ причин ее успеха и популярности, а также факты, не позволившие UNIX покорить мир.

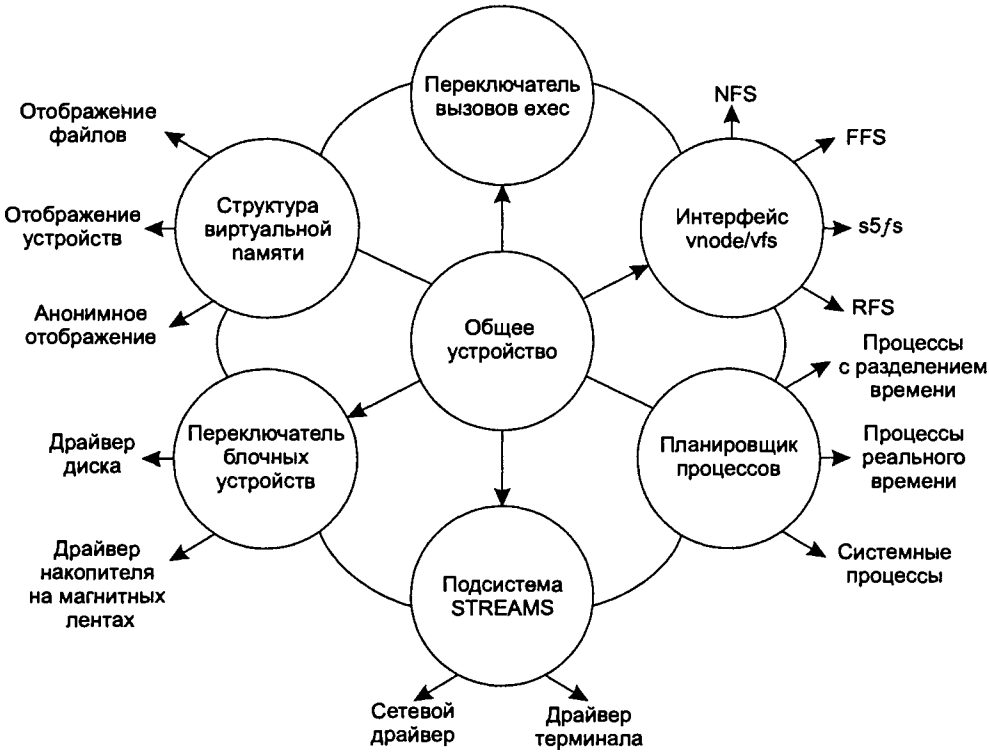


Рис. 1.2. Современное ядро UNIX

1.3.1. Преимущества UNIX

UNIX завоевала такую высокую степень популярности, на которую, возможно, даже не рассчитывали ее разработчики. Одной из главных причин успеха являлся способ распространения системы. Корпорация AT&T, ограниченная законами в своих возможностях, продавала лицензии и исходные коды системы по достаточно низкой цене, поэтому UNIX стала популярной среди многих пользователей по всему миру. Так как в комплект поставки входили и исходные коды, пользователи имели возможность экспериментировать с ними, улучшать их, а также обмениваться друг с другом созданными изменениями. Корпорация AT&T встраивала многие из новшеств в следующие версии системы.

Разработчики из Беркли также поддерживали эту традицию. Развитие системы UNIX оказалось очень открытым процессом. Добавления поступали из учебных заведений, коммерческих организаций и от хакеров-энтузиастов из разных континентов и стран мира. Даже после коммерциализации UNIX многие производители ОС поддержали концепцию открытых систем и сделали свои разработки доступными для других, создавая спецификации открытых систем, такие как NFS.

Оригинальная версия UNIX имела хороший дизайн, который являлся базисом последующего успеха более поздних реализаций и вариантов системы. Одну из сильнейших сторон системы можно охарактеризовать выражением «красота в краткости» [12]. Ядро небольшого размера имело минимальный набор основных служб. Утилиты небольшого размера производили малый объем манипуляций с данными. Механизм *конвейера* совместно с программируемой оболочкой позволял пользователям комбинировать эти утилиты различными способами, создавая мощные, производительные инструменты.

Файловая система UNIX является примером вышеописанного подхода. В отличие от других операционных систем, обладающих сложными методами доступа к файлам, такими как *Indexed Sequential Access Method* (Индексированный последовательный доступ к файлам, ISAM) или *Hierarchical Sequential Access Method* (Иерархический последовательный доступ к файлам, ISAP), система UNIX интерпретирует файлы как простую последовательность байтов. Приложения могут записывать в содержимое файлов любую структуру и использовать различные собственные методы доступа, не ставя об этом в известность операционную систему.

Многие системные приложения используют для представления своих данных символьные строки. К примеру, важнейшие базы данных, такие как `/etc/passwd`, `/etc/fstab` и `/etc/ttys`, являются обычными текстовыми файлами. Возможно, что структурированное хранение информации в двоичном формате представляется более эффективным, однако представление в виде текста позволило пользователям легко просматривать и производить манипуляции с этими файлами без применения специальных инструментов. Текст является общей, универсальной и обладающей высокой степенью переносимости формой данных, которые можно легко обрабатывать множеством различных утилит.

Еще одной особенностью системы UNIX стал простой унифицированный интерфейс с устройствами ввода-вывода. Представляя все устройства как файлы, система позволяет пользователям применять один и тот же набор команд и системных вызовов для доступа и работы с различными устройствами, равно как и для работы с файлами. Разработчики могут создавать программы, производящие ввод-вывод, не заботясь, с чем именно производится обмен, с файлом, терминалом пользователя, принтером или другим устройством. Таким образом, используемый совместно с перенаправлением данных интерфейс ввода-вывода системы UNIX — простой и одновременно мощный.

Причиной успеха и распространения UNIX стала ее высокая степень переносимости. Большая часть ядра системы написана на языке C. Это по-

зволило относительно легко адаптировать UNIX к новым аппаратным платформам. Первая реализация системы появилась на популярной тогда машине PDP-11 и затем была перенесена на VAX-11, имевшую не меньшую популярность. Многие производители «железа» после создания новых компьютеров имеют возможность просто перенести на них уже имеющуюся систему UNIX вместо того, чтобы создавать для своих разработок операционную систему заново.

1.3.2. Недостатки UNIX

Известно, что любая медаль имеет две стороны. Рассказав о преимуществах системы UNIX, необходимо привести и ее недостатки. Один из наиболее объективных обзоров UNIX был создан не кем иным, как Денисом Ритчи. В январе 1987 года на конференции USENIX в разделе «Ретроспективы UNIX» Ритчи сделал доклад, где провел анализ многих недостатков системы [13], которые кратко описаны ниже.

Хотя UNIX изначально была весьма простой системой, это вскоре закончилось. Например, AT&T добавила в стандартную библиотеку ввода-вывода буферизацию данных, что повысило ее эффективность и сделало программы переносимыми на не-UNIX-системы. Библиотека разрасталась и стала более сложной, чем системный вызов, лежащий в ее основе. Системные вызовы `read` и `write` были неделимыми операциями над файлами, в то время как буферизация библиотеки ввода-вывода уничтожила эту цельность.

UNIX сама по себе являлась замечательной операционной системой, однако большинству пользователей нужна была не сама система, а, в первую очередь, возможность выполнять определенное приложение. Пользователей не интересовала элегантность структуры файловой системы или модели вычислений. Они хотели работать с определенными программами (например, текстовыми редакторами, финансовыми пакетами или программами для создания изображений), потратив на это минимум расходов и усилий. Недостатки простого унифицированного (обычно графического) пользовательского интерфейса в первых системах UNIX были основной причиной его неприятия массами. Как сказал Ритчи: «UNIX является простой и понятной системой, но чтобы понять и принять ее простоту, требуется гений (или, как минимум, программист)».

Получилось так, что элегантность и эстетичность, свойственная UNIX, требует от пользователей, желающих эффективно работать в системе, творческого мышления и определенной изобретательности. Однако большинство пользователей предпочитают простые в изучении интегрированные многофункциональные программы, подобные тем, что применяются на персональных компьютерах.

В какой-то степени система UNIX явилась жертвой своего собственного успеха. Простота лицензионных условий и переноса на различные аппарат-

ные платформы стала причиной неконтролируемого роста и лавинообразного распространения различных реализаций ОС. Каждый пользователь имел право вносить свои собственные изменения в систему, в результате группы разработчиков часто создавали несовместимые между собой варианты. Изначально существовали две основные ветви развития UNIX, разрабатываемые компаниями AT&T и BSD. Каждая реализация имела оригинальную файловую систему, архитектуру памяти, сигналы и принципы работы с терминалами. Позже другие поставщики предложили новые варианты UNIX, стараясь привести их к некоторой степени совместимости с реализациями AT&T и BSD. Однако чем дальше, тем все менее предсказуемой становилась ситуация, а разработчикам приложений требовалось все больше усилий, чтобы приспособить свои программы ко всем различным вариантам UNIX.

Стандартизация систем стала лишь частичным решением проблемы, так как встретила определенное сопротивление. Поставщики стремились добавить в свои разработки какие-либо уникальные функции, стараясь создать продукт, имеющий отличия от остальных, и тем самых показать его преимущества среди конкурирующих вариантов.

Ричард Рашид (Richard Rashid), один из разработчиков системы Mach, предложил свою версию причин неудач UNIX. Во вступлении к курсу лекций о системе Mach (Mach Lecture Series [16]) Рашид сказал, что причиной создания ОС Mach стало наблюдение за эволюционированием системы UNIX, которая имела минимальные возможности для построения инструментов пользователя. Большие сложные инструменты создавались путем комбинирования множества простых функций. Однако такой подход не был перенесен на ядро системы.

Традиционное ядро UNIX было недостаточно гибким и расширяемым, оно имело минимальные возможности для дополнительного использования кода. Позже разработчики стали просто добавлять новые коды в ядро системы, делая его основой для новых функциональных средств. Ядро очень быстро стало раздутым, сложным и абсолютно немодульным. Разработчики Mach попытались решить эти проблемы, переписав систему заново с нуля, взяв за основу небольшое количество основных функций. В современных реализациях UNIX вышеописанная система решается различными способами, например добавляются гибкие структуры к подсистемам, как это было описано в разделе 1.2.9.

1.4. Границы повествования книги

Эта книга описывает современные системы UNIX. С целью полноты описания и передачи определенного исторического контекста приводится краткий рассказ о возможностях ранних реализаций ОС. В настоящее время существует множество вариантов системы UNIX, каждая из которых по-своему уникальна. Мы постарались разделить все системы на два типа: базовые и коммерческие

варианты. Базовые ОС включают в себя System V, 4BSD и Mach. Другие варианты происходят от одной из базовых ОС и содержат различные дополнительные возможности и расширения, созданные их разработчиками. Список таких систем включает в себя SunOS и Solaris 2.x корпорации Sun Microsystems, AIX компании IBM, HP-UX от Hewlett-Packard, а также ULTRIX и Digital UNIX от корпорации Digital.

Эта книга не специализируется на каком-то специфическом варианте или реализации системы UNIX. Вместо этого здесь проводится анализ важнейших разработок, их архитектуры и подхода к решению многих проблем. Книга уделяет внимание прежде всего системе SVR4, но вы можете найти здесь достаточно подробный рассказ о 4.3BSD, 4.4BSD и Mach. При рассказе о коммерческих вариантах ОС максимальное внимание уделяется SunOS и Solaris 2.x. Причиной повышенного акцента является не только успех систем компании Sun Microsystems на рынке ОС, но прежде всего то, что именно эта компания была разработчиком многих технических решений, интегрированных в базовые варианты систем, а также то, что компания создала большое количество книг по своим ОС.

В тексте книги часто приводятся общие ссылки на традиционный или коммерческий варианты системы UNIX. Под традиционными вариантами мы имеем в виду SVR3, 4.3BSD и их более ранние реализации. Часто в тексте книги встречаются фразы, касающиеся каких-либо возможностей традиционных систем (например, «в традиционных системах UNIX поддерживался один тип файловой системы»). Несмотря на то что между SVR3 и 4.3BSD имеются различия в каждой подсистеме, между ними есть и много общего. Общие фразы типа приведенной выше обращают внимание как раз на такие свойства систем. При рассмотрении современных ОС UNIX мы подразумеваем системы SVR4, 4.4BSD, Mach, а также реализации, основанные на них. Таким образом, общий комментарий, наподобие «современные системы UNIX поддерживают какую-либо реализацию журнальной файловой системы», означает, что такая возможность имеется во многих современных системах, но необязательно во всех из них.

1.5. Дополнительная литература

1. Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and Young, M, «Mach: A New Kernel Foundation for UNIX Development», Proceedings of the Summer 1986 USENIX Technical Conference, Jun. 1986, pp. 93–112[1].

[1] Все публикации прошлых лет в рамках конференции USENIX доступны на сайте http://www.usenix.org в формате PDF (кроме последних нескольких месяцев). Там же размещена полная библиография и имеется возможность поиска по автору, дате и ключевым словам. — *Прим. ред.*

2. Allman, E. «UNIX: The Data Forms», Proceedings of the Winter 1987 USENIX Technical Conference, Jan. 1987, pp. 9–15.
3. American Telephone and Telegraph, «The System V Interface Definition (SV1D)», Third Edition, 1989.
4. Bostic, K., «4.4BSD Release», Vol. 18, No. 5, Sep.–Oct. 1993, pp. 29–31.
5. Bourne, S., «The UNIX Shell», The Bell System Technical Journal, Vol. 57, No. 6, Part 2, Jul.–Aug. 1978, pp. 1971–1990.
6. Gerber, C. «USL Vs. Berkeley», UNIX Review, Vol. 10, No. 11, Nov. 1992, pp. 33–36.
7. Institute for Electrical and Electronic Engineers, Information Technology, «Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C language]», 1003.1–1990, IEEE, Dec. 1990.
8. Joy, W. N., Fabry, R. S., Leffler, S. J., McKusick, M. K., and Karels, M. J., «An Introduction to the C Shell», UNIX User's Supplementary Documents, 4.3 Berkeley.
9. «Software Distribution», Virtual VAX-11 Version, USENIX Association, 1986, pp. 41–46.
10. Organick, E. J., «The Multics System: An Examination of Its Structure», The MIT Press, Cambridge, MA, 1972.
11. Rashid, R. F., «Mach: Technical Innovations, Key Ideas, Status», Mach 2.5 Lecture Series, OSF Research Institute, 1989.
12. Richards, M., and Whitby-Strevens, C., «BCPL: The Language and Its Compiler», Cambridge University Press, Cambridge, UK, 1982.
13. Ritchie, D. M., and Thompson, K., «The UNIX Time-Sharing System», The Bell System Technical Journal, Vol. 57, No. 6, Part 2, pp. 1905–1930, Jul.–Aug. 1978.
14. Ritchie, D. M., «Unix: A Dialectic», Proceedings of the Winter 1987 USENIX Technical Conference, Jan. 1987, pp. 29–34.
15. Salus, P. H., «A Quarter Century of UNIX», Addison-Wesley, Reading, MA, 1994.
16. Thompson, K., and Ritchie, D. M., «The UNIX Time-Sharing System», Communications of the ACM, Vol. 17, No. 7, M. 1974, pp. 365–375.
17. «The X/OPEN Portability Guide (XPG)», Issue 4, Prentice-Hall, Englewood Cliffs, NJ, 1993.

Глава 2
Ядро и процессы

2.1. Введение

Основной функцией операционной системы является предоставление среды, под управлением которой могут выполняться пользовательские программы (также называемые *приложениями*). Система определяет базовую структуру для выполнения программ, а также предлагает набор различных служб, например таких, как операции работы с файлами или ввод-вывод, и предоставляет интерфейс взаимодействия с ними. Интерфейс программирования системы UNIX весьма гибок и богат своими возможностями [4], он может эффективно поддерживать широкий спектр приложений. В этой главе описываются основные компоненты систем UNIX, а также их взаимодействие между собой, предоставляющее пользователям мощное средство программирования.

Существует несколько вариантов системы UNIX. Часть из них — это различные реализации System V компании AT&T (на сегодняшний день последняя версия System V под названием SVR4 является собственностью корпорации Novell), реализации системы BSD Калифорнийского университета в Беркли, OSF/1 организации Open Software Foundation, а также SunOS и Solaris, поставляемые компанией Sun Microsystems. В этой главе описывается архитектура ядра и процессов традиционных систем UNIX, то есть систем, базирующихся на SVR2 [3], SVR3 [2], 4.3BSD [5], и их более ранних версий. Современные варианты UNIX, такие как SVR4, OSF/1, 4.4BSD и Solaris 2.x, значительно отличаются от базовой модели, их архитектура будет подробно описана в следующих главах книги.

Среда приложений системы UNIX основана на фундаментальной абстракции — *процессе*. В традиционных системах процесс выполняет единую последовательность инструкций в *адресном пространстве*. Адресное пространство процесса представляет собой набор адресов памяти, к которым тот имеет доступ и на которые может ссылаться. Процесс отслеживает последовательность выполняемых инструкций при помощи *контрольной точки*, используя аппаратный регистр, обычно называемый *указателем (счетчиком) команд*. Более поздние варианты UNIX поддерживают сразу несколько контрольных точек и, следовательно, несколько параллельно выполняемых последовательностей инструкций в одном процессе, называемых *нитями*.

Система UNIX является многозадачной. Это означает, что в ней одновременно функционируют несколько процессов. Для этих процессов система обеспечивает некоторые свойства *виртуальной машины*. В классической архитектуре виртуальной машины операционная система создает каждому процессу иллюзию того, что он является единственной задачей, выполняемой в данное время. Программист пишет приложение так, как будто только его код будет выполняться системой. В системах UNIX каждый процесс имеет собственные регистры и память, однако для операций ввода-вывода и взаимодействия с устройствами должен полагаться на операционную систему.

Адресное пространство процесса является виртуальным[1], и обычно только часть его соответствует участкам в физической памяти. Ядро хранит содержимое адресного пространства процесса в различных объектах хранения, в том числе в физической памяти, файлах на диске, а также в специально зарезервированных *областях свопинга* (swap areas), находящихся на локальных или удаленных дисках. Подсистема управления памятью ядра переключает *страницы* (блоки фиксированного размера) памяти процесса между этими объектами по мере необходимости.

Каждый процесс также имеет набор регистров, которые соответствуют реальным аппаратным регистрам. В системе может быть одновременно активно множество процессов, но набор аппаратных регистров только один. Ядро хранит регистры процесса, выполняющегося в текущий момент времени в аппаратных регистрах, и сохраняет регистры остальных процессов в специальных структурах данных, отводимых для каждого процесса.

Процессы соперничают между собой, пытаясь захватить различные ресурсы системы, такие как процессор (также называемый *CPU* или центральным процессором), память и периферийные устройства. Операционная система должна функционировать как диспетчер ресурсов, распределяя их оптимально. Процесс, не имеющий возможности получить необходимый ресурс, должен *блокироваться* системой (его выполнение приостанавливается) до тех пор, пока ресурс снова не станет доступен. Процессор является одним из таких ресурсов, поэтому на однопроцессорной системе только один процесс может по-настоящему выполняться в данный момент времени. При этом остальные блокируются, переходя в режим ожидания освобождения процессора или иных ресурсов. Ядро системы дает иллюзию одновременной работы, предоставляя процессам возможность пользоваться процессором в течение определенного короткого промежутка времени, называемого *квантом* и составляющего обычно около 10 миллисекунд. По истечении этого времени ресурсы процессора передаются следующему процессу. Таким образом, каждый про-

[1] Существует несколько вариантов системы UNIX, не использующих виртуальную память. Это самые ранние реализации UNIX (первые ОС, поддерживающие виртуальную память, появились в конце 70-х годов, см. раздел 1.1.4) и некоторые версии, работающие в режиме реального времени. В этой книге описываются только системы, поддерживающие виртуальную память.

цесс получает часть процессорного времени, в течение которого работает. Такая модель функционирования получила название *квантования времени* (time-slicing).

С другой точки зрения, компьютер предоставляет пользователю различные устройства, такие как процессор, диски, терминалы и принтеры. Разработчикам приложений нет необходимости вникать в детали функционирования и архитектурные особенности этих компонентов на низком уровне. Операционная система берет на себя полное управление этими устройствами и предоставляет высокоуровневый, абстрактный программный интерфейс, которым приложения могут пользоваться для доступа к аппаратным компонентам. Система скрывает все детали, связанные с оборудованием, сильно упрощая тем самым работу программиста[1]. Централизуя все управление устройствами, система также предоставляет дополнительные возможности, такие как синхронизация доступа (в тех случаях, когда два пользователя в один момент времени попытаются воспользоваться одним и тем же устройством) и устранение ошибок. Семантика любого взаимодействия между приложениями и операционной системой определяется *прикладным интерфейсом программирования* (*API*).

Мы уже стали относиться к операционной системе как к некой сущности, которая *делает нечто*. Что же в точности представляет собой эта сущность? С одной стороны, операционная система — это программа (часто называемая *ядром*), которая управляет аппаратурой, создает, уничтожает все процессы и управляет ими (рис. 2.1). Если рассматривать шире, операционная система не только включает в себя ядро, но является также основой функционирования остальных программ и утилит (командных интерпретаторов, редакторов, компиляторов, программ типа `date`, `ls`, `who` и т. д.), составляющих вместе пригодную для работы среду. Ядро само по себе мало пригодно для использования. Пользователи, приобретающие систему UNIX, ожидают получить вместе с ней большой набор дополнительных программ. Однако ядро, тем не менее, является весьма специфичной программой по многим причинам. Оно определяет программный интерфейс системы. Ядро — это единственная программа, являющаяся необходимой, без которой ничего не будет работать. Эта книга посвящена изучению ядра системы, и когда будет упоминаться *операционная система* или UNIX, это будет означать ядро, если не оговорено иное.

Немного изменим наш предыдущий вопрос: так что же такое ядро? Есть ли это процесс или нечто отличающееся от всех других процессов? Ядро — это специальная программа, работающая непосредственно с аппаратурой.

---

[1] Однако в некоторых местах разработчики UNIX слегка перестарались. К примеру, интерпретация устройств работы с магнитной лентой как потоков символов весьма усложнила приложениям правильную обработку ошибок и исключений. Интерфейс работы с магнитной лентой не совсем удачно вписывается в интерфейс работы с устройствами в системе UNIX [Allm 87].

Ядро находится на диске в файле, обычно имеющем название /vmunix или /unix (в зависимости от производителя ОС). Когда система стартует, с диска загружается ядро при помощи специальной процедуры *начальной загрузки* (bootstrapping). Ядро инициализирует систему и устанавливает среду для выполнения процессов. Затем создаются несколько начальных процессов, которые в дальнейшем порождают остальные процессы. После загрузки ядро находится в памяти постоянно до тех пор, пока работа системы не будет завершена. Ядро управляет процессами и предоставляет им различные службы.

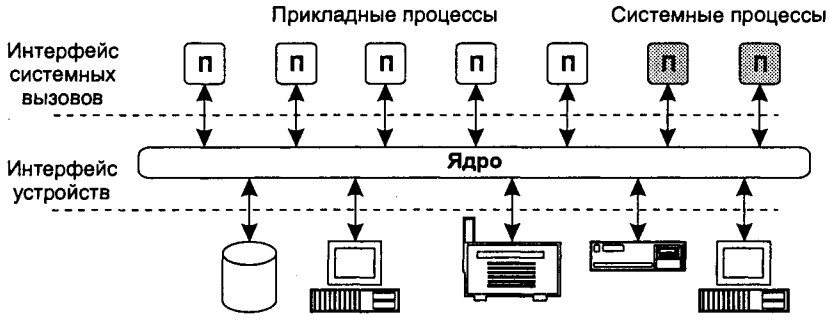


Рис. 2.1. Ядро взаимодействует с процессами и устройствами

Операционная система UNIX обеспечивает свою функциональность четырьмя различными способами:

- Прикладные процессы запрашивают от ядра необходимые службы при помощи *интерфейса системных вызовов* (см. рис. 2.1), являющегося центральным компонентом API системы UNIX. Ядро выполняет эти запросы от имени вызывающих процессов.
- Некоторые некорректные действия процесса, такие как попытки деления на ноль или переполнение стека процесса, являются причиной *аппаратных исключений*. Возникающие ошибки требуют вмешательства ядра, после чего происходит их обработка от имени процесса.
- Ядро обрабатывает аппаратные *прерывания* от периферийных устройств. Устройства используют механизм прерываний для оповещения ядра об окончании процесса ввода-вывода или изменении состояния. Ядро трактует прерывания как глобальные события, не относящиеся к какому-то одному определенному процессу.
- Набор специальных системных процессов, таких как *swapper* или *pagedaemon*, занимается выполнением обширных системных задач, таких как управление рядом активных процессов или поддержка пула свободной памяти.

В следующих разделах этой главы будут описаны вышеперечисленные механизмы и будет определено понятие контекста выполнения процесса.

2.2. Режим, пространство и контекст

Для возможности функционирования системы UNIX аппаратная часть компьютера должна поддерживать по крайней мере два режима выполнения: более привилегированный *режим ядра* и менее привилегированный *режим задачи*. Как и следовало ожидать, прикладные программы работают в режиме задачи, а функции ядра выполняются в режиме ядра. Ядро защищает часть адресного пространства от доступа в режиме задачи. Более того, наиболее привилегированные машинные инструкции могут выполняться только в режиме ядра.

Во многих аппаратных архитектурах поддерживается более двух режимов выполнения. Например, архитектура Intel 80x86[1] поддерживает четыре *уровня выполнения*[2], самым привилегированным из которых является нулевой. *UNIX использует только два из них.* Главной причиной появления различных режимов выполнения является безопасность. Если пользовательские процессы выполняются в менее привилегированном режиме, то они не могут случайно или специально повредить другой процесс или ядро системы. Последствия, вызванные ошибками в программе, носят локальный характер и обычно не влияют на выполнение других действий или процессов.

Большинство реализаций UNIX используют *виртуальную память*. В системе виртуальной памяти адреса, выделенные программе, не ссылаются непосредственно на физическое размещение в памяти. Каждому процессу предоставляется собственное *виртуальное адресное пространство*, а ссылки на виртуальные адреса памяти транслируются в их фактическое нахождение в физической памяти при помощи набора *карт трансляции адресов*. Многие системы реализуют такие карты как *таблицы страниц*, с одной записью для каждой *страницы* адресного пространства процесса (страница — это выделенный и защищенный блок памяти фиксированного размера). Аппаратно реализованный *блок управления памятью* (memory management unit, MMU) обычно обладает определенным набором регистров для определения карт трансляции адресов процесса, выполняющегося в данный момент времени (также называемого *текущим*). Когда текущий процесс уступает процессорное время другому процессу (*переключение контекста*), ядро размещает в этих регистрах указатели на карты трансляции адресов нового процесса. Регистры MMU являются привилегированными и могут быть доступны только в режиме ядра. Это дает гарантию того, что процесс будет ссылаться на адреса памяти только своего адресного пространства и не имеет доступа или возможности изменения адресного пространства другого процесса.

Определенная часть виртуального адресного пространства каждого процесса отображается на код и структуры данных ядра. Эта часть называется

[1] За исключением 8086 и 80186. — *Прим. ред.*

[2] Еще их называют уровнями привилегий или защищенности. — *Прим. ред.*

системным пространством или *пространством ядра* и может быть доступна только в режиме ядра. В системе может одновременно выполняться только одна копия ядра, следовательно, все процессы отображаются в единое адресное пространство ядра. В ядре содержатся глобальные структуры и информация, дающая возможность ему иметь доступ к адресному пространству любого процесса. Ядро может обращаться к адресному пространству текущего процесса напрямую, так как регистры MMU хранят всю необходимую для этого информацию. Иногда ядру требуется обратиться к адресному пространству, не являющемуся в данный момент текущим. В этом случае обращение происходит не непосредственно, а при помощи специального временного отображения.

В то время как ядро совместно используется всеми процессами, системное пространство защищается от доступа в режиме задачи. Процессы не могут напрямую обращаться к ядру и должны использовать для этого интерфейс системных вызовов. После того как процесс производит *системный вызов*, запускается специальная последовательность команд (называемая *переключателем режимов*), переводящая систему в режим ядра, а управление передается ядру, которое и обрабатывает операцию от имени процесса. После завершения обработки системного вызова ядро вызывает другую последовательность команд, возвращающую систему обратно в режим задачи (производится еще одно переключение режима), и снова передает управление текущему процессу. Интерфейс системных вызовов описан подробнее в разделе 2.4.1.

Существуют два важных для любого процесса объекта, которые управляются ядром. Они обычно реализуются как часть адресного пространства процесса. Это *область u* (u-area, или user-area) и *стек ядра* (kernel stack). Область u является структурой данных, содержащей нужную ядру информацию о процессе, такую как таблицу файлов, открытых процессом, данные для идентификации, а также сохраненные значения регистров процесса, пока процесс не является текущим. Процесс не может произвольно изменять эту информацию — и, следовательно, область u является защищенной от доступа в режиме задачи (некоторые реализации ОС позволяют процессу считывать эту информацию, но не изменять ее).

Ядро системы UNIX является реентерабельным, то есть к ядру могут обращаться одновременно несколько процессов. Фактически они могут выполнять одни и те же последовательности инструкций параллельно. (Разумеется, в один момент времени выполняется только один процесс, остальные при этом заблокированы или находятся в режиме ожидания[1].) Таким образом, каждому процессу необходим собственный стек ядра для хранения данных,

[1] Реентерабельностью называется возможность единовременного обращения различных процессов к одному и тому же машинному коду, загруженному в память компьютера в единственном экземпляре. — *Прим. ред.*

используемых последовательностью функций ядра, инициированной вызовом из процесса. Многие реализации UNIX располагают стек ядра в адресном пространстве каждого процесса, но при этом запрещают к нему доступ в режиме задачи. Концептуально область u и стек ядра хоть и создаются для каждого процесса отдельно и хранятся в его адресном пространстве, они, тем не менее, являются *собственностью* ядра.

Еще одним важным понятием является *контекст выполнения*. Функции ядра могут выполняться только в *контексте процесса* или в *системном контексте*. В контексте процесса ядро функционирует от имени текущего процесса (например, пока обрабатывает системный вызов) и может иметь доступ и изменять адресное пространство, область u и стек ядра этого процесса. Более того, ядро может заблокировать текущий процесс, если тому необходимо ожидать освобождения ресурса или реакции устройства.

Ядро также должно выполнять глобальные задачи, такие как обслуживание прерываний устройств и перасчет приоритетов процессов. Такие задачи не зависят от какого-либо конкретного процесса и, следовательно, обрабатываются в системном контексте (также называемом *контекстом прерываний*). Если ядро функционирует в системном контексте, то оно не должно иметь доступа к адресному пространству, области u и стеку ядра текущего процесса. Ядро также не должно в этом режиме блокировать процессы, так как это приведет к блокировке «невинного» процесса.

Рис. 2.2. Режим и контекст выполнения

Мы отметили основные различия между режимами задачи и ядра, пространством процесса и системы, контекстом процесса и системы. На рис. 2.2 проиллюстрированы все эти определения. Код приложения выполняется в режиме задачи и контексте процесса и может иметь доступ только к про-

странству процесса. Системные вызовы и исключения обрабатываются в режиме ядра, но в контексте процесса, однако эти задачи могут иметь доступ к пространству процесса и системы. Прерывания обрабатываются в режиме ядра и системном контексте и могут иметь доступ только к пространству системы.

2.3. Определение процесса

Так что же такое процесс в системе UNIX? Наиболее часто встречающийся ответ на этот вопрос таков: «Процесс — это экземпляр выполняемой программы». Однако такое определение является поверхностным, и для его более детального описания необходимо рассмотреть различные свойства процесса. Процесс — это нечто выполняющее программу и создающее среду для ее функционирования. В это понятие также входит адресное пространство и точка управления. Процесс — это основная единица расписания, так как только один процесс может в один момент времени занимать процессор. Кроме этого процесс старается перехватить ресурсы системы, такие как различные устройства или память. Он также запрашивает системные службы, которые выполняются для него и от его имени ядром.

Каждый процесс имеет определенное время жизни. Большинство процессов создаются при помощи системного вызова fork или vfork и работают до тех пор, пока не будут завершены вызовом exit. Во время функционирования процесс может запускать одну или несколько программ. Для запуска программ процесс использует системный вызов exec.

В системе UNIX процессы иерархически строго упорядочены. Каждый процесс имеет одного *родителя* (parent, или *родительский процесс*) и может иметь также одного или нескольких *потомков* (child, или *процесс-потомок*). Иерархия процессов может быть представлена как перевернутое дерево, в вершине (основании) которого находится процесс init. Процесс init (названный так в силу того, что он запускает программу /etc/init) является первым прикладным процессом, создаваемым во время загрузки системы. Этот процесс порождает все остальные прикладные процессы. Во время запуска системы создаются несколько различных процессов, например swapper или pagedaemon (называемого также pageout daemon), которые не являются потомками init. Если какой-либо процесс завершен и после него остаются функционирующие процессы-потомки, то они становятся *«осиротевшими»* (orphan) и наследуются процессом init.

2.3.1. Состояние процесса

В системе UNIX процессы всегда имели строго определенное *состояние*. Переход от одного состояния к другому осуществляется вследствие различных событий. На рис. 2.3 показаны важнейшие состояния процесса в системе UNIX, а также события, являющиеся причиной *перехода в иное состояние*.

Системный вызов fork создает процесс, который начинает свой жизненный цикл в *начальном* состоянии, также называемом *переходным* (idle). После того как создание процесса завершится, вызов fork переводит его в состояние *готовности к работе* (ready to run), в котором процесс ожидает своей очереди на обслуживание. В какой-то момент времени ядро выбирает этот процесс для выполнения и инициирует переключение контекста. Это производится вызовом процедуры ядра (обычно называемой swtch), которая загружает аппаратный контекст процесса (см. раздел 2.3.2) в системные регистры и передает ему управление. Начиная с этого момента новый процесс ведет себя так же, как и любой другой. Последующие изменения его состояния описаны ниже.

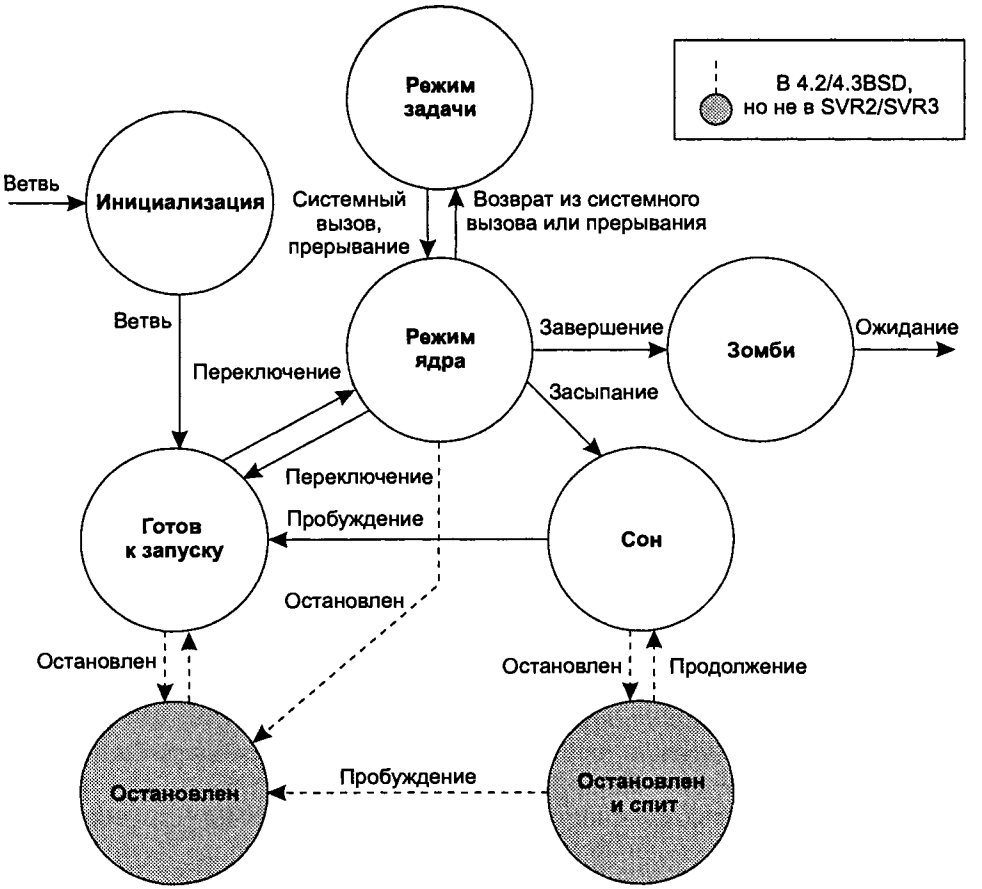


Рис. 2.3. Состояния процесса и переходы процесса в иные состояния

Процесс, функционирующий в режиме задачи, переходит в режим ядра в результате системного вызова или прерывания и возвращается обратно в ре-

жим задачи после завершения процедуры обработки[1]. Пока обрабатывается системный вызов или прерывание, процессу иногда необходимо ждать некоторого события или освобождения ресурса, недоступного в данный момент времени. Такое ожидание реализуется вызовом sleep(), который помещает процесс в очередь «спящих» процессов и переводит его в *спящее*, или *ожидающее* (asleep), состояние. После того как необходимое событие произойдет или ресурс станет доступен, ядро «разбудит» процесс, который с этого момента снова станет *готов к работе* и перейдет в очередь на выполнение.

Когда процесс планируется на выполнение, он изначально выполняется в режиме ядра (*состояние выполнения ядром*), где происходит переключение его контекста. Следующее изменение состояния зависит от того, чем занимался процесс до того, как достиг своей очереди на выполнение. Если процесс был только что создан или выполнял код программы (и был вытеснен из очереди на выполнение процессом, имеющим больший приоритет), то он сразу же переводится в режим задачи. Если процесс был блокирован в ожидании ресурса во время выполнения системного вызова, то он продолжит выполнение вызова в режиме ядра.

Завершение работы процесса происходит в результате системного вызова exit или по *сигналу* (сигналы — это средства оповещения, используемые ядром; см. главу 4). В любом случае ядро освобождает все ресурсы завершаемого процесса (сохраняя информацию о *статусе выхода* и *использовании ресурсов*) и оставляет процесс в состоянии зомби (zombie). Процесс остается в этом состоянии до тех пор, пока его процесс-родитель не вызовет процедуру wait (или один из ее вариантов), которая уничтожит процесс и передаст статус выхода родительскому процессу (см. раздел 2.8.6).

В системе 4BSD определены дополнительные состояния, не поддерживаемые в системах SVR2 или SVR3. Процесс может быть *остановлен* (stopped) или переведен в *состояние ожидания* при помощи сигнала *остановки* (SIGSTOP, SIGTSTP, SIGTTIN или SIGTTOU). В отличие от других сигналов, которые отрабатываются только во время выполнения процесса, сигнал остановки изменяет состояние процесса немедленно. Если процесс находится в состоянии работы или готовности к работе, то его состояние изменяется на остановленное. Если процесс находился в спящем режиме, когда поступил сигнал, его состояние изменится на спящее и остановленное. Остановленный процесс может продолжить работу при помощи сигнала *продолжения* (SIGCONT), который возвратит его в состояние готовности к работе. Если процесс был остановлен, находясь в спящем режиме, сигнал SIGCONT возвратит его обратно в этот режим. Позже такие возможности были добавлены и в SVR4 (см. раздел 4.5)[2].

---

[1] Прерывания могут произойти и тогда, когда система находится в режиме ядра. В этом случае система будет оставаться в этом режиме после завершения обработки прерывания.

[2] Система SVR3 поддерживает состояние остановки для процесса только с целью проведения процедуры *трассировки процесса* (см. раздел 6.2.4). Если трассируемый процесс получит любой сигнал, он перейдет в состояние остановки, а его родитель будет «разбужен» ядром.

2.3.2. Контекст процесса

Каждый процесс имеет четко определенный контекст, включающий всю информацию, необходимую для его описания. Ниже перечислены компоненты контекста:

- **Адресное пространство задачи.** Обычно делится на несколько составляющих: текст программы (выполняемый код), данные, стек задачи, совместно используемые области памяти и т. д.
- **Управляющая информация.** Ядро использует для поддержки управляющей информации о процессе две основные структуры данных: область u и структуру proc. Каждый процесс также обладает собственным стеком ядра и картами трансляции адресов.
- **Полномочия.** Включают в себя идентификаторы пользователя и группы, ассоциируемые с данным процессом (которые будут описаны в разделе 2.3.3).
- **Переменные окружения.** Это набор строк в форме:

  ```
  переменная=значение
  ```

  Переменные окружения наследуются от родительского процесса. Во многих вариантах UNIX такие строки хранятся в вершине стека. Стандартные пользовательские библиотеки имеют функции для добавления, удаления или изменения переменных, а также для возврата ее значения. При запуске новой программы вызывающий процесс может сообщить функции exec о том, что переменные окружения должны остаться «родительскими» или вместо этого предложить новый набор переменных.
- **Аппаратный контекст.** Включает содержимое регистров общего назначения, а также набора специальных системных регистров. Системные регистры содержат:
  - **Программный счетчик** (program counter, PC). Хранит адрес следующей выполняемой инструкции[1].
  - **Указатель стека** (stack pointer, SP). Содержит адрес верхнего элемента стека[2].
  - **Слово состояния процессора** (processor status word, PSW). Содержит несколько битов с информацией о состоянии системы, в том числе о текущем и предыдущем режимах выполнения, текущем и предыдущем уровнях приоритетов прерываний, а также биты переполнения и переноса.

---

[1] Обозначение характерно для машин PDP и процессоров ALPHA, в архитектуре Intel эту роль играет регистр (E)IP — указатель команд, instruction pointer. Обозначения SP и PSW в ней сохранились. В дальнейшем мы будем чаще пользоваться вторым термином. — *Прим. ред.*

[2] Или нижнего, для машин, в которых стек растет вниз. Также на некоторых системах указатель стека может содержать адрес, по которому будет занесен следующий помещаемый в стек элемент.

- **Регистры управления памятью**, в которых отображаются таблицы трансляции адресов процесса.
- **Регистры сопроцессора** (Floating point unit, FPU).

Машинные регистры содержат аппаратный контекст текущего выполняемого процесса. Когда происходит переключение контекста, эти регистры сохраняются в специальном разделе области u текущего процесса, который называется *блоком управления процессом* (process control block, PCB). Ядро выбирает следующий процесс для выполнения и загружает его аппаратный контекст из блока PCB.

2.3.3. Полномочия пользователя

Каждый пользователь системы имеет свой уникальный номер, называемый *идентификатором пользователя* (user ID, или UID). Системный администратор обычно также создает несколько различных групп пользователей, каждая из которых обладает уникальным *идентификатором группы* (user group ID, или GID). Эти идентификаторы определяют принадлежность файлов, права доступа, а также возможность посылки сигналов другим процессам. Все перечисленные атрибуты получили единое название *полномочий*.

Система различает *привилегированного пользователя*, называемого *суперпользователем* (superuser), который обычно входит в систему под именем root. Этот пользователь имеет UID, равный 0, и GID, равный 1. Он обладает многими полномочиями, недоступными обычным пользователям. Он может иметь доступ к чужим файлам независимо от установок их защиты, а также выполнять ряд привилегированных системных вызовов (например, mknod, используемый для создания специальных файлов устройств). Многие современные системы UNIX, такие как SVR4.1/ES, поддерживают *расширенные механизмы защиты* [8]. В этих системах поддержка суперпользователя заменена разделением привилегий при проведении различных операций.

Каждый процесс обладает двумя идентификаторами пользователя — *реальным* (real) и *действительным* (effective)[1]. После того как пользователь входит в систему, программа входа в систему выставляет обеим парам UID и GID значения, определенные в базе паролей (то есть в файле /etc/passwd или в некоем распределенном механизме, например службе *NIS* корпорации Sun Microsystems). Когда процесс создается при помощи fork, потомок наследует полномочия от своего прародителя.

Эффективный UID и эффективный GID влияют на создание файлов и доступ к ним. Во время создания файла ядро устанавливает ему атрибуты владельца файла как эффективные UID и GID процесса, из которого был сделан вызов на создание файла. Во время доступа процесса к файлу ядро использует

[1] Есть еще третий — сохраненный (saved). — *Прим. ред.*

эффективные идентификаторы процесса для определения, имеет ли он право обращаться к этому файлу (подробнее см. раздел 8.2). Реальный UID и реальный GID идентифицируют владельца процесса и влияют на право отправки сигналов. Процесс, не имеющий привилегий суперпользователя, может передавать сигналы другому процессу только в том случае, если реальный или эффективный UID отправителя совпадает с реальным UID получателя.

Существуют три различных системных вызова, которые могут переопределять полномочия. Если процесс вызывает exec для выполнения программы, установленной в *режиме suid* (см. раздел 8.2.2), то ядро изменяет эффективный UID процесса на UID, соответствующий владельцу файла программы. Точно так же, если программа установлена в *режиме sgid*, ядро изменяет GID вызываемого процесса.

Система UNIX предлагает описанную возможность с целью предоставления специальных прав пользователям для определенных целей. Классическим примером такого подхода является программа passwd, позволяющая пользователям изменять свои пароли. Этой программе необходимо записать результат в базу данных паролей, которая обычно недоступна пользователям для прямых изменений (с целью защиты от модификаций записей других пользователей). Таким образом, владельцем passwd является суперпользователь, но программа имеет установленный бит SUID. Это дает возможность обычному пользователю получить привилегии суперпользователя на время и в рамках выполнения программы passwd.

Пользователь также может настраивать свои полномочия при помощи системных вызовов setuid и setgid. Суперпользователю позволено при помощи этих вызовов изменять как реальные, так и эффективные идентификаторы UID и GID. Обычные пользователи могут обращаться к этим вызовам только для изменения своих эффективных идентификаторов UID или GID на реальные.

Существуют некоторые различия в интерпретации полномочий в ОС System V и BSD UNIX. В системе SVR3 поддерживаются *хранимые* (saved) UID и GID, которые имеют значения эффективных UID и GID перед вызовом exec. Системные вызовы setuid и setgid могут восстановить эффективные идентификаторы из хранимых значений. Хотя в системе 4.3BSD не поддерживается описанная возможность, пользователь ОС может войти в состав *дополнительных групп* (используя вызов setgroups). В то время как файлы, созданные пользователем, принадлежат его основной группе, он может иметь доступ к файлам, принадлежащим к его как основной, так и дополнительной группе (в зависимости от установок прав доступа к файлам для членов группы владельца).

В ОС SVR4 встроены все перечисленные возможности. Система поддерживает дополнительные группы, а также поддерживает хранимые идентификаторы UID и GID через вызов exec.

2.3.4. Область u и структура proc

Управляющая информация о процессе поддерживается с помощью двух структур данных, имеющихся у каждого процесса: области u и структуры proc. В различных реализациях UNIX ядро имеет массив фиксированного размера, состоящий из структур proc и называемый *таблицей процессов*. Размер этого массива зависит от максимального количества процессов, которые одновременно могут быть запущены в системе. Современные варианты UNIX, такие как SVR4, поддерживают динамическое размещение структур proc, но массив указателей на них также имеет фиксированный размер. Так как структура proc находится в системном пространстве, она всегда видна ядру в любой момент времени, даже когда процесс не выполняется.

Область u является частью пространства процесса. Это означает, что она отображаема и видима только в тот период времени, когда процесс выполняется. Во многих реализациях UNIX область u всегда отображается в один и тот же виртуальный адрес для каждого процесса, на который ядро ссылается через переменную u. Одной из задач переключателя контекста является сброс этого отображения, с тем чтобы ядро через переменную u «добралось» до физического расположения новой области u.

Иногда ядру системы необходимо получить доступ к области u процесса, не являющегося текущим. Такое возможно, но действие должно производиться не напрямую, а при помощи специального набора отображений. Различия в особенностях доступа обусловлены особенностями информации, хранящейся в структуре proc и области u. Область u содержит данные, необходимые только в период выполнения процесса. Структура proc включает в себя такую информацию, которая может потребоваться даже в том случае, если процесс не выполняется.

Основные поля области u перечислены ниже:

- блок управления процессом используется для хранения аппаратного контекста в то время, когда процесс не выполняется;
- указатель на структуру proc для этого процесса;
- реальные и эффективные UID и GID[1];
- входные аргументы и возвращаемые значения (или коды ошибок) от текущего системного вызова;
- обработчики сигналов и информация, связанная с ними (см. главу 4);
- информация из заголовка программы, в том числе размеры текста, данных и стека, а также иная информация по управлению памятью;

[1] В современных системах UNIX, таких как SVR4, пользовательские полномочия хранятся в структуре данных, располагаемой динамически, указатель на которую находится в структуре proc. Более подробное описание см. в разделе 8.10.7.

- таблица дескрипторов открытых файлов (см. раздел 8.2.3). Современные системы UNIX, такие как SVR4, расширяют эту таблицу динамически по мере необходимости;
- указатели на *vnode*. Объекты vnode представляют собой объекты файловой системы и будут подробнее описаны в разделе 8.7;
- статистика использования процессора, информация о профиле процесса, дисковых квотах и ресурсах;
- во многих реализациях UNIX стек ядра процесса является частью области u этого процесса.

Основные поля структуры proc охватывают:

- идентификацию: каждый процесс обладает уникальным *идентификатором процесса* (process ID, или *PID*) и относится к определенной *группе процессов*. В современных версиях системы каждому процессу также присваивается *идентификатор сеанса* (session ID);
- расположение карты адресов ядра для области u данного процесса;
- текущее состояние процесса;
- предыдущий и следующий указатели, связывающие процесс с очередью планировщика (или очередью приостановленных процессов, если данный процесс был заблокирован);
- канал «сна» для заблокированных процессов (см. раздел 7.2.3);
- приоритеты планирования задач и связанную информацию (см. главу 5);
- информацию об обработке сигналов: маски игнорируемых, блокируемых, передаваемых и обрабатываемых сигналов (см. главу 4);
- информацию по управлению памятью;
- указатели, связывающие эту структуру со списками активных, свободных или завершенных процессов (зомби);
- различные флаги;
- указатели на расположение структуры в *очереди хэша*, основанной на PID;
- информация об иерархии, описывающая взаимосвязь данного процесса с другими.

На рис. 2.4 продемонстрированы взаимосвязи процессов в системе 4.3BSD UNIX. На схеме представлены поля, описывающие иерархию процессов. Это — p_pid (идентификатор процесса), p_ppid (идентификатор родительского процесса), p_pptr (указатель на структуру proc родителя), p_cptr (указатель на старшего потомка), p_ysptr (указатель на следующий младший процесс того же уровня), p_osptr (указатель на следующий старший процесс того же уровня).

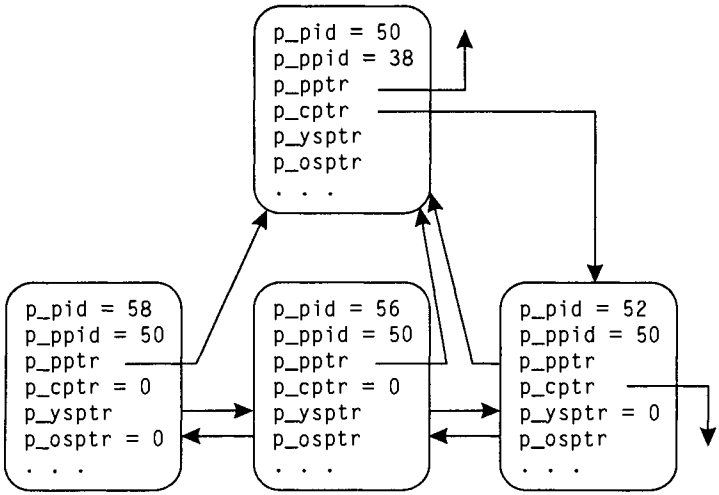


Рис. 2.4. Пример иерархии процессов в системе 4.3BSD UNIX

Во многих современных системах UNIX абстракция процесса была дополнена поддержкой нескольких нитей. Подробнее об этом будет сказано в главе 3.

2.4. Выполнение в режиме ядра

Существуют три различных типа событий, которые могут перевести систему в режим ядра. Это — прерывания устройств (interrupts), исключительные ситуации или просто исключения (exceptions), а также ловушки (traps) или программные прерывания (software interrupts). Каждый раз, когда ядру возвращается управление, оно обращается к таблице *диспетчеризации* (dispatch table), содержащей адреса низкоуровневых процедур обработки событий. Перед вызовом соответствующей процедуры ядро частично сохраняет состояние прерванного процесса (например, указатель команд и слово состояния процессора) в стеке ядра для этого процесса. После завершения работы процедуры ядро восстанавливает состояние процесса и изменяет режим его выполнения на прежнее значение. Прерывание может произойти и в тот момент, когда система уже находится в режиме ядра, в таком случае она останется в нем и после обработки прерывания.

Важно понимать разницу между прерываниями и исключительными состояниями. Прерывания — это асинхронные события, происходящие в периферийных устройствах, таких как диски, терминалы или аппаратный таймер. Так как прерывания не зависят от текущего процесса, они должны обрабатываться в системном контексте, при этом доступ в адресное пространство или область u процесса им не требуется. По этой же причине прерывания не должны производить блокировку, так как они могут заблокировать произвольный процесс. Исключительные состояния возникают по ходу работы

процесса по причинам, зависящим от него самого, например при попытке деления на ноль или обращения по несуществующему адресу. Обработчик исключительных состояний работает в контексте процесса и может обращаться к адресному пространству или области u процесса, а также блокировать процесс, если это необходимо. Программные либо системные прерывания (traps или ловушки) происходят во время выполнения процессом особых инструкций, например в процессе перехода в системные вызовы, и обрабатываются синхронно в контексте процесса.

2.4.1. Интерфейс системных вызовов

Программный интерфейс ОС определяется набором системных вызовов, предоставляемых ядром пользовательским процессам. Стандартная библиотека C, подключаемая по умолчанию ко всем программам пользователя, содержит *процедуру встраивания* для каждого системного вызова. Когда программа делает системный вызов, вызывается и соответствующая ему процедура встраивания. Она передает номер системного вызова (идентифицирующего каждый вызов ядра) в пользовательский стек и затем вызывает специальную инструкцию *системного прерывания*. Имя инструкции зависит от конкретной машины (например, это — syscall для MIPS R3000, chmk для VAX-11 или trap для Motorola 680x0). Функция этой инструкции заключается в изменении режима выполнения на режим ядра и передачи управления обработчику системного вызова, определенному по таблице диспетчеризации. Этот обработчик, обычно имеющий название syscall(), является точкой старта обработки любого системного вызова ядром.

Системные вызовы выполняются в режиме ядра, но в контексте процесса. Следовательно, они имеют возможность доступа к адресному пространству и области u вызывающего процесса. С другой стороны, они могут обращаться и к стеку ядра этого процесса. Обработчик syscall() копирует входные аргументы системного вызова из пользовательского стека в область u, а также сохраняет аппаратный контекст процесса в стеке ядра. Затем он использует номер системного вызова для доступа к его вектору (обычно называемому sysent[]), чтобы определить, какую именно системную функцию необходимо вызвать для обработки поступившего системного вызова. После завершения работы необходимой функции обработчик syscall() устанавливает возвращенные ею значения или код ошибки в соответствующие регистры, восстанавливает аппаратный контекст процесса, возвращает систему в режим задачи, передавая управление обратно библиотечной функции.

2.4.2. Обработка прерываний

Основная функция прерываний в компьютере заключается в том, чтобы позволить взаимодействовать периферийным устройствам с процессором, информируя его о завершении работы задачи, ошибочных состояниях и других событиях,

требующих немедленного внимания. Прерывания происходят независимо от текущих действий системы или какого-либо процесса. Это означает, что система не знает заранее, в каком месте выполнения потока инструкций может произойти прерывание. Функция, запускаемая для обслуживания прерывания, называется *обработчиком прерывания*, или *процедурой обслуживания прерывания*. Обработчик работает в режиме ядра и системном контексте. Так как прерванный процесс обычно не имеет никакого отношения к произошедшему в системе прерыванию, обработчик не должен обращаться к контексту процесса. По той же причине он также не обладает правом блокировки.

Однако прерывание оказывает некоторое влияние на выполнение текущего процесса. Время, потраченное на обработку прерывания, является частью кванта времени, отведенного процессу, даже если производимые действия не имеют ни малейшего к нему отношения. Так, обработчик прерываний системного таймера использует тики (промежутки времени между двумя прерываниями таймера) текущего процесса и потому нуждается в доступе к его структуре proc. Важно отметить, что контекст процесса защищен от доступа обработчиками прерываний не полностью. Неверно написанный обработчик может причинить вред любой части адресного пространства процесса.

Ядро также поддерживает программные или системные прерывания, которые могут генерироваться при выполнении специальных инструкций. Такие прерывания могут использоваться, к примеру, в переключателе контекстов или в планировании задач, функционирующих в режиме разделения времени и имеющих низкий приоритет. Несмотря на то что описанные прерывания происходят синхронно с нормальной работой системы, они обрабатываются точно так же, как и обычные.

Вследствие того что причиной возникновения прерываний может стать множество различных событий, представима и такая ситуация, когда прерывание происходит во время того, как обрабатывается другое. Перед разработчиками системы UNIX встала необходимость поддержки различных уровней приоритетов, для того чтобы прерывания более высокого уровня обслуживались раньше, чем прерывания более низких уровней. К примеру, прерывание аппаратного таймера должно обслуживаться раньше прерывания сети, так как последнее может потребовать больших объемов вычислений в течение нескольких тиков системного таймера.

В системах UNIX каждому типу прерывания принято назначать *уровень приоритета прерывания* (interrupt priority level, ipl). В первых реализациях системы уровень ipl находился в пределах от 0 до 7. В ОС BSD значение ipl возросло до 0–31. *Регистр состояния процессора* обычно содержит битовые поля, в которых хранится текущий (а иногда и предыдущий) ipl[1]. Номера приоритетов прерываний не одинаковы, так как зависят не только от конкретной реализации системы UNIX, но и от различия в архитектуре обору-

[1] Некоторые процессоры, например Intel 80x86, не поддерживают приоритеты прерываний на аппаратном уровне. В таких системах необходимо реализовывать уровни ipl программно. Эта проблема будет затронута в дальнейшем в упражнениях.

дования. В некоторых системах ipl 0 означает низший уровень приоритета, тогда как в иных это значение может оказаться наивысшим. Для облегчения создания процедур обработки прерываний и драйверов устройств в системах UNIX представлен набор триггеров для блокировки и разблокирования прерываний. Однако в различных реализациях системы для одних и тех же целей используются различные триггеры. В табл. 2.1 показаны некоторые триггеры, применяемые в 4.3BSD и SVR4.

Таблица 2.1. Установка уровней приоритетов прерываний в системах 4.3BSD и SVR4

4.3BSD	SVR4	Назначение
spl0	spl0 или splbase	Разрешить все прерывания
splsoftclock	spltimeout	Блокировать функции, планируемые таймерами
Splnet		Блокировать функционирование сетевых протоколов
	splstr	Блокировать прерывания STREAMS
spltty	spltty	Блокировать прерывания терминала
splbio	spldisk	Блокировать дисковые прерывания
splimp		Блокировать прерывания сетевых устройств
splclock		Блокировать прерывание аппаратного таймера
splhigh	spl7 или splhi	Запретить обработку всех прерываний
splx	splx	Восстановить ipl в предыдущее сохраненное значение

После того как в системе происходит прерывание, дальнейшие действия зависят от его уровня: если уровень ipl окажется выше текущего, то действия приостанавливаются, и запускается обработчик уже нового прерывания. Обработчик начинает свою работу на новом уровне ipl. После завершения процедуры обслуживания прерывания уровень ipl понижается до предыдущего значения (которое хранится в предыдущем слове состояния процессора в стеке прерываний), и ядро продолжает обработку текущего прерванного процесса. Если ядро получает прерывание, имеющее уровень более низкий или равный текущему значению ipl, то такое прерывание не будет обрабатываться незамедлительно. Оно будет сохранено в регистре прерываний и обработано после соответствующего изменения уровня ipl. Алгоритм обработки прерываний показан на рис. 2.5.

Уровни ipl сравниваются и устанавливаются на аппаратном уровне в зависимости от архитектуры конкретного компьютера. Ядро системы UNIX имеет механизмы четкого определения или установки уровней ipl. К примеру, ядро может повысить уровень ipl с целью блокировки прерываний на время выполнения некоторых критичных инструкций. Подробнее об этой возможности см. в разделе 2.5.2.

На некоторых аппаратных платформах поддерживается возможность организации отдельного глобального *стека прерываний*, используемого всеми обработчиками. На платформах, не имеющих подобного стека, обработчики

задействуют стек ядра текущего процесса. В таком случае должен быть обеспечен механизм изоляции остальной части стека ядра от обработчика. Для этого ядро помещает в свой стек *уровень контекста* перед вызовом обработчика. Этот уровень контекста, подобно кадру стека, содержит в себе информацию, необходимую для восстановления контекста выполнения, предшествующего вызову обработчика прерывания.

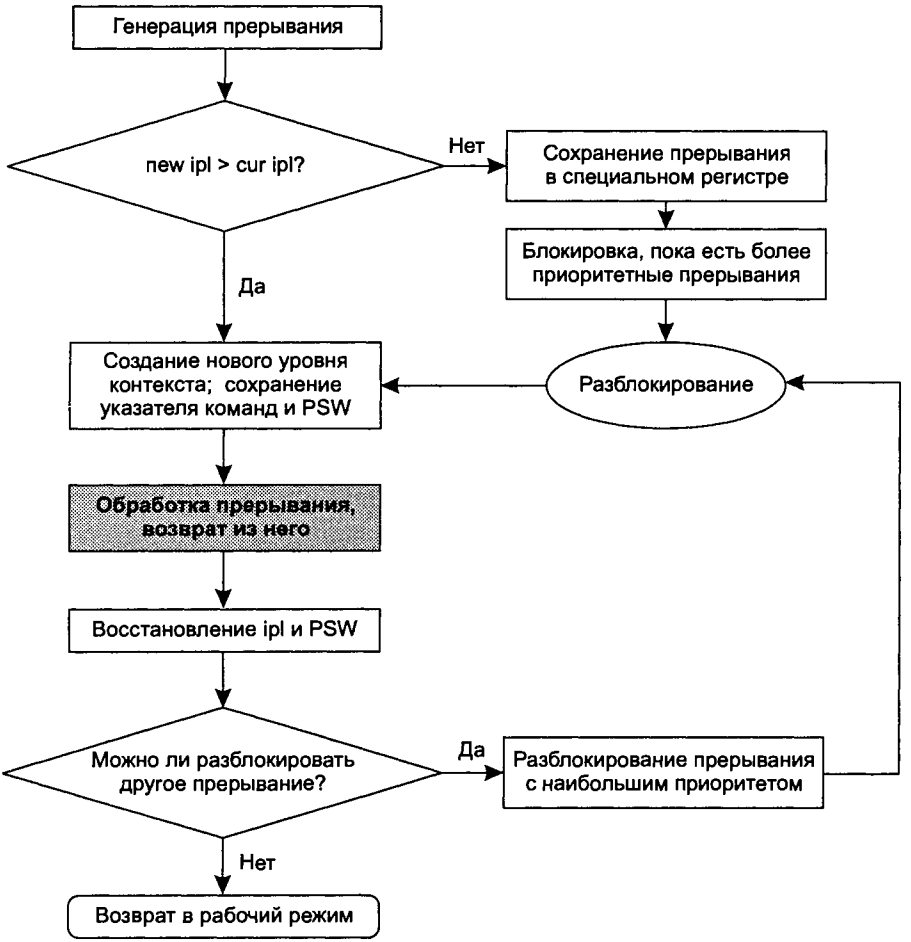


Рис. 2.5. Алгоритм обработки прерываний

2.5. Синхронизация

Ядро системы UNIX является реентерабельным. В любой момент времени в ядре могут быть активны сразу несколько процессов. Конечно, на однопроцессорных системах только один из них окажется текущим, в то время как

остальные будут блокированы, находясь в режиме ожидания освобождения процессора или иного системного ресурса. Так как все эти процессы используют одну и ту же копию структур данных ядра, необходимо обеспечивать некоторую форму синхронизации для предотвращения порчи ядра.

На рис. 2.6 показан пример, который показывает, к чему может привести отсутствие синхронизации. Представьте, что процесс пытается удалить элемент Б из связанного списка. После выполнения первой строки кода происходит прерывание, разрешающее другому процессу начать работу. Если второй процесс попытается получить доступ к тому же списку, то обнаружит его в противоречивом состоянии, как это показано на рис. 2.6, *б*. Становится очевидным, что необходимо использовать некий механизм, защищающий от возникновения подобных проблем.

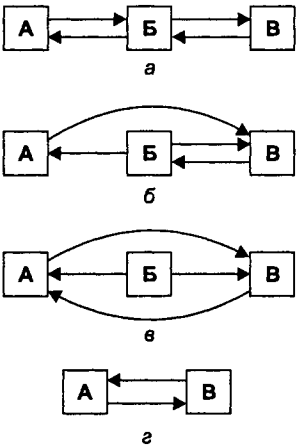


Рис. 2.6. Пример удаления элемента из связанного списка: *а* — начальное положение; *б* — после B->prev->next = B->next; *в* — после B->next->prev = B->prev; *г* — после free(B)

В системе UNIX применяется несколько различных технологий синхронизации. Система изначально создана невытесняющей. Это означает, что процесс, выполняющийся в режиме ядра, не может быть вытеснен другим процессом, *даже если отведенный ему квант времени уже исчерпан.* Процесс должен самостоятельно освободить процессор. Это обычно происходит в тот момент, когда процесс приостанавливает свою работу в ожидании необходимого ресурса или какого-то события; когда процесс завершил функционирование в режиме ядра и когда собирается возвращаться в режим задачи. В любом случае, так как процесс освобождает процессор добровольно, он может быть уверен, что ядро системы находится в корректном состоянии. (Современные системы UNIX, работающие в режиме реального времени, поддерживают возможность вытеснения при определенных условиях, см. подробнее раздел 5.6.)

Создание ядра системы не вытесняющим является гибким решением большинства проблем, связанных с синхронизацией. Вернемся к примеру, показанному на рис. 2.6. В данном случае ядро системы может обрабатывать связанный

список без его блокировки, не беспокоясь о возможном вытеснении. Существуют три ситуации, при возникновении которых необходима синхронизация:

- операции блокировки;
- прерывания;
- работа многопроцессорных систем.

2.5.1. Операции блокировки

Операция блокировки — это операция, которая блокирует процесс (то есть переводит процесс в *спящий* режим до тех пор, пока блокировка не будет снята). Так как ядро системы не является вытесняющим, оно может манипулировать большинством объектов (структурами данных и ресурсами) без возможности причинения им какого-либо вреда, так как заранее известно, что никакой другой процесс не может получить к ним в это время доступ. Однако существует некоторое количество объектов, которым необходима защита на время проведения блокировки. Для этого существуют специальные дополнительные механизмы. Например, процесс может производить операцию read (чтения) из файла в блочный буфер диска, находящийся в памяти ядра. Так как необходима процедура ввода-вывода с диска, процесс должен ожидать ее завершения, позволив в этот период времени выполняться другому процессу. Однако в этом случае ядру необходимо точно знать, что новый процесс ни в коем случае не получит доступ к буферу, так как тот находится в незавершенном состоянии.

Для защиты таких объектов ядро ассоциирует с ними атрибут защиты lock. Он может иметь простейшую реализацию и представлять собою однобитовый флаг, значение которого устанавливается, если объект заблокирован и сбрасывается в противоположном случае. Перед тем как начать пользоваться каким-либо объектом, каждый процесс обязан проверять, не заблокирован ли требуемый объект. Если тот свободен, процесс устанавливает флаг блокировки, после чего начинает использование ресурса. Ядро системы также ассоциирует с объектом еще один флаг — wanted (необходимость). Флаг устанавливается на объект процессом, если тот ему необходим, но в данный момент времени заблокирован. Когда другой процесс заканчивает использование объекта, то перед его освобождением проверяет флаг wanted и «будит» те процессы, которые ожидают данный объект. Такой механизм позволяет процессу блокировать ресурс на долгий период времени, даже если процесс приостанавливает свою работу, передавая возможность выполнения другими удерживая при этом необходимый ему ресурс.

На рис. 2.7 показан алгоритм блокировки ресурсов. Необходимо учитывать следующие замечания:

- Процесс блокирует себя в том случае, если не может получить необходимый ресурс, или в случае ожидания события, например завершения ввода-вывода. Для этого процесс вызывает процедуру sleep(). Вышеописанный процесс называется *блокировкой* по событию или ресурсу.

- Процедура sleep() помещает процесс в специальную очередь блокированных процессов, изменяет его режим на спящий и вызывает функцию swtch() для инициализации переключения контекста и дальнейшего разрешения выполнения следующего процесса.
- Процесс, освобождающий ресурс, вызывает процедуру wakeup() для пробуждения *всех* процессов, ожидающих этот ресурс[1]. Функция wakeup() находит все эти процессы, изменяет их режим на работоспособный и помещает их в очередь планировщика, в которой они ожидают выполнения.
- Иногда промежуток, прошедший между моментом, когда процесс был разбужен, и временем, когда подошла его очередь выполняться, бывает очень большим. Возможна такая ситуация, что другие текущие процессы могут снова занять необходимый ему ресурс.
- Следовательно, после пробуждения процессу необходимо снова проверить, доступен ли необходимый ресурс. Если тот оказывается занятым, то процесс снова переходит в спящий режим.

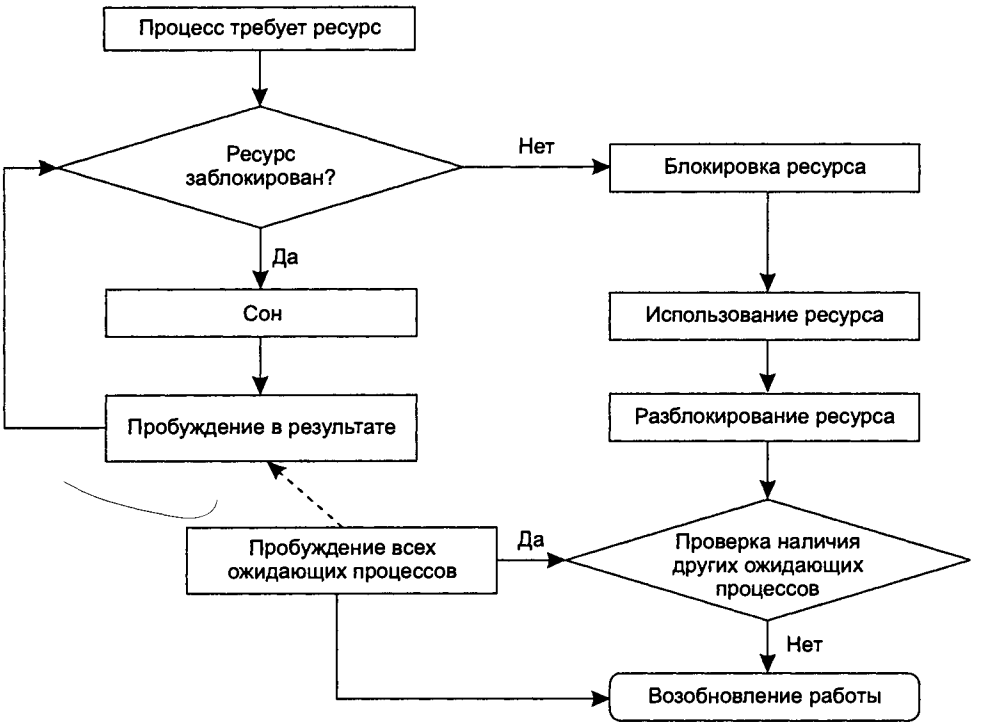


Рис. 2.7. Алгоритм блокировки ресурсов

[1] Более поздние версии системы UNIX поддерживают несколько альтернативных вариантов вызова wakeup(), таких как wake_one() и wakeprocs().

2.5.2. Прерывания

Несмотря на то что ядро системы защищено от вытеснения другим процессом, процесс, манипулирующий структурами ядра, может быть прерван различными устройствами. Если обработчик прерывания попытается получить доступ к таким структурам, то обнаружит, что они находятся в состоянии нарушения целостности. Возникшую проблему можно решить при помощи блокировки прерываний на период доступа к критически важным структурам данных. Ядро использует триггеры, приведенные в табл. 2.1 для повышения уровня ipl и блокировки всех прерываний. Такие области кодов называются *критическими секциями* (см. пример в листинге 2.1).

Листинг 2.1. Блокировка прерываний для критических областей

```
int x = splbio();    /* повышает уровень ipl, возвращает предыдущее
                        значение ipl */
...
                     / * изменение дискового кэша * /
...
splx(x);             /* восстанавливает предшествующий уровень ipl */
```

При маскировании прерываний следует учитывать следующие важные соображения:

- Прерывания обычно требуют незамедлительной обработки, следовательно, они не могут удерживаться слишком долго. Таким образом, критические области кода должны быть по возможности краткими и малочисленными.
- Необходимо блокировать только те прерывания, обработка которых требует обращения к данным, использующимся в критической области. В приведенном примере целесообразно блокировать только дисковые прерывания.
- Два различных прерывания могут иметь один и тот же уровень приоритета. Например, на многих системах прерывания терминала и диска происходят на уровне ipl 21.
- Блокирование прерывания приводит к блокированию всех прерываний, имеющих такой же или более низкий уровень приоритета.

ПРИМЕЧАНИЕ

При описании подсистем UNIX слово «блокирование» употребляется в нескольких различных значениях. Процесс блокируется на ресурсе или событии, переходя в спящий режим и ожидая освобождения необходимого ресурса или наступления события. Ядро блокирует прерывание или сигнал, задерживая на время его передачу. И наконец, подсистема ввода-вывода передает данные на устройства хранения или от них в блоках фиксированного размера.

2.5.3. Многопроцессорные системы

Появление многопроцессорных систем стало причиной возникновения нового класса проблем синхронизации. Защита ядра, основанная на отсутствии вытеснения, здесь уже не может быть применена. В однопроцессорных системах ядро может производить манипуляции с большинством структур данных, не опасаясь за их целостность, так как работа ядра не может быть вытеснена никем. Необходимо защищать только те данные, которые могут оказаться доступны обработчикам прерываний, а также те, целостность которых зависит от работы вызова sleep().

В многопроцессорных системах два процесса в состоянии одновременно выполняться в режиме ядра на разных процессорах[1], а также выполнять параллельно одну и ту же функцию. Таким образом, каждый раз, когда ядро обращается к глобальным структурам данных, оно должно защищать их от получения доступа с других процессоров. Сами механизмы защиты также должны быть защищены от особенностей выполнения в многопроцессорных системах. Если два процесса, выполняющиеся на различных процессорах, попытаются одновременно заблокировать один и тот же объект, только один должен завершить успешно эту процедуру.

Защита от прерываний является достаточно сложной задачей, так как все процессоры могут обрабатывать прерывания. Нецелесообразно производить блокировку на каждом процессоре, так как это чревато значительным снижением производительности системы. Многопроцессорные системы требуют более сложных механизмов синхронизации, которые будут подробнее описаны в главе 7.

2.6. Планирование процессов

Центральный процессор представляет собой ресурс, который используется всеми процессами системы. Часть ядра, распределяющая процессорное время между процессами, называется *планировщиком* (scheduler). В традиционных системах UNIX планировщик использует принцип *вытесняющего циклического планирования*. Процессы, имеющие одинаковые приоритеты, будут выполняться циклически друг за другом, и каждому из них будет отведен для этого определенный период (квант) времени, обычно равный 100 миллисекундам. Если какой-либо процесс, имеющий более высокий приоритет, становится выполняемым, то он вытеснит текущий процесс (конечно, если тот не выполняется в режиме ядра) даже в том случае, если текущий процесс не исчерпал отведенного ему кванта времени.

В традиционных системах UNIX приоритет процесса определяется двумя факторами: фактором «любезности» и фактором *утилизации*. Пользователи

[1] Под словом «многопроцессорных», автор, видимо, имел в виду двухпроцессорную систему. — *Прим. ред.*

могут повлиять на приоритет процесса при помощи изменения значения его «любезности», используя системный вызов nice (но только суперпользователь имеет полномочия увеличивать приоритет процесса). Фактор утилизации определяется степенью последней (то есть во время последнего обслуживания процесса процессором) загруженности CPU процессом. Этот фактор позволяет системе динамически изменять приоритет процесса. Ядро системы периодически повышает приоритет процесса, пока тот не выполняется, а после того, как процесс все-таки получит какое-то количество процессорного времени, его приоритет будет понижен. Такая схема защищает процессы от «зависания»[1], так как периодически наступает такой момент, когда ожидающий процесс получает достаточный уровень приоритета для выполнения.

Процесс, выполняющийся в режиме ядра, может освободить процессор в том случае, если произойдет его блокирование по событию или ресурсу. Когда процесс снова станет работоспособным, ему будет назначен приоритет ядра. Приоритеты ядра обычно выше приоритетов любых прикладных задач. В традиционных системах UNIX приоритеты представляют собой целые числа в диапазоне от 0 до 127, причем чем меньше их значение, тем выше приоритет процесса (так как система UNIX почти полностью написана на языке C, в ней используется стандартный подход к началу отсчета от нуля). Например, в ОС 4.3BSD приоритеты ядра варьируются в диапазоне от 0 до 49, а приоритеты прикладных задач — в диапазоне от 50 до 127. Приоритеты прикладных задач могут изменяться в зависимости от степени загрузки процессора, но приоритеты ядра являются фиксированными величинами и зависят от причины засыпания процесса. Именно по этой причине приоритеты ядра также известны как *приоритеты сна*. В табл. 2.2 приводятся примеры таких приоритетов для операционной системы 4.3BSD UNIX.

Более подробно работа планировщика будет изложена в главе 5.

Таблица 2.2. Приоритеты сна в ОС 4.3BSD UNIX

Приоритет	Значение	Описание
PSWP	0	Свопинг
PSWP + 1	1	Страничный демон
PSWP + 1/2/4	1/2/4	Другие действия по обработке памяти
PINOD	10	Ожидание освобождения inode
PRIBIO	20	Ожидание дискового ввода-вывода
PRIBIO + 1	21	Ожидание освобождения буфера
PZERO	25	Базовый приоритет
TTIPRI	28	Ожидание ввода с терминала
TTOPRI	29	Ожидание вывода с терминала

[1] Из-за отказа операционной системы его обслуживать. — *Прим. ред.*

Приоритет	Значение	Описание
PWAIT	30	Ожидание завершения процесса-потомка
PLOCK	35	Консультативное ожидание блокированного ресурса
PSLEP	40	Ожидание сигнала

2.7. Сигналы

Для информирования процесса о возникновении асинхронных событий или необходимости обработки исключительных состояний в системе UNIX используются *сигналы*. Например, когда пользователь нажимает на своем терминале комбинацию клавиш Ctrl+C, процессу, с которым пользователь в данный момент интерактивно работает, передается сигнал SIGINT. Когда процесс завершается, он отправляет своему процессу-родителю сигнал SIGCHLD. В ОС UNIX поддерживается определенное количество сигналов (31 в 4.3BSD и SVR3). Большинство из них зарезервированы для специальных целей, однако сигналы SIGUSR1 и SIGUSR2 доступны для использования в приложениях в произвольном назначении.

Сигналы используются для многих операций. Процесс может выслать сигнал одному или нескольким другим процессам, используя системный вызов kill. Драйвер терминала вырабатывает сигналы в ответ на нажатия клавиш или происходящие события для процессов, присоединенных к нему. Ядро системы вырабатывает сигналы для уведомления процесса о возникновении аппаратного исключения или в случаях возникновения определенных ситуаций, например превышения квот.

Каждый сигнал имеет определенную по умолчанию реакцию на него, обычно это завершение процесса. Некоторые сигналы по умолчанию игнорируются, а небольшая часть из них приостанавливает процесс. Процесс может указать системе на необходимость иной реакции на сигналы, отличной от заданной по умолчанию. Для этого используется вызов signal (System V), sigvec (BSD) или sigaction (POSIX.1). Действия, отличные от принятых по умолчанию, могут заключаться в запуске обработчика сигнала, определенного разработчиком приложения, его игнорировании, а иногда и в процедурах, противоположных тем, что производятся в обычных случаях. Процесс также имеет возможность временной блокировки сигнала. В таком случае сигнал будет доставлен процессу только после того, как тот будет разблокирован.

Процесс не в состоянии прореагировать на сигнал немедленно. После вырабатывания сигнала ядро системы уведомляет об этом процесс при помощи установки бита в маске ожидающих сигналов, расположенной в структуре proc данного процесса. Процесс должен постоянно быть готовым к получению сигнала и ответу на него. Это возможно только в том случае, если он находится в очереди на выполнение. В начале выполнения процесс обрабатывает

все ожидающие его сигналы и только затем продолжает работать в режиме задачи. (Эта процедура не включает функционирование самих обработчиков сигналов, которые также выполняются в режиме задачи.)

Что же происходит в том случае, если сигнал предназначен для спящего процесса? Будет ли он ожидать того момента, когда процесс снова станет работоспособным, или его «сон» будет прерван? Это зависит от причины приостановки работы процесса. Если процесс ожидает события, которое вскоре должно произойти (например, завершение операции ввода-вывода с диска), то нет необходимости «будить» такой процесс. С другой стороны, если процесс ожидает такое событие, как ввод с терминала, то заранее не известно, через какой промежуток времени оно может произойти. В таких случаях ядро системы будит процесс и прерывает выполнение системного вызова, из-за которого данный процесс был заблокирован. В операционной системе 4.3BSD поддерживается системный вызов siginterrupt, который служит для управления реакцией системного вызова[1]. Используя siginterrupt, разработчик может указать, будет ли обработка системных вызовов, прерываемых сигналами, прекращена или возобновлена. Более подробно сигналы будут описаны в главе 4.

2.8. Новые процессы и программы

Система UNIX является многозадачной средой, и различные процессы действуют в ней все время. Каждый процесс в один момент времени выполняет только одну программу, однако несколько процессов могут выполнять одну и ту же программу в параллельном режиме. Такие процессы умеют разделять между собой единую копию текста (кода) программы, хранящейся в памяти, но при этом содержать собственные области данных и стека. Более того, процесс способен загружать одну или больше программ в течение своего времени жизни. UNIX таким образом разделяет процессы и программы, которые могут выполняться при их помощи.

Для поддержки многозадачной среды в UNIX существует несколько различных системных вызовов, создающих и завершающих процессы, а также запускающих новые программы. Системные вызовы fork и vfork служат для создания новых процессов. Вызов exec загружает новую программу. Следует помнить о том, что работа процесса может быть завершена после принятия определенного сигнала.

2.8.1. Вызовы fork и exec

Системный вызов fork создает новый процесс. При этом вызывающий процесс становится родительским, а новый процесс является его потомком. Связь «родитель-потомок» создает иерархию процессов, графически изображенную

[1] Из-за которого процесс заблокирован. — *Прим. ред.*

на рис. 2.4. Процесс-потомок является точной копией своего родительского процесса. Его адресное пространство полностью повторяет пространство родительского процесса, а программа, выполняемая им изначально, также не отличается от той, что выполняет процесс-родитель. Фактически процесс-потомок начинает работу в режиме задачи после возврата из вызова fork.

Так как оба процесса (родитель и его потомок) выполняют одну и ту же программу, необходимо найти способ отличия их друг от друга и предоставления им возможности нормального функционирования. С другой стороны, различные процессы[1] невозможно заставить выполнять различные действия. Поэтому системный вызов fork возвращает различные значения процессу-родителю и его потомку: для потомка возвращается значение 0, а для родителя — идентификатор PID потомка.

Чаще всего после вызова fork процесс-потомок сразу же вызывает exec и тем самым начинает выполнение новой программы. В библиотеке языка C существует несколько различных форм вызова exec, таких как exece, execve и execvp. Все они незначительно отличаются друг от друга набором аргументов и после некоторых предварительных действий используют один и тот же системный вызов. Общее имя exec относится к любой неопределенной функции этой группы. В листинге 2.2 приведен фрагмент кода программы, использующий вызовы fork и exec.

Листинг 2.2. Использование вызовов fork и exec

```
If ((result=fork()))==0 {
  /* код процесса-потомка */
  ...
   if (execve("new_program", ...)<0)
       perror("execve failed");
       exit(1);
} else if (result<0) {
   perror("fork"); /* вызов fork неудачен */
}
/* здесь продолжается выполнение процесса-предка*/
```

После наложения новой программы на существующий процесс при помощи exec потомок не возвратит управление предыдущей программы, если не произойдет сбоя вызова. После успешного завершения работы функции exec адресное пространство процесса-потомка будет заменено пространством новой программы, а сам процесс будет возвращен в режим задачи с установкой его указателя команд на первую выполняемую инструкцию этой программы.

Так как вызовы fork и exec часто используются вместе, возникает закономерный вопрос: а может стоит использовать для выполнения задачи единый системный вызов, который создаст новый процесс и выполнит в нем новую программу. Первые системы UNIX [9] теряли много времени на дублирова-

[1] Имеется в виду оригинал и копия. — *Прим. ред.*

ние адресного пространства процесса-родителя для его потомка (в течение выполнения fork) для того, чтобы потом все равно заменить его новой программой.

Однако существует и немало преимуществ в разделении этих системных вызовов. Во многих клиент-серверных приложениях сервер может создавать при помощи fork множество процессов, выполняющих одну и ту же программу[1]. С другой стороны, иногда процессу необходимо запустить новую программу без создания для ее функционирования нового процесса. И наконец, между системными вызовами fork и exec процесс-потомок может выполнить некоторое количество заданий, обеспечивающих функционирование новой программы в желаемом состоянии. Это могут быть задания, выполняющие:

- операции перенаправления ввода, вывода или вывода сообщений об ошибках;
- закрытие не нужных для новой программы файлов, наследованных от предка;
- изменение идентификатора UID или GID (группы процесса);
- сброс обработчиков сигналов.

Если все эти функции попробовать выполнить при помощи единственного системного вызова, то такая процедура окажется громоздкой и неэффективной. Существующая связка fork-exec предлагает высокий уровень гибкости, а также является простой и модульной. В разделе 2.8.3 мы расскажем о способах минимизации проблем, связанных с быстродействием этой связки (из-за использования раздельных вызовов).

2.8.2. Создание процесса

Системный вызов fork создает новый процесс, который является почти точной копией его родителя. Единственное различие двух процессов заключается только в необходимости отличать их друг от друга. После возврата из fork процесс-родитель и его потомок выполняют одну и ту же программу (функционирование которой продолжается сразу же вслед за fork) и имеют идентичные области данных и стека. Системный вызов fork во время своей работы должен совершить следующие действия:

- зарезервировать пространство свопинга для данных и стека процесса-потомка;
- назначить новый идентификатор PID и структуру proc потомка;
- инициализировать структуру proc потомка. Некоторые поля этой структуры копируются от процесса-родителя (такие как идентификаторы

[1] С появлением современных многопотоковых систем UNIX такая возможность больше не востребована: теперь сервер может иметь возможность создания необходимого количества нитей выполнения.

пользователя и группы, маски сигналов и группа процесса), часть полей устанавливается в 0 (время нахождения в резидентном состоянии, использование процессора, канал сна и т. д.), а остальным присваиваются специфические для потомка значения (поля идентификаторов PID потомка и его родителя, указатель на структуру proc родителя);

- разместить карты трансляции адресов для процесса-потомка;
- выделить область u потомка и скопировать в нее содержимое области u процесса-родителя;
- изменить ссылки области u на новые карты адресации и пространство свопинга;
- добавить потомка в набор процессов, разделяющих между собой область кода программы, выполняемой процессом-родителем;
- постранично дублировать области данных и стека родителя и модифицировать карты адресации потомка в соответствии этими новыми страницами;
- получить ссылки на разделяемые ресурсы, наследуемые потомком, такие как открытые файлы и текущий рабочий каталог;
- инициализировать аппаратный контекст потомка посредством копирования текущего состояния регистров его родителя;
- сделать процесс-потомок выполняемым и поместить его в очередь планировщика;
- установить для процесса-потомка по возврату из вызова fork нулевое значение;
- вернуть идентификатор PID потомка его родителю.

2.8.3. Оптимизация вызова fork

Системный вызов fork должен предоставить процессу-потомку логически идентичную копию адресного пространства его родителя. В большинстве случаев потомок заменяет предоставленное адресное пространство, так как сразу же после выполнения fork вызывает exec или exit. Таким образом, создание копии адресного пространства (так, как это реализовано в первых системах UNIX) является неоптимальной процедурой.

Вышеописанная проблема была решена двумя различными способами. Сначала был разработан метод *копирования при записи*, впервые нашедший реализацию в ОС System V и в настоящий момент используемый в большинстве систем UNIX. При таком подходе страницы данных и стека родителя временно получают атрибут «только для чтения» и маркируются как «копируемые при записи». Потомок получает собственную копию карт трансляции адресов, но использует одни и те же страницы памяти вместе со своим родительским процессом. Если кто-то из них (родитель или потомок) попытается изменить страницу памяти, произойдет ошибочная исключительная

ситуация (так как страницы доступны только для чтения), после чего ядро системы запустит обработчик произошедшей ошибки. Обработчик увидит, что страница помечена как «копируемая при записи», и создаст новую ее новую копию, которую уже можно изменять. Таким образом, происходит копирование только тех страниц памяти, которые требуется изменять, а не всего адресного пространства целиком. Если потомок вызовет exec или exit, то защита страниц памяти вновь станет обычной, и флаг «копирования при записи» будет сброшен.

В системе BSD UNIX представлен несколько иной подход к решению проблемы, реализованный в новом системном вызове vfork. Программист может воспользоваться им вместо fork, если планирует вслед за ним вызвать exec. Функция vfork не производит копирования. Вместо этого процесс-родитель предоставляет свое адресное пространство потомку и блокируется до тех пор, пока тот не вернет его. Затем происходит выполнение потомка в адресном пространстве родительского процесса до того времени, пока не будет произведен вызов exec или exit, после чего ядро вернет родителю его адресное пространство и выведет его из состояния сна. Системный вызов vfork выполняется очень быстро, так как не копирует даже карты адресации. Адресное пространство передается потомку простым копированием регистров карты адресации. Однако следует отметить, что *вызов* vfork *является достаточно опасным, так как позволяет одному процессу использовать и даже изменять адресное пространство другого процесса.* Это свойство vfork используют различные программы, такие как csh.

2.8.4. Запуск новой программы

Системный вызов exec заменяет адресное пространство вызывающего процесса на адресное пространство новой программы. Если процесс был создан при помощи vfork, то вызов exec возвращает старое адресное пространство родительскому процессу. В ином случае этот вызов высвобождает старое адресное пространство. Вызов exec предоставляет процессу новое адресное пространство и загружает в него содержимое новой программы. По окончании работы exec процесс продолжает выполнение с первой инструкции новой программы.

Адресное пространство процесса состоит из нескольких определенных компонентов[1]:

- **Текст** (text). Содержит выполняемый код.
- **Инициализированные данные** (initialized data). Содержат объекты данных, которые в программе уже имеют начальные значения и соответствуют секции инициализированных данных в выполняемом файле.

[1] Такое разделение представляется весьма функциональным в теории, однако ядро не распознает так много различных компонентов. Например, в системе SVR4 адресное пространство видится просто как набор разделяемых или приватных отображений.

- **Неинициализированные данные** (uninitialized data). Имеет исторически сложившееся название *блока статического хранения* (block static storage, bss)[1]. Содержит переменные, которые были в программе описаны, но значения им присваивались. Объекты в этой области всегда заполнены нулями при первом обращении к ним. Так как хранение нескольких страниц нулей в выполняемом файле представляется нерациональным, в заголовке программы принято просто описывать общий размер этой области и предоставлять операционной системе самой сгенерировать заполненные нулями страницы.
- **Разделяемая память** (shared memory). Многие системы UNIX позволяют процессам совместно использовать одни и те же области памяти.
- **Разделяемые библиотеки** (shared libraries). Если система поддерживает библиотеки динамической связи, процесс может обладать несколькими отдельными областями памяти, содержащими библиотечный код, а также библиотечные данные, которые могут использоваться и другими процессами.
- **Куча** (heap). Источник динамически выделяемой памяти. Процесс берет память из кучи при помощи системных вызовов brk или sbrk, либо используя функцию malloc() из стандартной библиотеки C. Ядро предоставляет кучу каждому процессу и расширяет ее по мере необходимости.
- **Стек приложения** (user stack). Ядро выделяет стек каждому процессу. В большинстве традиционных реализаций системы UNIX ядро прозрачно отслеживает возникновение исключительных состояний, связанных с переполнением стека, и расширяет стек до определенного в системе максимума.

Применение разделяемой памяти является стандартной возможностью System V UNIX, но не поддерживается в системе 4BSD (до версии 4.3). Многие коммерческие реализации UNIX, основанные на BSD, поддерживают как разделяемую память, так и некоторые формы разделяемых библиотек в качестве дополнительных возможностей системы. В последующем описании работы exec мы рассмотрим программу, которая не использует ни одну из этих возможностей.

Система UNIX поддерживает различные форматы выполняемых файлов. Первым поддерживаемым форматом был a.out, имеющий 32-байтовый заголовок с последующими секциями текста, данных и таблицы символов. Заголовок программы содержит размеры *текста*, областей *инициализированных* и *неинициализированных данных*, а также *точку входа*, которая является адресом первой инструкции программы, которая будет выполнена. Заголо-

[1] Морис Бах в широко известной книге «Архитектура операционной системы UNIX» пишет, что сокращение bss имеет происхождение от ассемблерного псевдооператора для машины IBM 7090 и расшифровывается как block started by symbol (блок, начинающийся с символа). — *Прим. ред.*

вок также содержит *магическое число*, идентифицирующее файл как действительно выполняемый и дающее дополнительную информацию о его формате, такую как: требуется ли ему разбиение на страницы или начинается ли секция данных на краю страницы. Набор поддерживаемых магических чисел определен в каждой реализации UNIX по-своему.

Системный вызов exec выполняет следующие действия:

1. Разбирает путь к исполняемому файлу и осуществляет доступ к нему.
2. Проверяет, имеет ли вызывающий процесс полномочия на выполнение файла.
3. Читает заголовок и проверяет, что он действительно исполняемый[1].
4. Если для файла установлены биты SUID или SGID, то эффективные идентификаторы UID и GID вызывающего процесса изменяет на UID и GID, соответствующие владельцу файла.
5. Копирует аргументы, передаваемые в exec, а также *переменные среды* в пространство ядра, после чего текущее пользовательское пространство готово к уничтожению.
6. Выделяет пространство свопинга для областей данных и стека.
7. Высвобождает старое адресное пространство и связанное с ним пространство свопинга. Если же процесс был создан при помощи vfork, производится возврат старого адресного пространства родительскому процессу.
8. Выделяет карты трансляции адресов для нового текста, данных и стека.
9. Устанавливает новое адресное пространство. Если область текста активна (какой-то другой процесс уже выполняет ту же программу), то она будет совместно использоваться с этим процессом. В других случаях пространство должно инициализироваться из выполняемого файла. Процессы в системе UNIX обычно разбиты на страницы, что означает, что каждая страница считывается в память только по мере необходимости.
10. Копирует аргументы и переменные среды обратно в новый стек приложения.
11. Сбрасывает все обработчики сигналов в действия, определенные по умолчанию, так как функции обработчиков сигналов не существуют в новой программе. Сигналы, которые были проигнорированы или заблокированы перед вызовом exec, остаются в тех же состояниях.
12. Инициализирует аппаратный контекст. При этом большинство регистров сбрасывается в 0, а указатель команд получает значение точки входа программы.

[1] Вызов exec также может выполнять сценарии командного интерпретатора, имеющие первую строку типа #!shell_name.

2.8.5. Завершение процесса

Функция ядра exit() предназначена для завершения процесса. Она вызывается изнутри, когда процесс завершается по сигналу. С другой стороны, программа может выполнить системный вызов exit, который, в свою очередь, вызовет функцию exit(). Функция exit() производит следующие действия:

1. Отключает все сигналы.
2. Закрывает все открытые файлы.
3. Освобождает файл программы и другие ресурсы, например текущий каталог.
4. Делает запись в журнал данной учетной записи.
5. Сохраняет данные об использованных ресурсах и статус выхода в структуре proc.
6. Изменяет состояние процесса на SZOMB (зомби) и помещает его структуру proc в список процессов-зомби.
7. Устанавливает процесс init любому существующему потомку завершаемого процесса в качестве родителя.
8. Освобождает адресное пространство, область u, карты трансляции адресов и пространство свопинга.
9. Посылает родителю завершаемого процесса сигнал SIGCHLD. Этот сигнал обычно игнорируется и реально необходим только в тех случаях, если по какой-то причине родительскому процессу необходимо знать о завершении работы его потомка.
10. Будит родительский процесс, если тот был приостановлен.
11. Вызывает swtch() для перехода к выполнению следующего процесса в расписании.
12. После завершения работы exit() процесс находится в состоянии зомби. Вызов exit не освобождает структуру proc завершенного процесса, так как его родителю, возможно, будет необходимо получить статус выхода и информацию об использовании ресурсов. За освобождение структуры proc потомка отвечает его процесс-родитель, как будет описано подробнее позже. По завершении этой процедуры структура proc возвращается в список свободных структур, и на этом процесс «чистки следов» завершается.

2.8.6. Ожидание завершения процесса

Часто родительскому процессу необходимо обладать информацией о завершении работы своего потомка. Например, когда командный интерпретатор, исполняя команду, порождает интерактивный процесс (переводящий

ввод и вывод на себя) и, становясь его родителем, должен ждать завершения своего потомка, чтобы после этого вновь быть готовым ко вводу очередной команды. Когда завершается фоновый процесс, командному интерпретатору может потребоваться сообщить об этом пользователю (выводом соответствующего сообщения на терминал). Командный интерпретатор также может запрашивать статус выхода процесса-потомка, так как дальнейшие действия пользователя зависят от того, завершился ли процесс успешно или имела место ошибка. В системах UNIX поддерживаются следующие системные вызовы, дающие возможность отслеживания завершения работы процессов:

```
wait(stat_loc);                          /* System V, BSD и POSIX.1 */
wait3(statusp, options, rusagep);        /* BSD */
waitpid(pid, stat_loc, options);         /* POSIX.1 */
waitid(idtype, id, infop, options);      /* SVR4 */[1]
```

Системный вызов wait позволяет процессу ожидать завершения работы его потомка. Так как потомок может уже оказаться завершенным к моменту совершения вызова, то ему необходимо уметь распознавать такую ситуацию. После запуска вызов wait первоначально проверяет, имеет ли вызывающий его процесс потомков, функционирование которых завершено или приостановлено. Если он находит такие процессы, то немедленно происходит возврат из этого системного вызова. Если таких процессов нет, то wait блокирует вызвавший его процесс до тех пор, пока один из потомков не завершит свою работу, после чего последует возврат из вызова. В обоих случаях вызов wait возвратит PID завершившегося процесса, запишет его статус выхода в stat_loc и освободит его структуру proc (если более одного процесса-потомка завершили свою работу, wait обработает только первый из найденных). Если процесс-потомок находится в режиме трассировки, возврат из wait также произойдет, когда процесс-потомок получит сигнал. Ошибку wait вернет в случае, когда родительский процесс не имеет ни одного потомка (функционирующего или уже завершенного) либо если его работа была прервана поступившим сигналом.

Система 4.3BSD поддерживает вызов wait3 (названный так потому, что он имеет три аргумента), который также возвращает информацию об использовании ресурсов потомком (время работы в режиме ядра и в режиме задачи процесса-потомка, а также о всех его завершенных потомках). В стандарте POSIX.1 [6] описан системный вызов waitpid, в котором используется аргумент pid для ожидания потомка, имеющего определенный идентификатор процесса или группы процесса. Системные вызовы wait3 и waitpid поддерживают две опции: WNOHANG и WUNTRACED. Опция WNOHANG заставляет вызов

[1] В операционной системе SVR4 поддержка команд wait3 и waitpid реализована в виде библиотечных функций.

wait3 немедленно завершить работу, если он не нашел ни одного завершившегося процесса. Опция WUNTRACED завершает работу вызова, если потомок приостанавливается, или вновь продолжает функционирование. В ОС SVR4 системный вызов waitid поддерживает все вышеописанные возможности. Он позволяет вызывающему его процессу задавать PID или GID процесса, завершения которого должен ждать, а также определять события, по которым также произойдет возврат из вызова. Возвращает более подробную информацию о процессе-потомке.

2.8.7. Процессы-зомби

Когда процесс завершается, он остается в состоянии «зомби» (zombie) до тех пор, пока окончательно не будет «вытерт» родительским процессом. В этом режиме единственным занимаемым ресурсом остается структура proc, в которой хранится статус выхода, а также информация об использовании ресурсов системы[1]. Эта информация может быть важна для родительского процесса, который получает ее посредством вызова wait, который также освобождает структуру proc потомка. Если родительский процесс завершается раньше, чем его потомок, то тот усыновляется процессом init. После завершения работы потомка процесс init вызовет wait для освобождения его структуры proc.

Определенная проблема возникает в том случае, если процесс завершится раньше своего родителя и последний не вызовет wait. Тогда структура процесса-потомка proc не будет освобождена, а потомок останется в состоянии зомби до тех пор, пока система не будет перезагружена. Такая ситуация возникает очень редко, так как разработчики командных интерпретаторов знают о существовании проблемы и стараются не допустить ее возникновения в своих программах. Однако потенциальная возможность такой ситуации остается, когда недостаточно внимательно написанные программы не следят за всеми своими процессами-потомками. Это может быть достаточно раздражительным, так как процессы-зомби видимы при помощи ps, а пользователи никак не могут их завершить, *так как они уже завершены*. Более того, они продолжают занимать структуру proc, уменьшая тем самым максимальное количество активных процессов.

В некоторых более поздних реализациях UNIX поддерживается возможность указания на то, что процесс не будет ожидать завершения работы своих потомков. Например, в системе SVR4 процесс может выставить флаг SA_NOCLDWAIT в системном вызове sigaction, определяя действие на сигнал SIGCHLD. Это дает возможность ядру системы не создавать процессы-зомби, если потомок вызывающего процесса завершит функционирование.

---

[1] В некоторых реализациях UNIX для хранения таких данных используются специальные структуры zombie.

2.9. Заключение

В этой главе мы обсудили взаимодействие ядра системы и прикладных процессов в традиционных системах UNIX. Эти вопросы требуют более широкого обзора, поэтому нам необходимо рассмотреть специфические части системы подробнее. Современные варианты UNIX, такие как SVR4 или Solais 2.x, предлагают дополнительные возможности, описание которых можно найти в следующих главах книги.

2.10. Упражнения

1. Какие элементы контекста процесса необходимо сохранять ядру при обработке:
   - переключения контекста;
   - прерывания;
   - системного вызова?
2. В чем преимущества и недостатки динамического размещения таких объектов, как структура proc, и блоков таблицы дескрипторов?
3. Каким образом ядро системы узнает о том, какой из системных вызовов был сделан? Каким образом происходит доступ к аргументам вызова (хранящимся в пользовательском стеке)?
4. Найдите сходства и различия в обработке системных вызовов и исключений. В чем производимые действия сходны и чем они отличаются друг от друга?
5. Многие реализации UNIX совместимы с другими версиями системы при помощи функций пользовательских библиотек, реализующих системные вызовы других версий ОС. Объясните, различается ли с точки зрения разработчика приложений реализация такой функции в виде библиотеки и в виде системного вызова.
6. На что должен обратить внимание разработчик библиотеки, если он решает реализовать свою функцию в библиотеке как альтернативную системному вызову? Что нужно дополнительно учитывать, если библиотеке необходимо использовать несколько системных вызовов для реализации этой функции?
7. Почему необходимо ограничивать объем работы, выполняемой обработчиком прерывания?
8. Дана система с n различными уровнями приоритетов прерываний. Какое максимальное количество прерываний может поддерживаться системой одновременно? Как влияет это количество на размеры различных стеков?

9. Архитектура процессора Intel 80x86 не предусматривает приоритетов прерываний. Для управления прерываниями используются две инструкции: CLI для запрещения всех прерываний и STI для обратного действия. Опишите алгоритм программной поддержки уровней приоритетов для этого процессора.
10. Когда определенный ресурс системы становится доступным, вызывается процедура wakeup(), которая будит все процессы, ожидающие его освобождения. Какие есть недостатки у описанного подхода? Какие вы видите альтернативные решения?
11. Представьте, что существует некоторый системный вызов, комбинирующий функции вызовов fork и exec. Определите его интерфейс и синтаксис. Каким образом этот вызов будет поддерживать такие возможности системы, как перенаправление ввода-вывода, выполнение в интерактивном или фоновом режиме, а также каналы?
12. Какие возникают проблемы при возврате ошибки от системного вызова exec? Как эти проблемы решены ядром?
13. Создайте функцию, позволяющую процессу ожидать завершения работы своего родителя (для любой реализации UNIX по вашему выбору).
14. Представьте, что процессу не нужно блокироваться, пока не будет завершено функционирование его потомка. Как можно убедиться в том, что потомок был удален из системы полностью после завершения?
15. Почему завершившийся процесс будит своего родителя?

2.11. Дополнительная литература

1. Allman, E., «UNIX: The Data Forms», Proceedings of the Winter 1987 USENIX Technical Conference, Jan. 1987, pp. 9–15.
2. American Telephone and Telegraph, «The System V Interface Definition (SVID)», Issue 2, 1987.
3. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
4. Kernighan, B. W., and Pike, R., «The UNIX Programming Environment», Prentice-Hall, Englewood Cliffs, NJ, 1984.
5. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3 BSD UNIX Operating System», Addison-Wesley, Reading, MA, 1989.
6. Institute for Electrical and Electronic Engineers, Information Technology, «Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language]», 1003.1–1990, IEEE, Dec. 1990.

7. Institute for Electrical and Electronic Engineers, POSIX P1003.4a, Threads Extension for Portable Operating Systems, 1994.
8. Salemi, C., Shah, S., and Lund, E., «A Privilege Mechanism for UNIX System V Release 4 Operating Systems», Proceedings of the Summer 1992 USENIX Technical Conference, Jun. 1992, pp. 235–241.
9. Thompson, K., «UNIX Implementation», The Bell System Technical Journal, Vol. 57, No. 6, Part 2, Jul.–Aug. 1978, pp. 1931–1946.

Глава 3
Нити и легковесные процессы

3.1. Введение

Технология использования процессов обладает двумя важными ограничениями. Прежде всего, некоторым приложениям необходимо выполнять несколько крупных независимых друг от друга задач, при этом используя одно и то же адресное пространство, а также другие ресурсы. Примерами таких приложений являются серверные части систем обслуживания баз данных, мониторы прохождения транзакций, а также реализации сетевых протоколов среднего и верхнего уровня. Такие процессы должны функционировать параллельно — и, следовательно, для них должна применяться иная программная модель, поддерживающая параллелизм. Традиционные системы UNIX могут предложить таким приложениям либо выполнять отдельные задачи последовательно, либо придумывать неуклюжие и малоэффективные механизмы поддержки выполнения таких операций.

Вторым ограничением применения традиционной модели является невозможность в полной мере задействовать преимущества многопроцессорных систем, так как процесс способен одновременно использовать только один процессор. Приложение может создать несколько отдельных процессов и выполнять их на имеющихся процессорах компьютера. Необходимо найти методики, позволяющие таким процессам совместно использовать память и ресурсы, а также синхронизироваться друг с другом.

Современные системы UNIX предлагают решение приведенных проблем при помощи различных технологий, реализованных в ОС с целью поддержки параллельного выполнения заданий. К сожалению, скудость стандартной терминологии сильно усложняет задачу описания и сравнения большого количества существующих механизмов параллелизации. В каждом варианте UNIX для их обозначения применяются свои термины, например *нити ядра*, *прикладные нити*, *прикладные нити, поддерживаемые ядром*, *C-threads*, *pthreads* и *«легковесные» процессы* (lightweight processes). Эта глава классифицирует используемую терминологию, объясняет основные понятия и описы-

вает возможности основных реализаций UNIX. В конце главы вы увидите анализ преимуществ и недостатков описываемых механизмов. А начнем мы с рассмотрения важности и полезности технологии нитей.

3.1.1. Причины появления технологии нитей

Многие программы выполняют отдельные крупные независимые задачи, которые не могут функционировать последовательно. Например, сервер баз данных находится в режиме приема и обработки множества запросов от клиентов. Так как поступающие запросы не требуется обрабатывать в каком-то определенном порядке, то их можно считать отдельными, независимыми друг от друга задачами, имеющими принципиальную возможность функционирования параллельно. Если система позволяет выполнять параллельно отдельные задачи приложения, то его производительность существенно увеличивается.

В традиционных системах UNIX такие программы используют несколько процессов. Большинство серверных приложений запускают *процесс прослушивания*, функция которого заключается в ожидании запросов клиентов. После поступления запроса процесс вызывает fork для запуска нового процесса, обрабатывающего запрос клиента. Так как эта задача часто требует проведения операций ввода-вывода, существует потенциальная возможность блокирования работы процесса, что дает некоторую степень одновременности выполнения даже на однопроцессорных системах.

Представим еще одну ситуацию. Имеется некоторое научное приложение, вычисляющее значения различных элементов массива, каждый из которых независим от остальных. Для решения задачи можно создать отдельные процессы для каждого элемента массива и выполнять их параллельно, выделив для каждого процесса отдельную машину или процессор в многопроцессорной системе. Однако даже на однопроцессорной машине имеет смысл разделить поставленную задачу между несколькими процессами. Если один из них заблокируется, ожидая окончания ввода-вывода или обработки ошибки доступа к страницам памяти, то другие в это время смогут продолжить функционирование. В качестве еще одного примера можно привести утилиту make системы UNIX, позволяющую пользователям параллельно производить компиляцию нескольких файлов, при этом каждый их них будет обрабатываться в отдельном процессе.

Использование приложением нескольких процессов имеет некоторые недостатки. При их создании происходят значительные перегрузки системы, так как вызов fork является весьма затратным (даже в том случае, если применяется совместное использование адресного пространства при помощи техники *копирования-при-записи*). Каждый процесс находится в своем адресном пространстве, поэтому для взаимодействия между ними необходимо применять такие средства, как сообщения, или разделяемую память. Еще

больше затрат требуется на разделение процессов между несколькими процессорами или компьютерами, передачу информации между такими процессами, ожидание завершения и сбор результатов их работы. В заключение необходимо также упомянуть о том, что система UNIX не обладает необходимыми базовыми элементами для совместного использования таких ресурсов, как сетевые соединения и т. д. Применение подобной модели оправдано в тех случаях, когда преимущества одновременной работы процессов перекрывают расходы на их создание и управление.

Приведенные примеры показывают неудобства модели процессов и необходимость применения более производительных средств параллельной обработки. Мы можем теперь определить концепцию независимого вычислительного блока в качестве части общей задачи приложения. Такие блоки относительно мало взаимодействуют друг с другом и, следовательно, требуют малого количества действий по синхронизации. Приложение может содержать один или несколько блоков. Описываемый вычислительный блок называется *нитью*. Процесс в традиционной системе UNIX является однонитевым, то есть все действия в нем происходят последовательно.

Механизмы, обсуждаемые в этой главе, подчеркивают ограничения технологии процессов. Они, конечно, тоже имеют определенные недостатки, описание которых вы увидите в конце главы.

3.1.2. Нити и процессоры

Преимущества многонитевых систем хорошо заметны, если такие системы сочетать с многопроцессорными архитектурами. Если каждая нить будет функционировать на отдельном процессоре, то можно говорить о настоящей параллельности работы приложения. Если количество нитей превышает количество имеющихся процессоров, то такие нити должны быть мультиплексированы на эти процессоры[1]. В идеальном случае, если приложение имеет n нитей, выполняющихся на n процессорах, то однонитевая версия той же программы на однопроцессорной системе будет тратить на выполнение задачи в n раз больше времени. На практике часть времени занимает создание, управление и синхронизация нитей, однако при использовании многопроцессорной системы эта величина стремится к идеальному соотношению.

На рис. 3.1 показан набор однонитевых процессов, выполняющихся на однопроцессорной машине. Операционная система создает некоторую иллюзию одновременности при помощи выделения каждому процессу некоторого определенного промежутка времени для работы (*квантование времени*), после чего происходит переключение на следующий процесс. В приведенном

[1] Это означает, что каждый процессор должен использоваться для обработки работающих нитей. Конечно, в один момент времени на одном процессоре может выполняться только одна нить.

примере первые три процесса являются задачами серверной части клиент-серверного приложения. Серверная программа при поступлении каждого нового запроса запускает еще один процесс. Все процессы обладают схожими адресными пространствами и совместно используют данные при помощи механизмов межпроцессного взаимодействия. Нижние два процесса созданы другим серверным приложением.

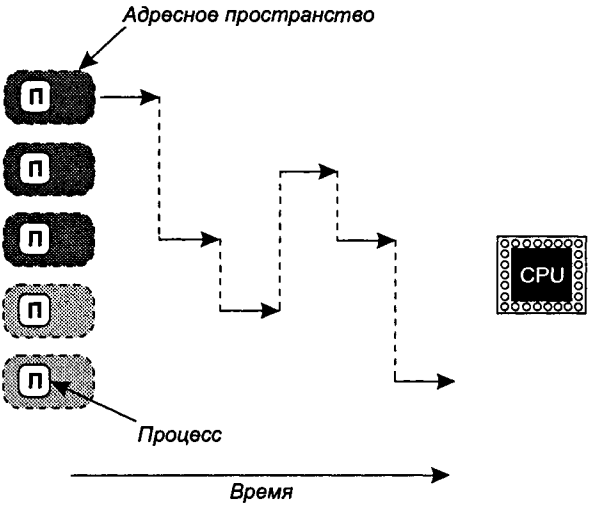


Рис. 3.1. Традиционная система UNIX: однонитевые процессы на однопроцессорной машине

На рис. 3.2 представлены два сервера, реализованные на нитях. Каждый сервер представляет собой единый процесс, имеющий несколько нитей, разделяющих между собой единое адресное пространство. Переключение контекста нитей в рамках одного процесса может обрабатываться либо ядром системы, либо на уровне прикладных библиотек, в зависимости от версии ОС. В обоих случаях в приведенном примере видны преимущества использования нитей. Исключение нескольких схожих адресных пространств приложений, порожденных для параллельной обработки чего-либо, сокращает нагрузку на подсистему памяти (даже при использовании современными системами такого метода разделения памяти, как *копирование при записи*, необходимо оперировать раздельными картами трансляции адресов для каждого процесса). Так как все нити приложения разделяют одно и то же адресное пространство, они могут использовать эффективные легковесные механизмы межнитевого взаимодействия и синхронизации.

Однако существуют и очевидные недостатки применения нитей. Однонитевому процессу не нужно заботиться о защите своих данных от других процессов. Многонитевые процессы должны следить за каждым объектом в собственном адресном пространстве. Если к объекту имеет доступ более

чем одна нить, то с целью предотвращения повреждения данных необходимо применять методы синхронизации.

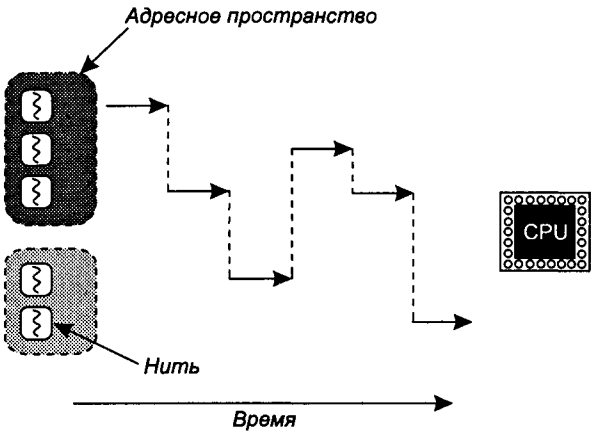


Рис. 3.2. Многонитевые процессы на однопроцессорной машине

На рис. 3.3 показаны два многонитевых процесса, выполняющиеся в многопроцессорной системе. Все нити каждого процесса совместно используют одно и то же адресное пространство, однако каждая из них выполняется на отдельном процессоре. Таким образом, все эти нити функционируют параллельно. Такой подход значительно увеличивает производительность системы, но, с другой стороны, существенно усложняет решение проблем синхронизации.

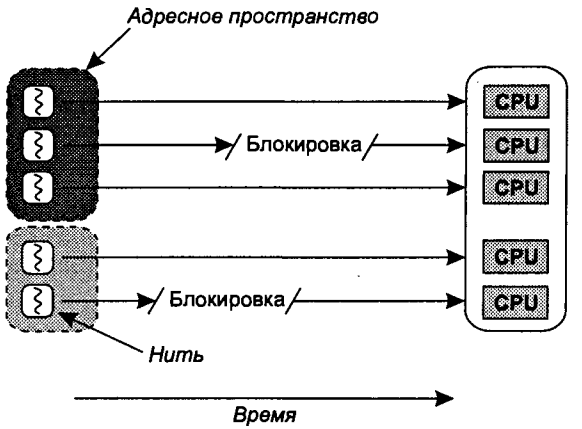


Рис. 3.3. Многонитевые процессы в многопроцессорной системе

Хотя оба средства (многонитевость и многопроцессорность) хорошо сочетаются друг с другом, их вполне можно использовать отдельно друг от друга. Многопроцессорные системы вполне пригодны для однонитевых приложе-

ний, так как в этом случае несколько процессов могут работать параллельно. Точно так же, имеются значительные преимущества в использовании многонитевых приложений на однопроцессорных системах. Если одна нить блокируется в ожидании окончания ввода-вывода или освобождения ресурса, другая нить может стать текущей, а приложение, таким образом, продолжит свое выполнение. Понятие нити более подходит для олицетворения встроенной одновременности выполнения в программе, нежели чем подгонки приложений под возможности многопроцессорных аппаратных архитектур.

3.1.3. Одновременность и параллельность

Для того чтобы понять основные типы элементов нитей, необходимо сначала определить разницу между терминами «одновременность» (concurrency) и «параллельность» (parallelism) [5]. *Параллельность* многопроцессорного приложения — это достигнутая им степень параллельного выполнения, которая в любом случае будет ограничиваться количеством процессоров, физически доступных приложению. Термин *одновременность* описывает максимальную степень параллельности, которую теоретически может достигать приложение с неограниченным количеством доступных процессоров. Эта степень зависит от того, как написано конкретное приложение, а также от количества нитей, которые могут выполняться одновременно, с доступными необходимыми им ресурсами.

Одновременность может поддерживаться на уровне либо системы, либо отдельного приложения. Ядро системы обеспечивает *системную одновременность* при помощи распознавания нескольких нитей внутри процесса (также называемых *«горячими» нитями, hot threads*) и планирования их выполнения независимо. Ядро разделяет такие нити между доступными процессорами. Приложение может воспользоваться преимуществами системной одновременности даже на однопроцессорных системах, поскольку если одна из его нитей будет блокирована в ожидании события или освобождения ресурса, ядро может назначить на выполнение другую нить.

Приложения могут обеспечивать *прикладную одновременность* через использование нитевых библиотек прикладного уровня. Такие нити, или *сопрограммы* (иногда называемые *«холодными» нитями*, cold threads), не распознаются ядром системы и должны управляться и планироваться самими приложениями. Этот подход не дает настоящей одновременности или параллельности, поскольку такие нити не могут в действительности выполняться одновременно, *но он является более естественной моделью выполнения программы для приложений, требующих одновременности.* Посредством использования неблокирующих системных вызовов приложение может одновременно поддерживать несколько взаимодействий в ходе своего выполнения. Использование прикладных нитей упрощает процесс выполнения программы, поскольку состояние таких взаимодействий считывается и размещается

в локальных переменных каждой нити (в стеке каждой нити), и при этом не используется глобальная таблица переменных.

Каждая модель одновременной обработки имеет определенные ограничения, заложенные в ее основе. Нити используются как средства организации работы и как средства использования многопроцессорных систем. Нити ядра позволяют производить параллельные вычисления на многопроцессорных системах, но они не подходят для структурирования приложений. Например, серверному приложению может потребоваться создание тысяч отдельных нитей, каждая из которых обрабатывает один клиентский запрос. Нити ядра потребляют такие важные ресурсы, как физическую память (так как многие реализации UNIX требуют постоянного хранения структур нитей в памяти); при этом, однако, они бесполезны для такого типа программ. С другой стороны, возможности нитей прикладного уровня полезны только для структурирования приложений, но они не позволяют выполнять код параллельно.

Во многих системах реализована модель *двойной одновременности*, которая совмещает в себе системную и прикладную одновременность. В этой модели нити процесса делятся на те, которые ядро распознает, и те, которые реализованы в библиотеках прикладного уровня и ядру не видимы. Прикладные нити позволяют синхронизировать одновременно исполняемые процедуры программы, не загружая систему дополнительными вызовами, и могут быть востребованы даже в системах, имеющих многонитевые ядра. Более того, уменьшение размера и функций ядра представляется хорошей дальнейшей стратегией развития UNIX, и разделение функциональности по поддержке нитей между ядром и нитевыми библиотеками как раз отвечает этим требованиям.

3.2. Основные типы нитей

Процесс представляет собой составную сущность, которую можно разделить на два основных компонента: набор нитей и набор ресурсов. *Нить* — это динамический объект, в процессе представленный отдельной точкой управления и выполняющий последовательность команд, отсчитываемую от этой точки[1]. Ресурсы, включающие адресное пространство, открытые файлы, полномочия, квоты и т. д., используются всеми нитями процесса совместно. Каждая нить, кроме того, обладает собственными объектами, такими как указатель команд, стек или контекст регистров. В традиционных системах UNIX процесс имеет единственную выполняющуюся нить. Многонитевые системы расширяют эту концепцию, позволяя каждому процессу иметь более одной выполняющейся нити.

[1] Проще говоря, выполняемая параллельно в рамках процесса часть программы. — *Прим. ред.*

Централизация ресурсов процесса имеет и некоторые недостатки. Представьте серверное приложение, которое производит различные файловые операции от имени удаленных клиентов. Для проверки прав доступа к файлам серверу при обслуживании клиента необходимо производить его идентификацию. Для этого сервер запускается в системе с полномочиями суперпользователя и периодически вызывает setuid, setgid и setgroups для временного изменения своих полномочий в соответствии с клиентскими. Использование многонитевости для такого сервера, сделанное с целью повышения одновременности его функционирования, может привести к серьезным проблемам безопасности. Так как процесс обладает всего одним набором полномочий, в один момент времени он может работать только от одного клиента. Таким образом, чтобы поддержать необходимый уровень безопасности, серверу приходится выполнять все системные вызовы последовательно, то есть в режиме выполнения одной-единственной нити.

Существуют различные типы нитей, каждая из которых обладает различными свойствами и применением. В этом разделе мы опишем три важнейших их типа: *нити ядра*, *легковесные процессы* и *прикладные нити*.

3.2.1. Нити ядра

Нити ядра не требуют связи с каким-либо прикладным процессом. Они создаются и уничтожаются ядром и внутри ядра по мере необходимости и отвечают за выполнение определенных функций. Такие нити используют совместно доступные области кода и глобальные данные ядра, но обладают собственным стеком в ядре. Они могут независимо назначаться на выполнение и используют стандартные механизмы синхронизации ядра, такие как sleep() или wakeup().

Нити ядра применяются для выполнения таких операций, как асинхронный ввод-вывод. Вместо поддержки каких-либо специальных механизмов ядро просто создает новую нить для обработки запросов для каждой такой операции. Запрос обрабатывается нитью синхронно, но для ядра представляется асинхронным событием. Нити ядра могут быть также использованы для обработки прерываний, подробнее об этом будет сказано в разделе 3.6.5.

Нити ядра являются малозатратными при создании и дальнейшем использовании. Единственными используемыми ими ресурсами являются стек ядра и область, в которой сохраняется контекст регистров на период приостановки работы нити (необходимо также поддерживать некую структуру данных, хранящую информацию для назначения ее на выполнение и синхронизацию). Переключение контекста между нитями ядра также происходит быстро, так что нет необходимости обновлять отображение памяти.

Применение нитей в ядре не является новым подходом. Такие системные процессы, как pagedaemon, в традиционных системах UNIX функционально

похожи на нити ядра. Процессы-демоны, наподобие nfsd (сервер Network File System), запускаются на прикладном уровне, однако после запуска полностью выполняются в ядре. После входа в режим ядра их прикладной контекст становится ненужным. Они также эквивалентны нитям ядра. Так как в традиционных системах UNIX отсутствовало понятие разделения применительно к представлению нитей ядра, такие процессы были вынуждены «таскать» за собою ненужный багаж, присущий традиционным процессам, в виде таких структур, как proc и user. Многонитевые системы позволили реализовать демоны намного проще в качестве нитей ядра.

3.2.2. Легковесные процессы

Легковесный процесс (или LWP, lightweight process) — это прикладная нить, поддерживаемая ядром. LWP является абстракцией высокого уровня, основанной на нитях ядра. Каждый процесс может иметь один или более LWP, любой из которых поддерживается отдельной нитью ядра (рис. 3.4). Легковесные процессы планируются на выполнение независимо от процесса, но совместно разделяют адресное пространство и другие ресурсы процесса. Они обладают возможностью производить системные вызовы и блокироваться в ожидании окончания ввода-вывода или освобождения ресурсов. На многопроцессорных системах процесс в состоянии использовать преимущества настоящей параллельной работы, так как каждый LWP может выполняться на отдельном процессоре. Однако даже на однопроцессорных системах применение LWP дает существенные преимущества, поскольку при ожидании ресурсов или окончания ввода-вывода блокируется не весь процесс в целом, а только отдельные LWP.

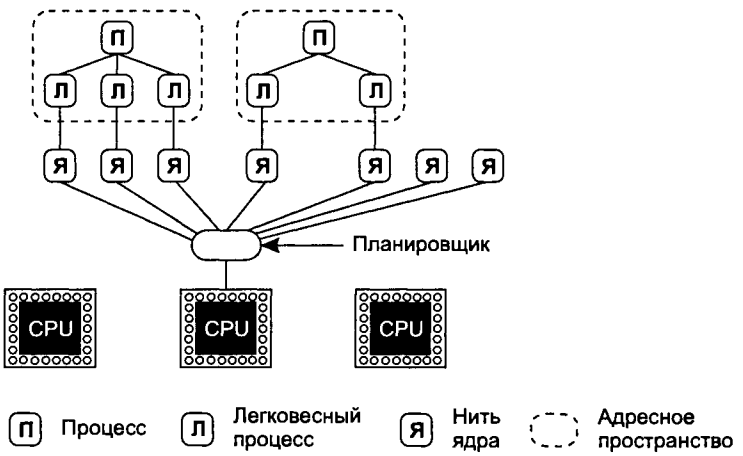


Рис. 3.4. Легковесные процессы

Кроме стека ядра и контекста регистров легковесному процессу необходимо поддерживать некоторое состояние задачи, что в первую очередь включает контекст регистров задачи, который необходимо сохранять перед тем, как обслуживание LWP будет прервано. Несмотря на то что каждый LWP ассоциирован с нитью ядра, некоторые нити ядра могут выполнять системные задачи и не относиться к какому-либо LWP.

Многонитевые процессы применяются прежде всего в тех случаях, когда каждая нить является полностью независимой и редко взаимодействует с другими нитями. Код приложения является полностью вытесняемым, при этом все LWP в процессе совместно используют одно и то же адресное пространство процесса. *Если доступ к каким-либо данным производится одновременно несколькими LWP, необходимо обеспечить некоторую синхронизацию доступа.* Для этого ядро системы предоставляет средства блокировки разделяемых переменных, при попытке прохождения в которые LWP будут блокированы. Такими средствами являются взаимные исключения (*mutual exclusion*, или *mutex*), атрибуты защиты, или защелки (*locks*), семафоры и условные переменные, которые будут более подробно рассмотрены в главе 7.

Рассказывая о легковесных процессах, необходимо также упомянуть и об их ограничениях. Многие операции над такими процессами, например создание, уничтожение и синхронизация, требуют применения системных вызовов. Однако системные вызовы по ряду причин являются весьма затратными операциями. Каждый вызов требует двух *переключений режима*: сначала из режима задачи в режим ядра и обратное переключение после завершения работы функции. При каждом переключении режима LWP пересекает *границу защиты* (protection boundary). Ядро должно скопировать все параметры системного вызова из пространства задачи в пространство ядра и привести их при необходимости в корректное состояние для защиты обработчиков этих вызовов от вредоносных или некорректно работающих процессов. При возврате из системного вызова ядро должно скопировать данные обратно в пространство задачи.

Когда легковесным процессам необходимо часто пользоваться разделяемыми данными, затраты на синхронизацию могут свести на нет увеличение производительности от их применения. Многие многопроцессорные системы поддерживают блокировку ресурсов, которая может быть активизирована из прикладного уровня при условии отсутствия удержания данного ресурса другой нитью [18]. Если нити необходим ресурс, который в данный момент недоступен, она может выполнить *цикл активного ожидания* (busy-wait) его освобождения без вмешательства ядра. Такие циклы подходят только для доступа к тем ресурсам, которые занимаются на короткие промежутки времени, в иных случаях необходимо блокирование нити. *Блокирование LWP требует вмешательства ядра* и вследствие этого является затратной процедурой.

Каждый легковесный процесс потребляет значительные ресурсы ядра, включая физическую память, отводимую под стек ядра. По этой причине система не может поддерживать большое количество LWP. Более того, так как система имеет единую реализацию LWP, такие процессы должны обладать достаточной универсальностью для поддержки наиболее типичных приложений. Таким образом, они могут содержать в себе такие свойства, которые окажутся ненужными большинству приложений. Применение LWP совершенно не подходит для приложений, использующих большое количество нитей или часто создающих и уничтожающих их. Наконец, легковесные процессы должны планироваться на выполнение ядром. Приложения, которым необходимо часто передавать управление от одной нити к другой, не могут делать это так легко, используя LWP. Применение легковесных процессов также способно повлечь некоторые неожиданные последствия, например, приложение имеет возможность монополизировать использование процессора, создав большое количество LWP.

Подводя итоги, скажем, что хотя ядро системы предоставляет механизмы создания, синхронизации и обработки легковесных процессов, за их правильное применение отвечает программист. Многим приложениям больше подходит реализация на прикладных нитях, описанных в следующем разделе.

ПРИМЕЧАНИЕ

Термин «легковесные процессы» (LWP) позаимствован из терминологии систем SVR4/MP и Solaris 2.x. Возможно, это не совсем точно, так как после появления четвертой версии SunOS термин LWP используется для обозначения прикладных нитей, описание которых вы можете увидеть в следующей главе. Однако в этой книге мы продолжим использование термина LWP для обозначения прикладных нитей, поддерживаемых ядром. В некоторых системах применяется термин «виртуальный процессор», который имеет то же значение, что и LWP[1].

3.2.3. Прикладные нити

Существует возможность поддержки нитей полностью на прикладном уровне, при этом ядру об их существовании ничего известно не будет. Указанная возможность реализована в таких библиотечных пакетах, как *C-threads* системы Mach и *pthreads* стандарта POSIX. Эти библиотеки содержат все необходимые функции для создания, синхронизации, планирования и обработки нитей без какой-либо специальной помощи ядра. Функционирование таких нитей не требует обслуживания ядром и вследствие этого является необычайно быстрым[2]. На рис. 3.5, *а* представлена схема функционирования прикладных нитей.

[1] Иногда легковесные процессы называют еще тяжеловесными нитями. — *Прим. ред.*

[2] Большинство нитевых библиотек требуют вмешательства ядра для поддержки средств асинхронного ввода-вывода.

Рисунок 3.5, *б* иллюстрирует совместное использование прикладных нитей и легковесных процессов, вместе создающих мощную среду выполнения программы. Ядро распознает, планирует на выполнение и управляет LWP. Библиотеки прикладного уровня мультиплексируют прикладные нити в LWP и предоставляют возможности для межнитевого планирования, переключения контекста и синхронизации без участия ядра. Получается, что библиотека функционирует как миниатюрное ядро для тех нитей, которыми она управляет.

Реализация прикладных нитей возможна по причине того, что прикладной контекст нити может сохраняться и восстанавливаться без вмешательства ядра. Каждая прикладная нить обладает собственным стеком в адресном пространстве процесса, областью для хранения контекста регистров прикладного уровня и другой важной информации, такой как маски сигналов. Библиотека планирует выполнение и переключает контекст между прикладными нитями, сохраняя стек и состояние регистров текущей нити и загружая стек и состояние регистров следующей по расписанию нити.

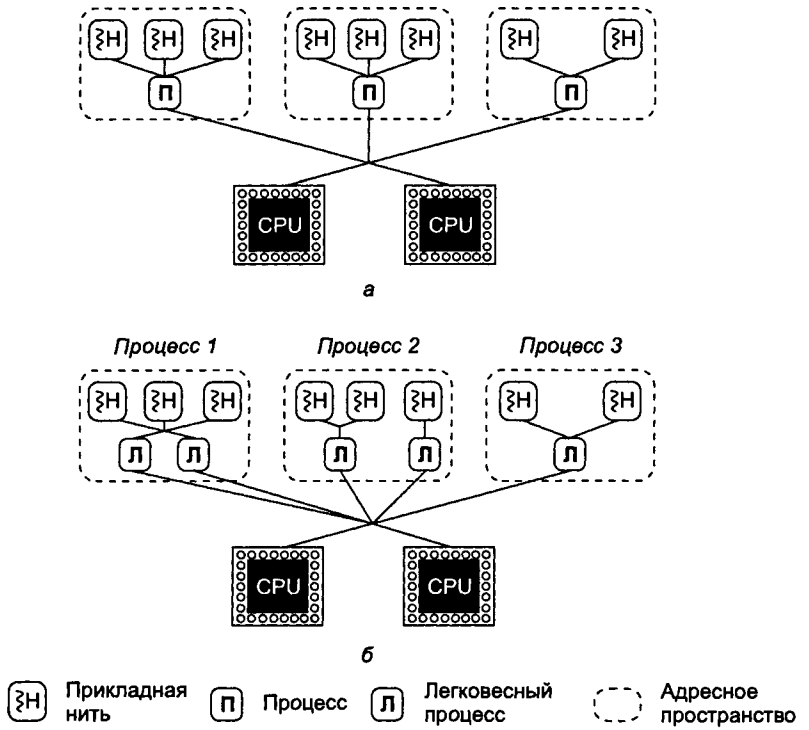


Рис. 3.5. Применение пользовательских нитей: *а* — прикладные нити обычных процессов; *б* — мультиплексирование прикладных нитей

Ядро системы отвечает за переключение процессов, так как только оно обладает привилегиями, необходимыми для изменения регистров управления памятью (регистров MMU). Поэтому прикладные нити не являются по-

настоящему планируемыми задачами, и ядро ничего не знает о них. Ядро просто планирует выполнение процесса (или LWP), содержащего в себе прикладные нити, который, в свою очередь, использует библиотечные функции для планирования выполнения своих нитей. Если процесс или LWP вытеснен кем-то, то той же участи подвергаются и все его нити. Точно так же, если нить делает блокирующий системный вызов, то он заблокирует и LWP этой нити. Соответственно, если процесс обладает всего одним LWP (или если прикладные нити реализованы на однонитевой системе), будут заблокированы все его нити.

Нитевая библиотека также включает в себя объекты синхронизации, обеспечивающие защиту совместно используемых структур данных. Такие объекты обычно содержат переменную блокировки (например, семафор) и очередь нитей, заблокированных по ней. Нити должны проверить блокировку перед тем, как начать доступ к защищенной структуре данных. Если объект уже заблокирован, библиотека приостановит выполнение нити, связав ее с очередью блокированных нитей, и передаст управление следующей нити.

Современные системы UNIX поддерживают механизмы асинхронного ввода-вывода, позволяющие процеесам выполнять ввод-вывод без блокирования. Например, в системе SVR4 для любого устройства, работа с которым может быть организована через STREAMS, предлагается ioctl-команда IO_SETSIG (технология STREAMS описана в главе 17). Последующие[1] операции write или read с потоком просто встают в очередь операций и возвращают значения без блокирования. Когда ввод-вывод завершится, процесс будет проинформирован об этом при помощи сигнала SIGPOLL.

Асинхронный ввод-вывод является весьма полезным средством, так как позволяет процессу выполнять другие задачи во время ожидания завершения ввода-вывода. Но, с другой стороны, его реализация ведет к усложнению модели выполнения программы. Одним из удачных решений является организация возможности асинхронной работы на уровне операционной системы, предоставляющая для выполнения программы синхронную среду. Нитевая библиотека обеспечивает такой подход посредством предоставления синхронного интерфейса, использующего асинхронный внутренний механизм. Каждый запрос является синхронным по отношению к делающей его нити, которая блокируется в ожидании окончания операции ввода-вывода. Однако при этом процесс будет продолжать свое выполнение, так как библиотека внутри себя преобразует запрос в виде асинхронной операции и назначит на выполнение следующую по очереди прикладную нить. После завершения ввода-вывода библиотека снова поместит в расписание выполнения приостановленную нить.

Использование прикладных нитей имеет несколько преимуществ. Они предоставляют более естественный способ программирования многих приложе-

[1] То есть сделанные после вышеуказанного вызова ioctl. — *Прим. ред.*

ний, например таких, как оконные системы. Прикладные нити также обеспечивают синхронный подход к выполнению программы, поскольку все сложные асинхронные операции скрыты в недрах нитевых библиотек. Одно только это делает применение прикладных нитей весьма привлекательным, даже в системах, не обладающих поддержкой нитей на уровне ядра. Система может предлагать разработчику несколько различных нитевых библиотек, каждая из которых оптимизирована для различных классов приложений.

Важнейшим преимуществом прикладных нитей является их производительность. Эти нити являются легковесными и не потребляют ресурсов ядра, кроме связанных с LWP. В основе производительности работы прикладных нитей лежит реализация всей их функциональности на прикладном уровне без применения системных вызовов. Такой подход не требует дополнительной обработки системных прерываний и перемещения параметров и данных через границы защиты. Одним из важных параметров нити является *критический размер* нити [4], который показывает тот объем работы, который нить должна выполнить для того, чтобы оправдать свое существование в качестве отдельной сущности. Этот размер зависит от затрат на создание и использование нити. Для прикладных нитей критический размер составляет порядка нескольких сотен инструкций, количество которых может быть сокращено менее чем на сотню, в зависимости от используемого компилятора. Прикладные нити требуют значительно меньшего количества времени для создания, уничтожения и синхронизации. Таблица 3.1 показывает длительность различных операций, производимых процессами, LWP и прикладными нитями на машине SPARCstation 2 [21].

Таблица 3.1. Длительность операций, производимых прикладными нитями, LWP и процессами (на SPARCstation 2)

	Создание, мкс	**Синхронизация с использованием семафоров, мкс**
Пользовательская нить	52	66
LWP	350	390
Процесс	1700	200

Однако прикладные нити обладают и рядом ограничений, большинство из которых является следствием полного разделения информации между ядром и нитевой библиотекой. Так как ядро системы ничего не знает о прикладных нитях, оно не может использовать свои механизмы для их защиты друг от друга. Каждый процесс имеет свое адресное пространство, защищаемое ядром от несанкционированного доступа других процессов. Прикладные нити лишены такой возможности, так как функционируют в общем адресном пространстве процесса. Нитевая библиотека должна обеспечивать средства синхронизации, необходимые для совместной работы нитей.

Использование модели разделенного планирования может стать причиной возникновения множества других проблем. Нитевая библиотека занимается планированием выполнения прикладных нитей, ядро планирует выполнение процессов или легковесных процессов, в которых эти нити функционируют, и в итоге никто из них не знает о действиях друг друга. Например, ядро может вытеснить LWP, чья прикладная нить находится в области, защищаемой *циклической блокировкой* (spin lock), удерживая эту блокировку активной. Если иная прикладная нить иного LWP попытается снять эту блокировку, то она перейдет в цикл активного ожидания до тех пор, пока удерживающая циклическую блокировку нить не получит возможности работать снова. С другой стороны, так как ядро системы не знает относительных приоритетов прикладных нитей, оно вполне может вытеснить один легковесный процесс с выполняемой внутри нитью высокого приоритета, поставив на выполнение другой LWP, нить которого имеет более низкий приоритет.

Механизмы синхронизации, организованные на прикладном уровне, могут в некоторых случаях работать некорректно. Большинство приложений создаются с предположением, что все работающие нити периодически попадают в очередь на выполнение. Это действительно так, если каждая нить находится в отдельном LWP, но предположение может не выполняться в тех случаях, когда некоторое количество прикладных нитей мультиплексируется в меньшем количестве LWP. Так как LWP может блокироваться в ядре, когда его прикладная нить делает блокирующий системный вызов, такой LWP процесса умеет останавливать свою работу даже тогда, когда остаются работающие нити, а в системе — доступные процессоры. Разрешить эту проблему можно при помощи использования механизмов асинхронного ввода-вывода.

И наконец, следует упомянуть о том, что без явной поддержки на уровне ядра прикладные нити в состоянии увеличить одновременность выполнения, но не могут увеличить параллельность. Даже на многопроцессорных системах прикладные нити, разделяющие между собой один легковесный процесс, не могут выполняться параллельно.

В этом разделе были описаны три основных, наиболее используемых типа нитей. Нити ядра являются объектами самого нижнего уровня и невидимы для приложений. Легковесные процессы — это нити, видимые на прикладном уровне, но распознаваемые при этом ядром и опирающиеся на нити ядра. Прикладные нити представляют собой объекты высокого уровня, не видимые ядром. Они могут использовать легковесные процессы (если таковые поддерживаются системой) или реализовываться в обычных процессах UNIX без специальной поддержки ядром. И прикладные нити, и LWP имеют некоторые существенные недостатки, ограничивающие их область применения. В разделе 3.5 вы увидите описание новой фундаментальной структуры, основанной на *активациях планировщика*, которая показывает многие из этих проблем. Однако сначала мы затронем некоторые вопросы, касающиеся прикладных нитей и LWP, подробнее.

3.3. Легковесные процессы: основные проблемы

Существует несколько различных факторов, влияющих на устройство легковесных процессов. В первую очередь, необходимо сохранять семантику, принятую в системах UNIX, по крайней мере, для однонитевых вариантов. Это означает, что процесс, содержащий единственный LWP, должен функционировать как традиционный процесс UNIX. (Стоит снова упомянуть, что под LWP мы понимаем прикладные нити, поддерживаемые ядром, а не легковесные процессы системы SunOS 4.0, которые являются объектами полностью прикладного уровня.)

Существуют различные области, где концепцию UNIX нельзя легко перенести на многонитевую систему. Следующие разделы посвящены описанию возникающих проблем и способам их решения.

3.3.1. Семантика вызова fork

Системные вызовы в многонитевых средах принимают несколько необычные значения. Многие вызовы, имеющие отношение к созданию процессов, манипуляциям над адресным пространством или действиям над ресурсами процесса (такими как открытые файлы), должны быть переписаны заново. Есть две важные рекомендации. Во-первых, системный вызов должен придерживаться традиционной семантики системы UNIX в случае применения одной нити. Во-вторых, при использовании многонитевых процессов системный вызов обязан вести себя приемлемым образом, не уходя далеко от его семантики на однонитевых системах. Помня об этих рекомендациях, давайте проведем анализ некоторых важнейших системных вызовов, которые необходимо изменить для многонитевых реализаций.

В традиционных системах UNIX вызов fork создает процесс-потомок, являющийся точной копией его родителя. Их единственное различие заключается в необходимости отличия потомка от его родителя. Семантика вызова fork полностью отвечает требованиям однонитевого процесса. В случае многонитевых процессов имеется дополнительный параметр, позволяющий дублировать либо все LWP родителя, либо только тот из них, что вызвал функцию fork.

Представьте, что вызову fork необходимо произвести копирование вызывающего его LWP в новый процесс, и только его. Такой вариант является более эффективным. Он также весьма удобен в тех случаях, если процесс-потомок после своего появления запустит в себе новую программу при помощи exec. Однако подобное взаимодействие имеет и некоторые проблемы [20]. LWP часто используются для поддержки нитевых библиотек прикладного уровня. Такие библиотеки представляют каждую прикладную нить в виде структуры данных в пространстве процесса. Если вызов fork продублирует только вызывающий его LWP, то новый процесс будет содержать прикладные нити, не

относящиеся ни к одному своему LWP. Более того, процесс-потомок не должен пытаться снять блокировку, удерживаемую нитями, не существующими в нем, так как это может привести к состоянию клинча. В реальности иногда сложно избежать конфликта, так как библиотеки часто создают скрытые нити, о которых ничего не знает разработчик приложения.

Представим иной случай, когда вызов fork дублирует все LWP родительского процесса. Такой вариант наиболее предпочтителен в случае, когда необходимо сделать именно копию всего процесса, нежели выполнить новую программу. Однако и в этом случае возникают определенные проблемы. Какой-либо LWP родителя может быть заблокирован системным вызовом, и это состояние не будет определено в потомке. Возможность обойти данную ситуацию заключается в том, чтобы заставить системный вызов вернуть код ошибки EINTR (системный вызов прерван), что позволило бы LWP сделать его заново по мере необходимости. LWP также может иметь открытые сетевые соединения. Закрытие соединения в потомке может стать причиной отправки на удаленный узел незапланированных сообщений. Некоторые LWP умеют обрабатывать внешние общие структуры данных, которые могут быть повреждены при клонировании такого LWP вызовом fork.

Ни одно из решений не в силах правильно обработать все возможные ситуации. Во многих системах определенный компромисс достигнут при помощи двух различных вариантов fork, один из которых применяется для дублирования процесса полностью, а второй — копирует только одну нить. Для последнего варианта в таких системах определен набор безопасных функций, которые могут быть вызваны потомком перед выполнением exec. Альтернативным вариантом является разрешение на регистрацию процессом одного или нескольких обработчиков fork, которые являются функциями, запускаемыми в родителе или потомке до или после вызова fork в зависимости от параметров, указанных при их регистрации.

3.3.2. Другие системные вызовы

Для корректной работы в многонитевых системах необходимо пересматривать не только fork, но и многие другие системные вызовы. Все LWP процесса разделяют между собой общий набор файловых дескрипторов. Это может стать причиной возникновения конфликта в случае, если один из LWP закроет файл, который в текущий момент времени считывается или в него ведется запись другим LWP. Файловый *указатель на смещение*[1] также используется совместно всеми нитями через дескриптор, поэтому применение функции lseek одним из LWP повлияет на работу с этим файлом всех остальных нитей. Эта проблема проиллюстрирована на рис. 3.6. Легковесному процессу Л1

[1] Используется при считывании из файла или записи в файл для запоминания той позиции, откуда необходимо начинать чтение или запись при следующем вызове read или write. — *Прим. ред.*

необходимо прочесть данные из файла, начиная со смещения off1, для чего он вызывает функцию lseek, а затем — read. Между двумя означенными вызовами другой процесс Л2 применяет lseek в отношении того же файла, указывая при этом другое смещение. Такая ситуация приведет к тому, что Л1 считает не те данные. Приложение должно решать подобные проблемы самостоятельно, используя какой-либо протокол блокирования файлов. Альтернативным решением являются механизмы ядра системы, которые позволяют производить произвольный ввод-вывод атомарно (см. подробнее в разделе 3.6.6).

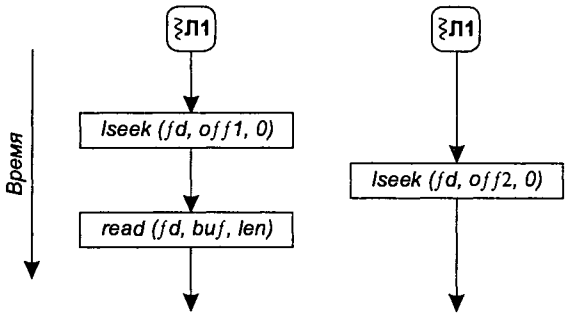


Рис. 3.6. Проблемы, возникающие при одновременном доступе к файлу

Каждый процесс имеет один текущий рабочий каталог и использует одну структуру пользовательских полномочий. Так как полномочия могут изменяться в любой момент времени, ядро должно использовать их атомарно и только однажды перед системным вызовом.

Все LWP процесса используют совместно одно и то же адресное пространство и могут манипулировать им одновременно при помощи различных системных вызовов, таких как mmap или brk. Такие вызовы должны быть безопасными (в отношении нитей) во избежание повреждения адресного пространства процесса. Программистам следует внимательно относиться к последовательности таких операций, так как в противном случае результат может быть непредсказуем.

3.3.3. Доставка и обработка сигналов

В системах UNIX доставка и обработка сигналов производится на уровне процесса. В многонитевых системах необходимо определять, какой из LWP процесса будет заниматься обработкой сигналов. При использовании прикладных нитей имеется аналогичная проблема: после того, как ядро передаст сигнал в LWP, нитевая библиотека должна определить, в какую нить его направить. Существует несколько вариантов решения данной проблемы:

- пересылка сигналов каждой нити;
- объявление одной из нитей процесса «главной», после чего все сигналы передаются только этой нити;

- отправка сигналов в любую произвольно выбранную нить;
- использование эвристических методов для определения, какой из нитей необходимо отправить данный сигнал;
- создание новой нити для обработки каждого сигнала.

Первый метод является очень затратным, и, более того, он несовместим с большинством обычных ситуаций, в которых используются сигналы. Однако в некоторых случаях он весьма удобнен. Например, если пользователь нажимает комбинацию Ctrl+Z на терминале, он может желать приостановки всех нитей процесса. Второй метод приводит к асимметричной обработке нитей, что несовместимо с современным подходом к нитям и с симметричными многопроцессными системами, часто ассоциируемыми с многонитевыми ядрами. Последнее решение, приведенное в списке, подходит только для определенных ситуаций.

Выбор между двумя оставшимися методами зависит от природы выработанного сигнала. Некоторые сигналы, например SIGSEGV (ошибка сегментации) и SIGILL (непредусмотренное исключение), создаются вследствие действий нити. Наиболее удобным решением в данном случае представляется доставка такого сигнала той нити, которая и стала причиной его возникновения. Другие сигналы, такие как SIGSTP (сигнал остановки, вырабатываемый терминалом) или SIGINT (сигнал прерывания), создаются при возникновении внешних событий и не могут быть как-то ассоциированы с конкретной нитью процесса.

Еще одним аспектом, о котором стоит упомянуть, является применяемый метод обработки и маскирования сигналов. Должны ли все нити использовать общий набор обработчиков сигналов или каждая будет определять свой собственный? Хотя последний вариант является более гибким и универсальным, он привносит каждой нити дополнительные затраты, что противоречит главной цели применения многонитевых процессов. Такие же проблемы возникают при маскировании сигналов, поскольку обычно маскирование происходит с целью защиты важных участков кода. Следовательно, лучшим вариантом представляется разрешение каждой нити на указание собственной маски сигналов. Перегрузки, возникающие при применении таких масок, менее значительны и поэтому более приемлемы.

3.3.4. Видимость

Важно определить, в какой степени LWP будет видимым вне процесса. Бесспорно, ядро системы знает о существующих LWP и планирует их выполнение независимо. Однако большинство реализаций систем не позволяет процессам обладать информацией о конкретных LWP других процессов, а также взаимодействовать с ними.

Вместе с тем внутри процесса необходимо предоставлять какому-либо LWP возможность получать информацию о существовании остальных LWP в рамках своего процесса. Многие системы предлагают для этой цели специаль-

ные системные вызовы, которые позволяют одному LWP отправлять сигналы другому LWP, принадлежащему тому же самому процессу.

3.3.5. Рост стека

Если какой-нибудь из процессов в системе UNIX переполняет свой стек, в результате этого возникает ошибка нарушения сегментации. Ядро распознает появление таких ситуаций в сегменте стека и автоматически увеличивает размер стека[1], не посылая никаких сигналов процессу.

Многонитевые процессы обладают несколькими стеками, по одному на каждую прикладную нить. Эти нити размещаются на прикладном уровне при помощи нитевых библиотек. Если ядро системы попытается расширить стек, то возникнет определенная проблема, так как такая операция может привести к конфликту с обработчиком стека в нитевой библиотеке прикладного уровня.

Даже в многонитевой системе ядро не имеет никакого представления о стеках прикладных нитей[2]. Такие стеки не всегда являются специальными областями и могут быть взяты прямо из кучи. Обычно если нить указывает размер необходимого ей стека, то библиотека может защитить стек от переполнения при помощи размещения страницы памяти, защищенной от записи, сразу после конца стека. Такой подход приводит к ошибке защиты при возникновении переполнения стека[3], и в этом случае ядро системы посылает сигнал SIGSERV соответствующей нити. После этого нить может либо увеличить размер стека, либо решить проблему иным путем[4].

3.4. Нитевые библиотеки прикладного уровня

При разработке пакетов функций для работы с прикладными нитями необходимо найти ответы на следующие два важных вопроса: какого рода программный интерфейс библиотеки будет представлен программисту и как такой пакет

---

[1] До определенного установленного лимита. В системе SVR4 размер стека ограничивается значением переменной RLIMIT_NOFILE. Эта переменная содержит *жесткие* и *мягкие границы*. Для получения значения границ применяется системный вызов getrlimit. При помощи вызова setrlimit можно уменьшить жесткий лимит, а также уменьшить или увеличить мягкую границу до значения, не превышающего жесткий лимит.

[2] Некоторые многонитевые системы, например SVR4.2/MP, имеют средства, позволяющие автоматически увеличивать стек прикладной нити.

[3] Помещаемые в стек данные попадут в область стоящей за стеком страницы. — *Прим. ред.*

[4] Конечно, такой сигнал должен обрабатываться в специальном стеке, раз уже обычный стек не имеет свободного места для функционирования обработчика сигнала. В современных системах UNIX существуют способы задания приложением альтернативного стека для обработки сигналов (см. подробнее в разделе 4.5).

может быть реализован при помощи средств, предлагаемых конкретной операционной системой. Существуют различные варианты нитевых библиотек, например Chorus [2], Topaz [23] и *C-threads* ОС Mach [7]. Группой P1003.3a IEEE стандартов POSIX было разработано несколько предварительных вариантов пакета функций для работы с нитями под названием pthreads [13]. Современные системы UNIX должны поддерживать pthreads для совместимости с этими стандартами (см. подробнее в разделе 3.8.3).

3.4.1. Программный интерфейс

Интерфейс, обеспечиваемый пакетами функций для работы с нитями, должен обладать несколькими важными средствами. Он должен поддерживать большой набор различных операций над нитями, таких как:

- создание и уничтожение нитей;
- перевод нитей в режим ожидания и их восстановление в работоспособное состояние;
- назначение приоритетов для отдельных нитей;
- планирование выполнения нитей и переключение контекста;
- синхронизацию действий при помощи таких средств, как семафоры или взаимные исключения;
- обмен сообщениями между нитями.

Пакеты функций для работы с нитями должны по возможности минимизировать участие ядра, так как переключение между режимом задачи и режимом ядра может быть весьма затратной процедурой. Поэтому нитевые библиотеки предоставляют столько возможностей, сколько могут. Обычно ядро системы не обладает информацией о прикладных нитях, однако нитевые библиотеки могут использовать системные вызовы для реализации ряда своих возможностей. Из этого вытекает ряд важных моментов. Например, приоритет нити не имеет никакого отношения к приоритету процесса или LWP этой нити, назначенному в расписании ядра. Приоритет нити имеет смысл только внутри своего процесса и используется *планировщиком нитей* для выбора одной из них на выполнение.

3.4.2. Реализация нитевых библиотек

Реализация конкретной библиотеки зависит от средств многонитевости, предоставляемых ядром системы. Многие существующие пакеты функций для работы с нитями созданы для традиционных вариантов UNIX, вообще не имеющих специальной поддержки нитей. В таких системах нитевые библиотеки функционируют как миниатюрные ядра, обрабатывая самостоятельно всю информацию о состоянии каждой нити и производя все операции над

ними на прикладном уровне. Хотя этот подход обеспечивает довольно эффективную последовательную обработку, он дает и некоторую степень одновременности при использовании средств асинхронного ввода-вывода системы.

Во многих современных системах ядро поддерживает многонитевые процессы через LWP. В таком случае библиотеки прикладных нитей могут быть реализованы различными способами:

- Каждой нити назначается свой легковесный процесс. Вариант наиболее прост для реализации, но требует большого количества ресурсов ядра и дает лишь малое увеличение производительности. Он также требует участия ядра во всех операциях синхронизации и планирования выполнения.
- Прикладные нити мультиплексируются в меньший набор легковесных процессов. Этот вариант более эффективен, так как требует меньшего количества ресурсов ядра. Он наиболее подходит в тех случаях, когда все нити процесса примерно эквивалентны друг другу. Однако здесь нет простого способа, гарантирующего предоставление ресурсов определенной нити.
- Связанные и несвязанные нити смешиваются в одном процессе. Такой вариант дает возможность приложению полностью использовать параллельность и одновременность, предоставляемые системой. Он также позволяет обрабатывать преимущественно связанные нити посредством повышения уровня приоритетов LWP, их содержащего, или вообще давая их LWP эксклюзивное право на пользование процессором. О связанных и несвязанных нитях подробнее в разделе 3.6.3.

Нитевая библиотека содержит алгоритм планирования, по которому выбирается очередная нить для выполнения. Он обрабатывает приоритеты и состояния каждой нити, не зависящие от состояния или приоритета LWP, внутри которого находятся эти нити. На рис. 3.7 показан пример шести прикладных нитей, мультиплексированных в два LWP. Библиотека планирует выполнение по одной нити в каждом LWP. Такие нити (в данном случае H5 и H6) находятся в *выполняющемся* состоянии, даже если сам легковесный процесс, их содержащий, был заблокирован системным вызовом или был вытеснен и находится в ожидании своей очереди на выполнение.

Нить (такая как H1 или H2 на рис. 3.7) изменяет свое состояние на *блокированное* в том случае, если попытается использовать объект синхронизации, заблокированный другой нитью. После освобождения объекта библиотека разблокирует нить и переведет ее в очередь на выполнение. Нити H3 и H4 уже находятся в состоянии *готовности* и ожидают своей очереди на выполнение. Планировщик нитей выбирает на выполнение нить из этой очереди, исходя из приоритета и связанного с ней LWP. *Этот механизм очень схож с алгоритмами планирования и ожидания ресурсов ядра.* Как уже говорилось ранее, нитевая библиотека функционирует как миниатюрное ядро для нитей, которыми управляет.

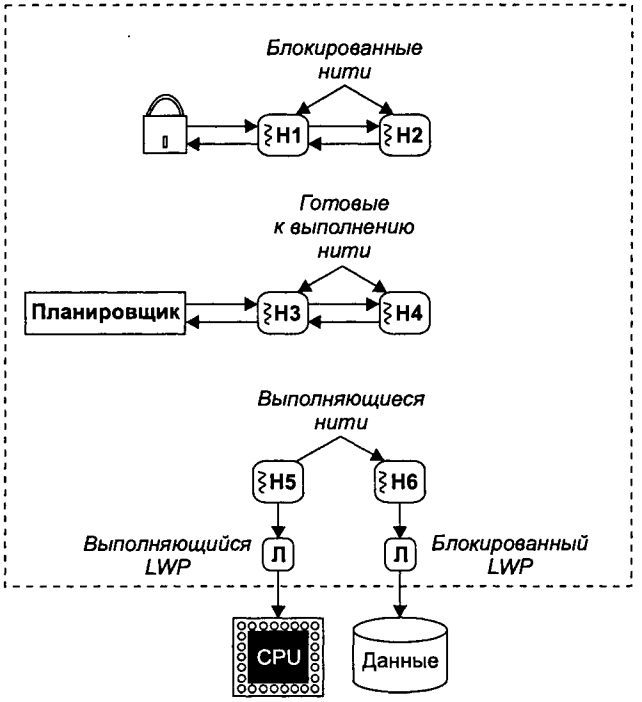


Рис. 3.7. Состояния прикладных нитей

Более подробно о реализациях прикладных нитей можно прочесть в [9], [18] и [20].

3.5. Активации планировщика

В двух предыдущих разделах описывались преимущества и недостатки легковесных процессов и прикладных нитей. Ни одна из приведенных моделей не является полностью удовлетворительной. Разработчики хотят сочетать производительность прикладных нитей и их гибкость. Однако такие нити имеют меньшую функциональность, чем легковесные процессы, поскольку не имеют интеграции с ядром системы. В [1] описывается совершенно новая архитектура нитей, в которой используются преимущества обеих моделей. Эта архитектура получила признание среди специалистов по операционным системам и появилась в коммерческих реализациях механизмов нитей от таких производителей, как SGI[1] [4].

Основной принцип новой модели заключался в тесной интеграции прикладных нитей и ядра. Ядро отвечает за выделение процессора, а нитевая

[1] Silicon Graphics. — *Прим. ред.*

библиотека — за планирование. Библиотека информирует ядро системы о событиях, которые требуют выделения процессора. Она может запросить либо дополнительные процессоры, либо оставить свой текущий. Ядро полностью контролирует использование процессоров и может периодически «забирать» процессоры, предоставляя их другим процессам.

Библиотека, получив некоторые процессоры, всецело контролирует нити и планирует их выполнение на этих процессорах. Если ядро забирает один из процессоров, то оно проинформирует об этом библиотеку, которая переназначит нити соответствующим образом. Если нить блокируется в ядре, процесс не потеряет занимаемый им процессор. Ядро сообщит об этом факте библиотеке, которая немедленно поставит на выполнение другую прикладную нить на тот же самый процессор.

Для реализации описываемой модели необходимо ввести два новых понятия *—обратного вызова* (upcall) и *активации планировщика* (scheduler activation). Обратный вызов — это вызов, сделанный ядром системы в нитевую библиотеку. Активация планировщика — контекст выполнения, который может быть использован для выполнения прикладной нити. Он подобен LWP и имеет собственный стек в пространстве процесса и стек ядра. Когда ядро осуществляет обратный вызов, оно передает в библиотеку активацию, которая будет использована для обработки события, выполнения новой нити или для какого-либо системного вызова. Ядро системы не квантует время активаций на процессоре. В любой момент времени процесс обладает одной активацией для каждого процессора из назначенных ему.

Отличительной особенностью модели активаций планировщика является ее обработка блокирующих операций. Когда прикладная нить блокируется в ядре, последнее производит новую активацию и сообщает об этом библиотеке посредством обратного вызова. Библиотека сохраняет состояние нити от предыдущей активации и информирует ядро системы о том, что эту активацию можно использовать заново. Затем библиотека переходит к выполнению следующей нити с новой активацией. Когда блокирующая операция завершается, ядро производит еще один обратный вызов, указывающий библиотеке на произошедшее событие. Такой вызов требует новой активации. Ядро может назначить новый процессор для выполнения этой активации или вытеснить одну из текущих активаций данного процесса. В последнем случае вызов оповестит библиотеку о двух событиях: во-первых, о том, что можно продолжить выполнение изначальной нити (которая была блокирована), и во-вторых, о том, что нить, выполняющаяся ранее на этом процессоре, была вытеснена. Библиотека поместит обе нити в список нитей, готовых к выполнению, и затем решит, какая из них будет выполняться первой.

Технология активаций планировщика обладает рядом преимуществ. Прежде всего, активации происходят очень быстро, так как большинство операций не требует участия ядра. В работе [1] приведены измерения, которые показа-

ли, что пакеты функций работы с нитями, основанные на активациях, функционируют производительнее, чем другие типы нитевых библиотек. Так как ядро системы информирует библиотеку о событиях блокирования или вытеснения, библиотека может самостоятельно принимать решения по планированию и синхронизации, а также избегать взаимных блокировок и некорректной семантики. Например, если ядро заберет процессор, который в текущий момент занят нитью, находящейся в состоянии циклической блокировки, библиотека может переключить выполнение этой нити на другой процессор и выполнять ее там до тех пор, пока ожидаемый нитью ресурс не снимет блокировку.

Оставшаяся часть главы посвящена описанию реализаций нитей в системах Solaris, SVR4, Mach и Digital UNIX.

3.6. Многонитевость в Solaris и SVR4

Корпорация Sun Microsystems начала поддерживать нити на уровне ядра в системе Solais версий 2.x[1]. Компания UNIX System Laboratories для своей ОС SVR4.2/MP приняла технологию нитей системы Solaris. Разработанная архитектура предлагает большое количество базовых элементов как для уровня ядра, так и на прикладном уровне, что позволяет создавать приложения с широкими возможностями.

Система Solaris поддерживает нити ядра, легковесные процессы и прикладные нити. Процесс может обладать несколькими сотнями нитей, сохраняя параллельность выполнения программы. Нитевая библиотека проведет мультиплексирование этих нитей в меньшее количество легковесных процессов. Существует возможность управления количеством LWP для оптимизации использования ресурсов системы. Также существует возможность группирования некоторых нитей в отдельные легковесные процессы (см. подробнее в разделе 3.6.3).

3.6.1. Нити ядра

Нить ядра в системе Solaris — это основной легковесный объект, который может независимо планироваться и отправляться на выполнение одному из процессоров системы. Такой объект не нуждается в ассоциации с каким-либо процессом, он может быть создан, запущен и уничтожен ядром при помощи специальных функций. В результате ядру системы не нужно переотображать виртуальное адресное пространство при переключении нитей ядра [16]. Следовательно, переключение контекста нити ядра менее затратно, чем переключение контекста процесса.

[1] Нити ядра были представлены в Solaris 2.0, а доступный разработчикам интерфейс в Solaris 2.2.

Нить ядра требует минимального количества ресурсов в виде небольшой структуры данных и стека. Структура данных нити ядра содержит следующую информацию:

- сохраненная копия регистров ядра;
- приоритет и информация, связанная с расписанием;
- указатели на местонахождение нити в очереди планировщика или, если выполнение нити блокировано, в очереди *ожидания ресурсов*;
- указатель на стек;
- указатели на связанные с нитью структуры lwp и proc (равны NULL, если нить выполняется не в рамках LWP);
- указатели на очередь всех нитей процесса и очередь всех нитей в системе;
- информация об LWP, связанном с нитью, если таковой существует (см. раздел 3.6.2).

Ядро системы Solaris организовано как набор внутренних нитей. Некоторые из них выполняют легковесные процессы, другие отвечают за внутренние функции ядра. Нити ядра являются полностью вытесняемыми. Они могут относиться к любому классу расписания задач системы (см. раздел 5.5), в том числе и к классу реального времени. Нити используют специализированные версии базовых элементов синхронизации (семафоров, условий и т. д.), защищающих от *инверсии приоритетов*, то есть ситуации, при которой нить с более низким приоритетом удерживает ресурс, необходимый нити, имеющей более высокий приоритет, что замедляет ее выполнение. Такие средства системы будут подробно описаны в разделе 5.6.

Нити ядра используются при проведении асинхронных операций, таких как отложенная запись на диск, выполнение обслуживающих процедур STREAMS и *отложенные вызовы* (callouts, см. раздел 5.2.1). Это позволяет ядру системы задавать приоритет каждой из подобных операций (посредством установки приоритета нити) и исходя из этих приоритетов планировать их выполнение. Нити также используются для поддержки легковесных процессов. Для этого каждый процесс LWP «прикреплен» к ядру нитью ядра (однако не все нити ядра имеют свой LWP).

3.6.2. Реализация легковесных процессов

Легковесные процессы обеспечивают многонитевое выполнение внутри одного процесса. Планирование выполнения LWP происходит независимо, и такие процессы могут выполняться параллельно на многопроцессорных системах. Каждый LWP связан со своей собственной нитью ядра, такая связь не прерывается на протяжении всего жизненного цикла процесса.

Традиционные структуры proc и user недостаточны для представления многонитевых процессов. Данные в этих структурах должны быть разделены на информацию, касающуюся каждого процесса и каждого LWP. В системе Solaris структура proc используется для хранения всех данных каждого процесса, в том числе и процессо-зависимой части традиционной области u.

Вдобавок к структурам, описывающим процесс в ядре, появляется новая структура — lwp, которая хранит информацию о каждой LWP-составляющей контекста процесса. Структура lwp содержит следующую информацию:

- сохраненные значения регистров прикладного уровня (когда LWP не выполняется);
- аргументы системного вызова, результаты работы и код ошибки;
- информацию об обработке сигналов;
- данные об использовании ресурсов и данные профиля процесса;
- время подачи сигналов тревоги;
- значения времени работы в режиме задачи и использования процессора;
- указатель на нить ядра;
- указатель на структуру proc.

Структура lwp может быть выгружена вместе с легковесным процессом, поэтому невыгружаемая по определению информация, например маски некоторых сигналов, хранится в структуре нити, связанной с LWP. В реализации системы под архитектуру Sparc для хранения указателя на текущую нить используется глобальный регистр %g7, что дает возможность быстрого доступа к текущему LWP и процессу.

Для легковесных процессов (и нитей ядра) доступны основные средства синхронизации, такие как взаимные исключения, условные переменные, семафоры и защелки чтения-записи. Эти средства будут более подробно описаны в разделе 7. Каждое из приведенных средств может определять различные варианты поведения нитей. Например, если нить попытается получить доступ к объекту mutex, удерживаемому другой нитью, то она либо перейдет в цикл активного ожидания освобождения объекта mutex, либо блокируется до момента этого события. Когда объект синхронизации инициализируется, источник вызова этого объекта должен указать, какое поведение для него ожидается.

Все LWP используют общий набор обработчиков сигналов. Однако каждый LWP может обладать собственной маской сигналов, решая самостоятельно, какие из полученных сигналов нужно игнорировать или блокировать. Любой LWP также имеет возможность определить свой собственный альтернативный стек для обработки сигналов. Все сигналы делятся на две категории: ловушки и прерывания. *Ловушки* представляют собой сигналы, вырабатываемые в ходе действий самого LWP (например, SIGSEGV, SIGFPE и SIGSYS). Такие сигналы всегда передаются легковесному процессу, действия которого при-

вели к их появлению. *Сигналы прерываний* (такие как SIGSTOP и SIGINT) могут быть доставлены любому LWP, который не маскирует эти сигналы.

Легковесные процессы не имеют глобального пространства имен и вследствие этого невидимы для других процессов. *Процесс не может направить сигнал напрямую определенному LWP, принадлежащему другому процессу, а также знать, какой LWP послал ему сообщение.*

3.6.3. Прикладные нити

Прикладные нити реализованы в системе при помощи *нитевой библиотеки.* Они могут создаваться, уничтожаться и обрабатываться без участия ядра. Библиотека также предоставляет средства синхронизации и планирования нитей. Это позволяет процессу использовать большое количество нитей, не потребляя при этом ресурсы ядра и не загружая систему лишними вызовами. Хотя в системе Solaris прикладные нити реализованы на основе LWP, нитевая библиотека скрывает эти детали, позволяя разработчикам приложений иметь дело только с прикладными нитями.

По умолчанию библиотека создает для каждого процесса набор LWP и мультиплексирует в него все прикладные нити. Размер такого набора зависит от количества процессоров и прикладных нитей. Разработчик приложения волен изменить начальные установки и самостоятельно указать количество создаваемых легковесных процессов. Он также вправе потребовать от системы назначения LWP какой-то определенной нити. Таким образом, процесс может обладать прикладными нитями двух типов: *свободными* (связанными с единственным LWP) нитями и *несвязанными*, разделяющими общий набор легковесных процессов (рис. 3.8).

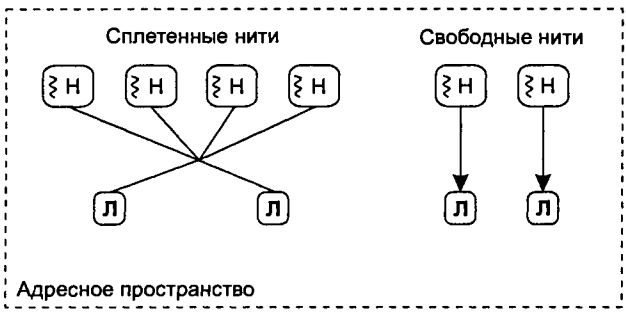


Рис. 3.8. Схема процесса в системе Solaris 2.x

Мультиплексирование большого количества нитей в небольшое число LWP дает возможность одновременной работы при достаточно низких затратах. Например, в оконной системе каждый объект (диалоговое окно, меню, пиктограмма и т. д.) может быть представлен в виде нити. В один момент времени обычно активно лишь некоторое количество окон, следовательно,

только эти нити должны поддерживаться LWP. Количество легковесных процессов определяет максимальную параллельность, которую может достигать приложение (как минимум равную количеству имеющихся процессоров). Оно также ограничивает количество одновременно имеющихся блокированных операций в процессе. В некоторых случаях превышение количества нитей над LWP является недостатком. Например, при вычислении произведения двух двухмерных массивов мы могли бы вычислять каждый элемент результирующего массива в отдельной нити. Если количество процессоров невелико, то этот метод, вероятно, окажется непродуктивным, так как библиотека может терять много времени на переключение контекста между нитями. Более эффективным решением может стать создание одной нити для каждой строки вычисляемого массива и привязка каждой нити к отдельному LWP.

Использование связанных и несвязанных прикладных нитей в одном приложении бывает весьма эффективным в тех случаях, когда требуется какая-либо обработка, время выполнения которой критично. Такая обработка может быть выполнена нитями, связанными с LWP, которым в расписании назначен приоритет реального времени. При этом другие нити отвечают за фоновые операции, имеющие более низкий приоритет. В приведенном ранее примере оконной системы нити реального времени могут использоваться для обработки движений мыши, так как их результаты должны немедленно появляться на экране монитора[1].

3.6.4. Реализация прикладных нитей

Каждая прикладная нить должна поддерживать информацию следующего содержания:

- **Идентификатор нити** (thread ID). Позволяет нитям взаимодействовать друг с другом в рамках процесса при помощи сигналов и прочих средств.
- **Сохраненное состояние регистров** (saved register state). Содержит указатель команд и указатель стека.
- **Стек в приложении** (user stack). Каждая нить обладает своим собственным стеком, размещаемым при помощи библиотеки. Ядро системы не знает о существовании подобных стеков.
- **Маска сигналов** (signal mask). Каждая нить может обладать собственной маской сигналов. Когда приходит сигнал, библиотека доведет его до соответствующей нити, исходя из информации, содержащейся в масках всех нитей.

---

[1] Неудачный пример. *Обработка* (не отображение на уровне драйвера) движений мыши в развитых графических оболочках (UNIX, Windows, OS/2 — для Intel 80x86) возможна только после выборки сообщений из общей очереди, что несмотря на все приоритеты не позволяет получить немедленные результаты. Можно наблюдать иллюстрацию сказанного при высокой загрузке процессора. Представлять каждый оконный объект в виде нити также нецелесообразно. — *Примеч. ред.*

- **Приоритет** (priority). Прикладная нить имеет приоритет внутри процесса, который используется планировщиком нитей. Ядро системы не обладает информацией о таких приоритетах и производит планирование выполнения только для LWP, содержащих эти нити.
- **Локальная область хранения нити** (thread kernel storage). Каждой нити позволено иметь некоторую собственную область для хранения данных (управляемую библиотекой) с целью поддержки реентерабельных версий интерфейсов библиотеки C [13]. Например, многие функции библиотеки C возвращают код ошибки в глобальную переменную errno. Если одну из таких функций вызовут одновременно несколько нитей, то это может привести к хаосу. Для предупреждения подобных проблем многонитевые библиотеки помещают значения errno в локальную область хранения нити [20].

Нити используют средства синхронизации, предоставляемые библиотекой, которые похожи на аналогичные средства ядра (условные переменные, семафоры и т. д.). Система Solaris позволяет нитям разных процессов синхронизироваться друг с другом при помощи переменных синхронизации, помещаемых в совместно используемый участок памяти. Такие переменные могут также быть размещены в файлах, а доступ к ним может быть организован при помощи механизма отображения файла mmap. Такой подход позволяет объектам синхронизации иметь время жизни большее, чем у создавшего их процесса, и, следовательно, задействовать эти переменные для синхронизации нитей разных процессов.

3.6.5. Обработка прерываний

Обработчики прерываний часто манипулируют данными, используемыми также и ядром системы. Это требует синхронизации доступа к разделяемым данным. В традиционных системах UNIX ядро обеспечивает ее, повышая *уровень приоритета прерываний* (ipl) для блокирования тех прерываний, которые в состоянии получить доступ к таким данным. Часто объекты защищаются от прерываний, хотя обращение к ним со стороны каких-либо прерываний малореально. Например, очередь спящих процессов должна быть защищена от прерываний, хотя большинству из них нет необходимости обращаться к этой очереди.

Эта модель обладает несколькими существенными недостатками. Во многих системах процедура увеличения или уменьшения уровня ipl является весьма затратной и требует выполнения нескольких инструкций. Прерывания представляют собой важные и обычно экстренные события, поэтому их блокирование уменьшает производительность системы в большинстве случаев. В многопроцессорных системах эта проблема еще актуальнее, так как ядру

системы приходится защищать намного большее количество объектов и обычно приходится блокировать прерывания на всех имеющихся процессорах.

В операционной системе Solaris традиционная модель прерываний и синхронизации заменена новой технологией, увеличивающей производительность, в первую очередь, на многопроцессорных системах [11], [17]. Новая модель не использует уровни ipl для защиты от прерываний. Вместо этого применяется набор различных объектов ядра для осуществления синхронизации, таких как взаимные исключения или семафоры. Для обработки прерываний используется набор нитей ядра. Такие *нити прерываний* могут быть созданы «на лету»; им будет назначен более высокий приоритет выполнения, чем у любых других существующих нитей. Нити прерываний используют те же основные элементы синхронизации, что и любые другие нити, и, следовательно, могут блокироваться в тех случаях, когда необходимый им ресурс занят другой нитью. Ядро блокирует обработку прерываний только при возникновении малого количества исключительных ситуаций, например при попытке освободить mutex, защищающий очередь спящих процессов.

Хотя создание нитей ядра является относительно несложной процедурой, однако организация новой нити для каждого прерывания слишком накладна. Ядро содержит набор предварительно выделенных и частично инициализированных нитей. По умолчанию такой набор состоит из количества нитей, равного количеству уровней прерываний помноженному на количество процессоров плюс еще одна общая системная нить для таймера. Поскольку каждая нить требует около 8 Кбайт для хранения стека и данных, весь набор занимает значительное количество памяти. На системах, обладающих небольшим объемом памяти, имеет смысл уменьшить количество нитей в наборе, так как ситуация, когда необходимо будет одновременно обрабатывать все прерывания, маловероятна.

На рис. 3.9 показана схема обработки прерываний в системе Solaris. Нить H1 выполняется на процессоре П1 в тот момент времени, когда он получает прерывание. Обработчик прерывания в первую очередь поднимает уровень ipl для предотвращения дальнейших прерываний того же или более низкого уровня (предохраняющая семантика системы UNIX). Далее происходит выделение нити прерывания H2 из набора нитей и переключение контекста на нее. Пока нить H2 выполняется, H1 остается *прикрепленной* (pinned). Это означает, что она не может выполняться на другом процессоре. После завершения H2 происходит обратное переключение контекста к нити H1, которая затем продолжает свою работу.

Нить прерываний H2 выполняется без полной инициализации. Это означает, что она не является полноценной нитью и не может быть вытеснена. Инициализация завершается только в том случае, если нить имеет причину для блокирования. Тогда она сохраняет свое состояние и становится независимой нитью, выполняющейся на любом свободном процессоре. Если нить H2 заблокируется, управление возвратится к H1, таким образом откреп-

ляя нить H1. В итоге перегрузка, вызываемая полной инициализацией нити, ограничивается случаями, когда нить прерывания должна заблокироваться.

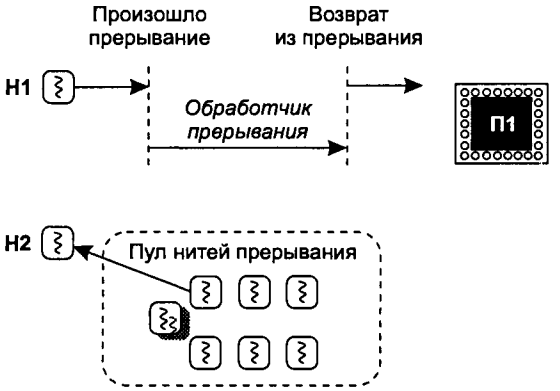


Рис. 3.9. Использование нитей для обработки прерываний

Реализация обработки прерываний через нити несколько увеличивает загруженность системы (на Sparc это порядка 40 инструкций). С другой стороны, такой подход избавляет от необходимости блокирования прерываний для каждого объекта синхронизации, что каждый раз экономит порядка 12 инструкций. Так как операции синхронизации являются более частыми, чем прерывания, общим результатом применения вышеописанного метода является увеличение производительности системы, поскольку прерывания блокируются не так часто.

3.6.6. Обработка системных вызовов

В системе Solaris вызов fork дублирует каждый LWP родителя для создаваемого потомка. Любые легковесные процессы, находящиеся в состоянии выполнения какого-либо системного вызова, вернутся с ошибкой EINTR. В ОС Solaris предлагается дополнительный системный вызов fork1, который похож на fork, но клонирует только ту нить, которая его вызвала. Вызов fork1 удобен в тех случаях, когда процесс-потомок вскоре собирается запустить новую программу.

Решение проблемы одновременных произвольных операций ввода-вывода над файлом сведено в Solaris 2.3 к добавлению системных вызовов pread и pwrite, которые принимают в качестве аргументов смещение в файле. К сожалению, система не предоставляет аналогичных вызовов, заменяющих readv и writev, которые осуществляют ввод-вывод методом сборки-разъединения (scatter-gather I/O, подробнее см. в разделе 8.2.5).

В заключение упомянем о том, что система Solaris предлагает богатый набор программных интерфейсов с ранее описанной двухуровневой моделью.

Поддержка прикладных нитей совместно с легковесными процессами дает возможность разделять то, что видит разработчик приложения, и то, как это представлено в системе. *Программист может создавать приложения, изначально используя только нити, и далее оптимизировать их путем манипуляции с содержащими эти нити легковесными процессами, добиваясь необходимой степени одновременности для данного приложения.*

3.7. Нити в системе Mach

Операционная система Mach изначально создавалась многонитевой. Mach поддерживает нити как на уровне ядра, так и при помощи библиотек прикладного уровня. Система предлагает дополнительные механизмы для управления функционированием нитей на разных процессорах многопроцессорных систем. ОС Mach поддерживает семантику 4.3BSD UNIX на уровне программного интерфейса полностью, включая все системные вызовы и библиотеки[1]. Этот раздел описывает реализацию нитей в системе Mach. Следующий раздел расскажет о системном интерфейсе Digital UNIX — системы, основанной на Mach. В разделе 3.9 вы увидите описание нового механизма под названием *продолжений* (continuations), впервые представленного в Mach 3.0.

3.7.1. Задачи и нити в системе Mach

Ядро Mach поддерживает два фундаментальных элемента системы: *задачу* и *нить* [22]. *Задача* — это статический объект, занимающий адресное пространство и набор системных ресурсов, называющихся *правами порта* (см. раздел 6.4.1). Сама по себе задача является не выполняемым объектом, а средой, в которой может выполняться одна или большее количество нитей.

Нить является основным выполняемым элементом, функционирующим в контексте задачи. Задача может содержать ноль и более нитей, каждая из которых разделяет ее ресурсы. Нить обладает стеком ядра, используемым для обработки системных вызовов. Она также имеет собственные переменные состояния, такие как указатели команд и стека, регистры общего назначения и т. д. Планирование нитей на выполнение процессором происходит независимо. Нити, относящиеся к прикладным задачам, эквивалентны легковесным процессам. Нити ядра, используемые ядром, относятся к *задачам ядра*.

Система Mach также поддерживает *наборы процессоров*, которые более подробно описаны в разделе 5.7.1. Все процессоры, доступные системе, могут быть поделены на не перекрывающие друг друга наборы. Каждая задача или нить может быть связана с любым набором процессоров (большинство опе-

[1] Система Mach версии 2.5 поддерживает функции 4.3BSD на уровне ядра. Версия 3.0 реализует эти функции в виде серверной программы на прикладном уровне.

раций над наборами процессоров требуют привилегий суперпользователя). Такой подход позволяет назначить несколько процессоров многопроцессорной системы для выполнения одной или более определенных задач. Это гарантирует доступность ресурсов для наиболее приоритетных задач.

Структура task описывает *задачу* и хранит следующую информацию:

- ✦ указатель на карту адресации, описывающую виртуальное адресное пространство задачи;
- ✦ заголовок списка нитей, относящихся к задаче;
- ✦ указатель на набор процессоров, с которым связана задача;
- ✦ указатель на структуру utask (см. раздел 3.8.1);
- ✦ порты и другую информацию, относящуюся к межпроцессному взаимодействию (IPC, см. подробнее в разделе 6.4).

Ресурсы, удерживаемые задачей, совместно используются всеми ее нитями. Каждая нить описана структурой thread, содержащей:

- ✦ связь нити с очередью планировщика или очередью ожидания;
- ✦ указатели на task и на набор процессоров, к которому относится нить;
- ✦ связь нити со списком всех нитей задачи и со списком всех нитей набора процессоров;
- ✦ указатель на *блок управления процессом* (PCB), содержащий сохраненный контекст регистров;
- ✦ указатель на стек ядра;
- ✦ состояние выполнения (готовый к работе, спящий, блокированный и т. д.);
- ✦ информацию для планировщика, такую как приоритет, правила планирования и данные по использованию процессора;
- ✦ указатели на связанные с нитью структуры uthread и utask (см. раздел 3.8.1);
- ✦ информацию об IPC, относящуюся к нити (см. подробнее в разделе 6.4.1).

Задачи и нити играют дополняющие друг друга роли. Задача владеет ресурсами системы, включая адресное пространство. Нить выполняет код. Процесс в традиционной системе UNIX представляет собой задачу, содержащую единственную нить. Многонитевый процесс состоит из одной задачи и некоторого количества нитей.

Система Mach предлагает набор системных вызовов для работы с задачами и нитями. Вызовы task_create, tast_terminate, task_suspend и task_resume используются для работы с задачами. Вызовы thread_create, thread_terminate, thread_suspend и thread_resume выполняют операции над нитями. Название каждого вызова красноречиво самодокументирует выполняемые им действия. Вдобавок к перечисленным вызовам имеются вызовы thread_status и thread_mutate для чтения и изменения состояния регистров нити, а вызов task_threads возвращает список всех нитей задачи.

3.7.2. Библиотека C-threads

Система Mach содержит библиотеку C-threads, которая обеспечивает простой интерфейс для создания и управления нитями. Например, функция

```
cthread_t cthread_fork (void* (*func)(), void* arg);
```

создает новую нить, в которой будет запущена функция func(). Нить может вызвать

```
void* cthread_join (cthread_t T);
```

для приостановки своей работы до тех пор, пока не закончит функционирование нить T. В вызывающий код нити вернется значение функции верхнего уровня нити T или код выхода, выработанный после выполнения нитью T функции cthread_exit().

Библиотека C-threads поддерживает взаимные исключения и условные переменные с целью осуществления синхронизации. В библиотеке имеется функция cthread_yield(), которая запрашивает планировщик на разрешение выполнения другой нити вместо себя. Такая функция необходима только в случаях применения сопрограмм, описанных ниже.

Существуют три различных реализации библиотеки C-threads. Разработчик приложения может использовать вариант, наиболее отвечающий перечисленным ниже требованиям к разрабатываемому продукту.

- **Приложение основано на сопрограммах** (coroutine-based), где прикладные нити мультиплексируются в однонитевую задачу (процесс UNIX). Такие нити являются не вытесняемыми, библиотека выполнит переключение в другую нить только в случае выполнения процедур синхронизации (когда текущая нить должна быть блокирована на взаимном исключении или семафоре). Помимо этого библиотека полагается на нити, вызывающие функцию cthread_yield(), что защитит другие нити от долгого «стояния» в очереди. Эта реализация библиотеки подходит для отладки, так как порядок переключения контекста нитей является повторяемым.
- **Приложение основано на нитях** (thread-based), каждая нить C-thread использует отдельную нить, поддерживаемую ядром Mach. Такие нити являются вытесняемыми и могут выполняться на многопроцессорных системах параллельно. Эта реализация является стандартной и используется для разработки различных вариантов программ, базирующихся на нитях C-thread.
- **Приложение основано на задачах** (task-based), где используется одна задача, поддерживаемая ядром Mach (процесс UNIX) на каждую нить C-thread. Для совместного использования памяти между нитями применяются элементы виртуальной памяти, обеспечиваемые ядром Mach. Такой вариант используется только в случаях, когда необходима специализированная семантика разделения памяти.

3.8. Digital UNIX

Операционная система Digital UNIX (ранее известная как DEC OSF/1) основана на ядре Mach 2.5. С точки зрения разработчика приложений эта система предоставляет полный программный интерфейс UNIX. Однако внутренняя реализация многих возможностей традиционного варианта UNIX в этой системе опирается на базовые элементы ОС Mach. Эта реализация организована на уровне совместимости системы Mach с ОС 4.3BSD, расширенного фондом Open Software Foundation до совместимости с SVR3 и SVR4. Это свойство существенно повлияло на устройство системы Digital UNIX.

Система предлагает изящный набор средств, расширяющих понятие процесса [19]. Многонитевые процессы поддерживаются как на уровне ядра, так и на уровне совместимых со стандартами POSIX нитевых библиотек. Процесс UNIX реализован как верхний уровень задач и нитей системы Mach.

3.8.1. Интерфейс UNIX

Хотя задачи и нити достаточно адекватно обеспечивают интерфейс выполнения программ системы Mach, они не в полной мере описывают процесс UNIX. Процесс обеспечивает некоторые свойства, которые не отражены в Mach, такие как полномочия пользователя, дескрипторы открытых файлов, обработчики сигналов и группы процессов. Более того, для предотвращения изменения традиционного интерфейса UNIX был осуществлен перенос кода уровня, обеспечивающего совместимость Mach 2.5 с 4.3BSD, который, в свою очередь, был перенесен из оригинальной реализации 4.3BSD. Точно так же был произведен перенос многих драйверов устройств из системы Digital ULTRIX, также основанной на ОС BSD. Перенесенный код делает множественные ссылки на структуры proc и user также для обеспечения совместимости.

Применение оригинального варианта структур proc и user является причиной возникновения двух проблем. Во-первых, некоторая информация из этих структур уже отражена в структурах task и thread. Во-вторых, они не могут адекватно представлять многонитевые процессы. Например, традиционная область u содержит *блок управления процессом*, который хранит контекст регистров процесса. В случае многонитевости каждая нить обладает собственным контекстом регистров. Следовательно, обе структуры должны быть существенно изменены.

Область u заменена двумя объектами: единой структурой utask, которая используется задачей целиком, и по одной структуре uthread выделено для каждой нити задачи. Новые структуры не занимают фиксированное адресное пространство процесса и не участвуют в его свопинге.

Структура utask содержит следующую информацию:

- ✦ указатели на объекты vnode текущего и корневого каталогов;
- ✦ указатель на структуру proc;

- массив обработчиков сигналов и других полей, относящихся к сигналам;
- таблицу дескрипторов открытых файлов;
- маску создания файлов, используемую по умолчанию (cmask);
- данные об использовании ресурсов, квотах и информацию профиля.

Если одна из нитей открывает файл, то его дескриптор может быть использован совместно всеми нитями задачи. Также все нити будут иметь один и тот же текущий рабочий каталог. Структура uthread описывает ресурсы, относящиеся к каждой нити процесса UNIX, и содержит следующую информацию:

- указатель на сохраненные регистры прикладного уровня;
- поля для просматриваемых путей;
- текущие и ожидающие сигналы;
- обработчики сигналов, определенные для данной нити.

Для упрощения переноса ссылки на поля старой области u были преобразованы в ссылки на поля структуры utask или uthread. Такое преобразование можно осуществить при помощи макроса, например:

```
#define u_cmask      utask->uu_cmask
#define u_pcb        uthread->uu_pcb
```

Структура proc претерпела незначительные изменения, но большинство ее функций теперь возложено на структуры task и thread. В результате большинство ее полей не используется, хотя они и сохранены из «исторических» соображений. Например, поля, относящиеся к расписанию и приоритету, не являются необходимыми, так как в ОС Digital UNIX каждая нить планируется на выполнение индивидуально. Структура proc операционной системы Digital UNIX содержит следующую информацию:

- связи с очередью размещенных процессов, процессов-зомби или свободных процессов;
- маски сигналов;
- указатель на структуру полномочий процесса;
- информацию об идентификации и иерархии процесса (PID процесса, PID предка, указатели на процесс-предок, процессы-потомки, процессы того же уровня и т. д.);
- группу процесса и информацию сеанса;
- поля, относящиеся к расписанию (не используются);
- поля для хранения состояния и использования ресурсов при выходе;
- указатели на структуры task и utask, а также на первую в списке структуру thread.

На рис. 3.10 показана связь между структурами данных в системе Mach и традиционной UNIX. Структура task содержит связанный список своих нитей. Структура task указывает на utask, а каждая структура thread указывает на соответствующую структуру uthread. Структура proc содержит указатели на структуры task, utask и ссылается на первую в списке структуру thread. Структура utask содержит обратный указатель на proc, а каждая thread включает в себя обратные указатели на task и utask. Такая организация взаимосвязей дает возможность быстрого доступа ко всем структурам.

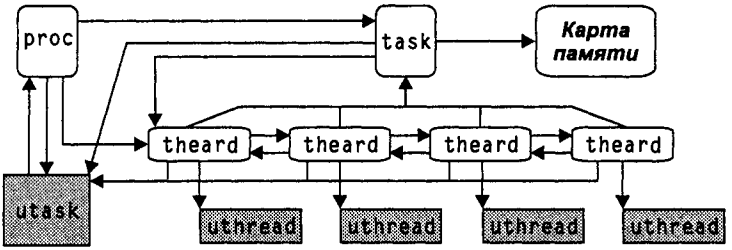


Рис. 3.10. Структуры данных задач и нитей в системе Digital UNIX

Не всем нитям присваивается прикладной контекст. Некоторые могут создаваться непосредственно ядром для выполнения различных системных функций, таких как замена страниц памяти. Такие нити связаны с задачей ядра, которая не имеет прикладного адресного пространства. Задача ядра и нити ядра не имеют связанных с ними структур utask, uthread и proc.

3.8.2. Системные вызовы и сигналы

В операционной системе Digital UNIX системный вызов fork создает новый процесс, обладающий единственной нитью, которая является точной копией нити, вызвавшей fork. Система не поддерживает альтернативного варианта вызова, дублирующего все нити.

Так же как и в ОС Solaris, все сигналы классифицируются на синхронные (системные прерывания, или ловушки, — *traps*) или асинхронные сигналы (прерывания, *interrupts*). Ловушка доставляется той нити, которая стала причиной ее срабатывания. Сигналы прерываний доставляются любой нити, которая примет их. Однако, в отличие от операционной системы Solaris, все нити процесса используют единый набор масок сигналов, который хранится в структуре proc. Каждая нить может создавать собственный набор обработчиков синхронных сигналов, но все нити процесса должны пользоваться единым набором обработчиков асинхронных сигналов.

3.8.3. Библиотека pthreads

Библиотека pthreads предлагает совместимый с POSIX программный интерфейс прикладного уровня для реализации нитей, являющийся более простым, чем

системные вызовы системы Mach. С каждой нитью pthread библиотека связывает одну нить Mach. Функционально pthreads подобны C-threads или иной нитевой библиотеке, но именно в pthreads реализован интерфейс, принятый как стандарт.

Библиотека pthreads реализует функции асинхронного ввода-вывода, определенные стандартом POSIX. Например, если нить вызывает определенную в POSIX функцию aioread(), то библиотека создаст новую нить для синхронного чтения. Когда чтение завершится, ядро разбудит заблокированную на этой операции нить, которая, в свою очередь, уведомит вызывающую ее нить при помощи сигнала. Асинхронный ввод-вывод проиллюстрирован на рис. 3.11.

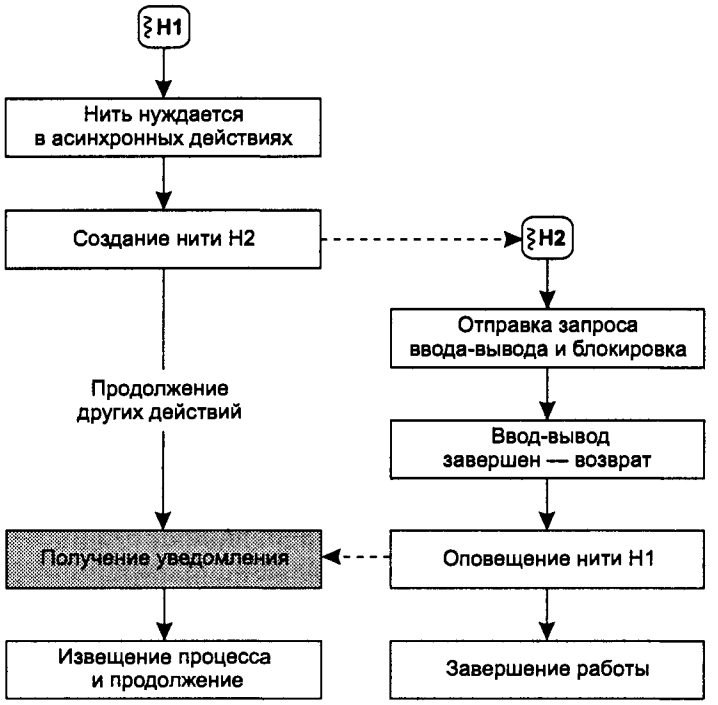


Рис. 3.11. Реализация асинхронного ввода-вывода при помощи создания отдельной нити

Библиотека pthreads предоставляет полный программный интерфейс, включающий функции обработки сигналов и планирования, а также набор элементов синхронизации. Синхронизация между нитями может быть реализована на прикладном уровне, однако если требуется заблокировать LWP, должно вмешаться ядро системы.

Digital предлагает свою собственную библиотеку, cma_threads, поддерживающую некоторые дополнительные возможности [8]. Программы, использующие эту библиотеку, будут работать только на платформах Digital VMS и Windows, на других UNIX-системах они функционировать не будут.

3.9. Продолжения в системе Mach

Хотя нити ядра более легковесны, чем процессы, они все равно занимают больший объем памяти ядра, используя ее преимущественно для стека. Обычно стеки ядра занимают как минимум 4 Кбайт памяти, что составляет почти 90% пространства ядра, используемого нитью. В системах с большим количеством нитей (достигающим нескольких сотен) это может привести к уменьшению производительности. Одним из решений проблемы является мультиплексирование прикладных нитей в меньшее количество нитей Mach или легковесных процессов, что предотвратит потребность в стеке ядра для каждой прикладной нити. Такой подход имеет свои минусы, так как прикладные нити не могут планироваться на выполнение независимо и, следовательно, не дают такого же уровня одновременности, как в случае соответствия каждой прикладной нити одной нити ядра. Более того, раз нити ядра не переходят границы задачи, соответственно каждая задача должна обладать по крайней мере одной нитью ядра, что создает проблемы на системах с большим количеством активных задач. В этом разделе описан подход к решению вышеописанных проблем, предлагаемый системой Mach 3.0, при помощи средства, получившего название *продолжений* (continuations).

3.9.1. Модели выполнения программ

Ядро системы UNIX использует модель выполнения программ, называемую *моделью процессов*. Каждая нить имеет стек ядра, востребуемый нитью, когда она переходит в режим ядра для выполнения системного вызова или обработки исключения. Если нить блокируется в ядре, ее стек содержит состояние ее выполнения, в том числе последовательность вызова[1] и автоматические переменные. Основным достоинством описанного подхода является его простота, поскольку нити ядра могут блокироваться без необходимости явного сохранения каких-либо состояний. Главный недостаток модели процессов — это большой расход памяти.

Некоторые операционные системы, такие как QuickSilver [14] и V [6], используют иную программную модель под названием *модели прерываний*. Ядро обрабатывает системные вызовы и исключения как прерывания, организуя для всех операций ядра единый стек (один для каждого процессора). Следовательно, если нити необходимо заблокироваться, находясь в ядре, ей в первую очередь нужно где-то сохранить свое состояние. Ядро использует сохраненную информацию для восстановления состояния нити при следующем ее запуске.

Основным достоинством модели прерываний является экономия памяти, достигаемая за счет использования единого стека ядра. Главным недостат-

[1] То есть когда одна функция вызывает другую; она, в свою очередь, — третью, и так далее, и на каком-то шаге происходит блокирование. — *Примеч. ред.*

ком является необходимость сохранения состояния нити при проведении любой операции, потенциально могущей привести к ее блокированию. Это осложняет применение модели, так как сохраняемая информация может пересечь границы модуля. Следовательно, если нить блокируется, находясь в глубоко вложенной процедуре, ей необходимо определить (для сохранения) состояние, необходимое для всех вызовов в последовательности.

Условия, при которых нить должна быть блокирована, определяются той моделью, которая является наиболее пригодной. Наиболее подходящим вариантом является задание блокируемой нитью условий, при которых необходимо использовать ту или иную модель. Если нить блокируется где-то глубоко внутри последовательности вызовов, ей больше подойдет модель процессов. Однако если нити требуется сохранить при блокировании информацию о состоянии небольшого размера, то в этом случае модель прерываний окажется более предпочтительной. К примеру, многие серверные программы периодически блокируются в ядре, ожидая запроса клиента, и затем обрабатывают запросы по мере их получения. Такая программа может легко освобождать свой стек.

Механизм продолжений системы Mach комбинирует преимущества обеих моделей и позволяет ядру выбрать метод блокирования в зависимости от условий. Следующий раздел книги посвящен описанию устройства и реализации этого средства.

3.9.2. Использование продолжений

Для блокирования нити система Mach использует функцию thread_block(). В Mach версии 3.0 эта функция была изменена и теперь обладает аргументом:

```
thread_block (void (*contfn)());
```

где contfn() — это *функция продолжения*, которая будет запущена при следующем выполнении нити. Передача функции аргумента NULL указывает на необходимость традиционного поведения при блокировании. При таком подходе нить может выбирать, какое продолжение задействовать далее.

Если нить собирается использовать продолжение, то, в первую очередь, она нуждается в сохранении того состояния, которое будет ей необходимо при возобновлении выполнения. Структура нити содержит для этой цели 28-байтовую временную область. Если потребуется больший объем пространства, то нить выделит дополнительную структуру данных. Ядро блокирует нити и забирает ее стек. После возобновления функционирования нити ядро дает ей новый стек и вызывает функцию продолжения. Эта функция восстанавливает состояние из сохраненной области. Такой подход требует, чтобы продолжение и вызываемая функция имели точное представление о том, какое состояние сохранено и где.

Использование продолжений показано на следующем примере. Листинг 3.1 демонстрирует традиционный подход к блокированию нити.

Листинг 3.1. Блокировка нити без использования продолжений

```
syscall_1 (arg1)
{
      ...
      thread_block();
      f2(arg1);
      return;
}
f2(arg1)
{
      ...
      return;
}
```

Листинг 3.2 иллюстрирует применение продолжений.

Листинг 3.2. Блокировка нити с использованием продолжений

```
syscall_1 (arg1)
{
      ...
      сохранение arg1 и другой информации о состоянии
      ...
      thread_block(f2);
      /* сюда выполнение не доходит */
}
f2()
{
      ...
      восстановление arg1 и другой информации о состоянии
      ...
      thread_syscall_return (status);
}
```

Следует упомянуть о том, что в случае вызова функции thread_block() с аргументом не произойдет возврата в вызвавший ее код. После восстановления функционирования нити ядро передает управление f2(). Функция thread_syscall_return() используется для возврата на прикладной уровень из системного вызова. Весь этот процесс является прозрачным для разработчика, так как он видит лишь синхронный выход из системного вызова.

Ядро использует продолжения при условии, что сохраненное перед блокированием состояние будет небольшим. Например, одна из наиболее частых блокирующих операций происходит при ошибке обработки страницы. В традиционных реализациях UNIX код обработчика вызывается вследствие запроса чтения с диска и блокирует работу до тех пор, пока чтение не закончится. Затем ядро системы возвращает нить на прикладной уровень, после чего приложение может дальше продолжать функционирование. Работа, которая должна быть выполнена после окончания операции чтения с диска,

требует небольшого сохраненного состояния (например, указателя на считанную страницу и данные отображения памяти, которые должны быть обновлены). Этот пример показывает ситуацию, когда применение продолжений оправданно.

3.9.3. Оптимизация работы

Главным достоинством продолжений можно назвать сокращение количества стеков в ядре системы. Продолжения также позволяют провести ряд важных оптимизаций. Представьте, что при переключении контекста ядро обнаружило, что предыдущая и последующая нити используют продолжения. Предыдущая нить уже освободила свой стек ядра, а следующая нить еще не имеет такового. В этом случае ядро может передать стек от старой нити новой напрямую, как это продемонстрировано на рис. 3.12. Помимо исключения перегрузок, которые бы происходили при выделении нового стека, такой подход помогает сократить кэш-промахи[1] и *буферы ассоциативной трансляции* (translation lookaside buffer, TLB, см. подробнее в разделе 13.3.1), ассоциированные с переключением контекста, так как используется та же область памяти.

Преимущества продолжений используются также в реализации IPC (межпроцессное взаимодействие) системы Mach. Передача сообщения включает две стадии. Клиентская нить использует для отправки сообщения и ожидания ответа системный вызов `mach_msg`, а серверная нить использует тот же вызов для отправки ответа клиентам и ожидания новых запросов. Сообщение отправляется в порт, а также принимается из порта, являющегося защищенной очередью сообщений. Отправка и получение сообщений осуществляются независимо друг от друга. Если получатель не готов, ядро поместит сообщение в очередь порта.

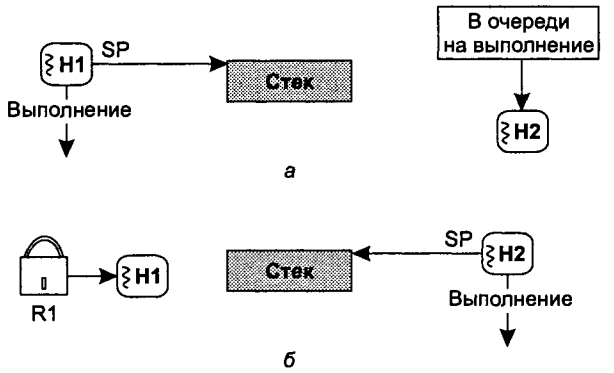


Рис. 3.12. Передача стека при использовании продолжений: *а* — перед блокировкой нити H1 с функцией продолжения; *б* — после контекстного переключения

[1] То есть когда требуемые данные в кэше отсутствуют. — *Прим. ред.*

Если получатель находится в режиме ожидания, то процедура пересылки может быть оптимизирована при помощи продолжений. Когда отправитель обнаружит, что получатель находится в режиме ожидания, он передаст свой стек получателю и заблокируется на функции продолжения mach_msg_continue(). Получающая нить восстановит свою работу, используя при этом стек отправителя, который уже содержит всю необходимую информацию о передаваемом сообщении. Такой подход предотвращает перегрузку, возникающую при помещении в очередь и извлечении из нее сообщения, а также ощутимо увеличивает скорость обмена сообщениями. После ответа сервера происходит передача его стека клиентской нити и возобновление работы клиента описанным выше способом.

3.9.4. Анализ производительности

Механизм продолжений системы Mach показал себя очень эффективным. Так как его применение не является обязательным, нет необходимости менять программную модель целиком, и его использование может быть наращиваемым. Механизм продолжений очень сильно сокращает количество запросов, размещаемых в памяти ядра. Измерения производительности показали [10], что в среднем в системе, которой требуется 2002 байт стека в ядре на каждый процессор, пространство ядра для каждой нити сокращается с 4664 до 690 байт.

Операционная система Mach 3.0 очень неплохо подходит для продолжений, так как обладает микроядром, имеющим небольшое количество базовых элементов и предлагающим скромный интерфейс. В частности, та часть кода, которая сохраняла совместимость системы с UNIX, была удалена из ядра, и ее реализация состоялась в виде серверов прикладного уровня [12]. В результате оказалось только 60 потенциальных мест, в которых ядро может блокировать выполнение, а 99% всех случаев блокирования происходит в шести «горячих точках». При концентрировании внимания на них обеспечивается определенное преимущество, заключающееся в уменьшении усилий, затрачиваемых на разработку приложений. В противоположность Mach, традиционные системы UNIX могут производить блокирование в сотнях мест, при этом не имея ни одной так называемой «горячей точки».

3.10. Заключение

В этой главе вы увидели несколько вариантов устройства многонитевых систем. Существует большое количество типов средств, относящихся к нитям, а комбинирование их системой дает возможность создания сложной с элементами одновременности программной среды. Нити могут поддерживаться

как на уровне ядра, так и на уровне прикладных библиотек, допустима и комбинация обоих вариантов.

Разработчикам приложений предложен выбор приемлемого соотношения прикладных средств и возможностей, предоставляемых ядром системы. Единственная проблема, с которой им придется столкнуться, это различия в наборах системных вызовов для создания и управления нитями, предлагаемые в различных операционных системах, что затрудняет написание переносимого многонитевого кода, который бы эффективно использовал системные ресурсы. Стандарт POSIX 1003.4a определяет функции нитевой библиотеки, однако в нем не описываются интерфейсы или реализации для ядра.

3.11. Упражнения

1. Для каждого из перечисленных ниже приложений проанализируйте пригодность использования легковесных процессов, прикладных нитей или иных программных моделей:
   - серверный компонент распределенной службы имен;
   - оконная система, такая как X server;
   - научное приложение, работающее на многопроцессорной системе и производящее большое количество параллельных вычислений;
   - утилита make, которая компилирует файлы параллельно по мере возможности.
2. В каких ситуациях применение нескольких процессов в приложении будет более оправдано, нежели использование LWP или прикладных нитей?
3. Почему каждому LWP необходим отдельный стек ядра? Может ли система экономить ресурсы, создавая стек ядра только в тех случаях, когда LWP делает системный вызов?
4. Структура proc и область u содержат атрибуты и ресурсы процесса. Какие из полей этих структур в многонитевых системах можно разделить между всеми LWP процесса, а какие из них должны существовать для каждого LWP отдельно?
5. Представьте, что LWP вызывает fork в тот момент времени, когда другой LWP, принадлежащий тому же процессу, вызывает exit. Что произойдет в результате, если система использует fork для дублирования всех LWP процесса? Что случится, если вызов fork будет дублировать только один (вызывающий fork) LWP?
6. Возникнут ли какие-нибудь проблемы с вызовом fork в многонитевой системе, поддерживающей его как единый вызов, атомарно совмещающий fork и exec?

7. Раздел 3.3.2 описывает проблемы, возникающие при использовании единого разделяемого набора ресурсов, таких как дескрипторы файлов или текущий каталог. Почему не следует эти ресурсы предоставлять отдельно для каждого легковесного процесса или прикладной нити? Более подробно поднятая проблема исследуется в [3].
8. Стандартная библиотека определяет для каждого процесса переменную errno, содержащую код ошибки, возвращенный последним сделанным системным вызовом. Какие проблемы это может создать в многонитевом процессе? Каким образом их можно преодолеть?
9. Во многих системах библиотечные функции подразделяются на нитезащищенные и нитенезащищенные. К чему может привести использование функции, которая не защищена при использовании в многонитевых приложениях?
10. Перечислите недостатки использования нитей для запуска в них обработчиков прерываний.
11. Какие вы видите недостатки в планировании выполнения LWP посредством ядра?
12. Предложите интерфейс, который позволил бы управлять планированием и выбирать, какой из LWP первым отправится на выполнение. К возникновению каких проблем это может привести?
13. Сравните базовые многонитевые элементы в системах Solaris и Digital UNIX. Какие преимущества имеются у каждого из них?

3.12. Дополнительная литература

1. Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M., «Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism», Proceedings of the Thirteenth Symposium on Operating System Principles, Oct. 1991, pp. 95–109.
2. Armand, F., Hermann, F., Lipkis, J., and Rozier, M., «Multi-threaded Processes in Chorus/MIX», Proceedings of the Spring 1990 European UNIX Users Group Conference, Apr. 1990.
3. Barton, J. M., and Wagner, J. C., «Beyond Threads: Resource Sharing in UNIX», Proceedings of the Winter 1988 USENIX Technical Conference, Jan. 1988, pp. 259–266.
4. Bitar, N., «Selected Topics in Multiprocessing», USENIX 1995 Technical Conference Tutorial Notes, Jan. 1995.
5. Black, D. L., «Scheduling Support for Concurrency and Parallelism in the Mach Operating System», IEEE Computer, May 1990, pp. 35–43.

6. Cheriton, D. R., «The V Distributed System», Communications of the ACM, Vol. 31, No. 3, Mar. 1988, pp. 314–333.
7. Cooper, E. C., and Draves, R. P., «C Threads», Technical Report CMU–CS–88–154, Department of Computer Science, Carnegie Mellon University, Sep. 1990.
8. Digital Equipment Corporation, «DEC OSF/1 — Guide to DECthreads», Part No. AA–Q2DPB–TK, July 1994.
9. Doeppner, T. W., Jr., «Threads, A System for the Support of Concurrent Programming», Brown University Technical Report CS–87–11, Jun. 1987.
10. Draves, R. P., Bershad, B. N,, Rashid, R. F., and Dean, R. W., «Using Continuations to Implement Thread Management and Communication in Operating Systems», Technical Report CMU–CS–9–115R, Department of Computer Science, Carnegie Mellon University, Oct. 1991.
11. Eykholt, J. R., Kleiman, S. R., Barton, S., Faulkner, R., Shivalingiah, A., Smith, M., Stein, D., Voll, J., Weeks, M., and Williams, D., «Beyond Multiprocessing: Multithreading the SunOS Kernel», Proceedings of the Summer 1992 USENIX Technical Conference, Jun. 1992, pp. 11–18.
12. Golub, D., Dean, R., Forin, A., and Rashid, R., «UNIX as an Application Program», Proceedings of the Summer 1990 USENIX Technical Conference, Jun. 1990, pp. 87–95.
13. Institute for Electrical and Electronic Engineers, «POS1X PI003.4a, Threads Extension for Portable Operating Systems», 1994.
14. Haskin, R., Malachi, Y., Sawdon, W., and Chan, G., «Recovery Management in Quicksilver», ACM Transactions on Computer Systems, Vol. 6, No. 1, Feb. 1988, pp. 82–108.
15. Kepecs, J., «Lightweight Processes for UNIX Implementation and Applications», Proceedings of the Summer 1985 USENIX Technical Conference, Jun. 1985, pp. 299–308.
16. Keppel, D., «Register Windows and User-Space Threads on the SPARC», Technical Report 91–08–01, Department of Computer Science and Engineering, University of Washington, Seattle, WA, Aug. 1991.
17. Kleiman, S. R., and Eykholt, J. R., «Interrupts as Threads», Operating Systems Review, Vol. 29, No. 2, Apr. 1995.
18. Mueller, F., «A Library Implementation of POSIX Threads under UNIX», Proceedings of the Winter 1993 USENIX Technical Conference, Jan. 1993, pp. 29–41.
19. Open Software Foundation, «Design of the OSF/1 Operating System — Release 1.2», Prentice-Hall, Englewood Cliffs, NJ, 1993.

20. Powell, M. L., Kleiman, S. R., Barton, S., Shah, D., Stein, D., and Weeks, M., «SunOS Multi-thread Architecture», Proceedings of the Winter 1991 USENIX Technical Conference, Jan. 1991, pp. 65–80.
21. Sun Microsystems, «SunOS 5.3 System Services», Nov. 1993.
22. Tevanian, A., Jr., Rashid, R. F., Golub, D. B., Black, D. L., Cooper, E., and Young, M. W., «Mach Threads and the UNIX Kernel: The Battle for Control», Proceedings of the Summer 1987 USENIX Technical Conference, Jun. 1987, pp. 185–197.
23. Vandevoorde, M., and Roberts, E., «WorkCrews: An Abstraction for Controlling Parallelism», International Journal of Parallel Programming, Vol. 17, No. 4, Aug. 1988, pp. 347–366.

Глава 4
Сигналы и управление сеансами

4.1. Введение

Сигналы используются для оповещения процесса о возникновении системных событий. Еще одной функцией сигналов является простой механизм, используемый для коммуникаций и синхронизации между прикладными процессами. Программный интерфейс, поведение, а также внутренняя реализация сигналов сильно разнятся от одной версии UNIX к другой, а иногда и в различных версиях одной и той же операционной системы. Словно для того, чтобы еще более запутать разработчика, операционная система предоставляет дополнительные системные вызовы и библиотечные функции для поддержки ранних интерфейсов сигналов, а также для обеспечения обратной (backward) совместимости[1].

Оригинальная реализация сигналов в ОС System V была изначально неэффективной и ненадежной. Многие ее проблемы были решены после появления системы 4.2BSD UNIX, в которой был предложен новый, надежный механизм сигналов (расширенный в следующей версии, 4.3BSD). Однако механизм системы 4.2BSD явился несовместимым с интерфейсом System V по некоторым аспектам, что послужило основой для возникновения определенных проблем как у разработчиков приложений, желающих создавать переносимые программы, так и у поставщиков операционных систем, стремившихся к совместимости своего продукта одновременно с BSD и System V.

Стандарт POSIX 1003.1 (также известный как POSIX.1, [5]) дал возможность наведения некоторого порядка в хаосе различных реализаций сигналов. Он определил стандартный интерфейс, который должны поддерживать все совместимые с POSIX операционные системы. Однако стандарт не опи-

[1] Такой подход приводит к возникновению и других проблем. Если библиотека, связанная с приложением, использует один набор сигнальных интерфейсов, а приложение — другой, результатом может оказаться некорректное функционирование программы.

сывает, каким именно образом этот интерфейс должен быть реализован. Разработчики операционных систем могут решать сами, на каком уровне они будут поддерживать рекомендации стандарта: в ядре, или через прикладные библиотеки, или через комбинацию обеих составляющих.

Разработчики ОС SVR4 ввели в систему новую, POSIX-совместимую реализацию сигналов, включающую в себя многие возможности механизма сигналов системы BSD. Современные варианты UNIX (такие как Solaris, AIX, HP-UX, 4.4BSD и Digital UNIX) также предлагают совместимые с POSIX решения. Реализация сигналов в 3SVR4 помимо совместимости со стандартом сохранила совместимость с более ранними версиями System V.

Эта глава начинается с объяснения, что такое сигналы, и последующего анализа проблем, имеющихся в оригинальной ОС System V. Затем вы увидите, как эти проблемы были решены в современных операционных системах, располагающих механизмом надежных сигналов. В конце главы мы расскажем об управлении заданиями и сеансами — понятиями, имеющими тесную связь с сигналами.

4.2. Генерирование и обработка сигналов

Сигналы дают возможность вызвать какую-либо процедуру при возникновении события из определенного их набора. События обозначаются целыми числами и представляются символьными константами. Некоторые из событий являются асинхронными уведомлениями (возникающими, например, когда пользователь посылает сигнал прерывания, нажав комбинацию Ctrl+C на терминале), в то время как другие представляют собой синхронные ошибки или исключения (например, при попытке обращения по несуществующему адресу).

Процесс оповещения состоит из двух этапов: генерирования и доставки. Сигнал генерируется после того, как возникает определенное событие, требующее уведомления о нем процесса, который явился виновником его появления. Сигнал считается доставленным (или обработанным), когда получивший его процесс определяет факт доставки сигнала и производит необходимые действия. Между этими двумя событиями сигнал находится в режиме ожидания процесса.

В оригинальном варианте ОС System V определено 15 различных сигналов. Системы 4BSD и SVR4 поддерживают по 31 сигналу. Каждому из них присваивается номер от 1 до 31 (установка номера сигнала в 0 для различных функций имеет специальные значения, например «никаких сигналов»). Адресация сигналов по их номерам отличается в системах System V и BSD UNIX (например, SIGSTOP имеет номер 17 в 4.3BSD и номер 23 в SVR4). Более

того, многие коммерческие реализации UNIX (такие как AIX) поддерживают больше чем 31 сигнал. Обычно программисты предпочитают использовать символические имена для идентификации сигналов. Стандарт POSIX 1003.1 определяет символические имена для всех поддерживаемых им сигналов. Эти имена являются переносимыми как минимум для всех реализаций систем, совместимых со стандартом POSIX.

4.2.1. Обработка сигналов

Каждый сигнал обладает некоторым *действием, установленным по умолчанию*, которое производится ядром системы, если процесс не имеет определенного альтернативного обработчика. Всего таких действий пять.

- **Аварийное завершение** (abort). Завершает процесс после создания *дампа состояния процесса* (core dump), представляющего собой содержимое адресного пространства процесса и его контекст регистров, записанные в файл core, расположенный в текущем каталоге процесса[1]. Создаваемый файл может быть подвергнут дальнейшему анализу при помощи отладчика или других утилит.
- **Выход** (exit). Завершает процесс без создания дампа состояния процесса.
- **Игнорирование** (ignore). Игнорирует сигнал.
- **Остановка** (stop). Приостанавливает процесс.
- **Продолжение** (continue). Возобновляет работу приостановленного процесса (или игнорируется в ином случае).

Процесс может переопределить действия, производимые по умолчанию для любого сигнала. Таким альтернативным вариантом может быть игнорирование сигнала или запуск определенной в приложении функции, называемой *обработчиком сигнала*. Процесс может в любое время указать новое действие либо, наоборот, сбросить установки на действия по умолчанию. Процесс вправе временно блокировать сигнал (не поддерживается в SVR2 и более ранних версиях этой системы). В таком случае сигнал не будет доставлен до тех пор, пока не будет разблокирован. Сигналы SIGKILL и SIGSTOP являются специальными, и приложения не могут игнорировать, блокировать или определять собственные обработчики для них. Полный список сигналов приведен в табл. 4.1, в которой также имеются ссылки на их действия по умолчанию и существующие ограничения.

---

[1] В системе 4.4BSD файл дампа имеет название core.prog, где prog — это первые 16 символов программы, выполнявшейся во время получения сигнала (такой подход более разумен, так как в случае аварийного завершения еще какого-либо процесса, имеющего того же владельца, файл core не затрется новым). — *Прим. ред.*

Таблица 4.1. Сигналы UNIX

Сигнал	Описание	Действие по умолчанию	Доступен в[1]	Примечания[2]
SIGABRT	Процесс аварийно завершен	Аварийное завершение	APSB	
SIGALRM	Сигнал тревоги реального времени	Выход	OPSB	
SIGBUS	Ошибка шины	Аварийное завершение	OSB	
SIGCHLD	Потомок завершил работу или приостановлен	Игнорирование	OJSB	6
SIGCONT	Возобновить приостановленный процесс	Продолжение/ игнорирование	JSB	4
SIGEMT	Ловушка эмулятора	Аварийное завершение	OSB	
SIGFPE	Арифметическая ошибка	Аварийное завершение	OAPSB	2
SIGHUP	Освобождение линии терминала	Выход	OPSB	
SIGILL	Выполнение недопустимой инструкции	Аварийное завершение	OAPSB	
SIGINFO	Запрос состояния (Ctrl+T)	Игнорирование	B	
SIGINT	Прерывание терминала (Ctrl+C)	Выход	OAPSB	
SIGIO	Асинхронное событие ввода-вывода	Выход/ игнорирование	SB	3
SIGIOT	Ловушка ввода-вывода	Аварийное завершение	OSB	
SIGKILL	Завершить процесс	Выход	OPSB	1
SIGPIPE	Запись в канал при отсутствии считывающих процессов	Выход	OPSB	
SIGPOLL	Событие опрашиваемого устройства	Выход	S	
SIGPROF	Профилирование таймера	Выход	SB	
SIGPWR	Неполадки питания	Игнорирование	OS	
SIGQUIT	Сигнал выхода из терминала (Ctrl+\)	Аварийное завершение	OPSB	

Сигнал	Описание	Действие по умолчанию	Доступен в[1]	Примечания[2]
SIGSEGV	Ошибка сегментации	Аварийное завершение	OAPSB	
SIGSTOP	Остановить процесс	Остановка	JSB	1
SIGSYS	Неверный системный вызов	Выход	OAPSB	
SIGTERM	Завершить процесс	Выход	OAPSB	
SIGTRAP	Аппаратная ошибка	Аварийное завершение	OSB	2
SIGTSTP	Сигнал остановки терминала (Ctrl+Z)	Остановка	JSB	
SIGTTIN	Чтение из терминала фоновым процессом	Остановка	JSB	
SIGTTOU	Запись в терминал фоновым процессом	Остановка	JSB	5
SIGURG	Экстренное событие канала ввода-вывода	Игнорирование	SB	
SIGUSR1	Определяется произвольно	Выход	OPSB	
SIGUSR2	Определяется произвольно	Выход	OPSB	
SIGVTALRM	Сигнал тревоги виртуального времени	Выход	SB	
SIGWINCH	Изменение размера окна	Игнорирование	SB	
SIGXCPU	Превышение лимита процессора	Аварийное завершение	SB	
SIGXFSZ	Превышение лимита размера файла	Аварийное завершение	SB	

[1] Обозначения: O — сигнал системы SVR2; A — ANSI C; B — 4.3BSD; S — SVR4; P — POSIX.1; J — POSIX.1, при условии поддержки управления заданиями.

[2] 1 — не может быть перехвачен, блокирован или игнорирован; 2 — не сбрасывается в значение по умолчанию, даже в реализациях System V; 3 — в SVR4 действие по умолчанию — выход, в 4.3BSD — игнорирование; 4 — по умолчанию выполнение процесса, если тот был приостановлен, в ином случае сигнал игнорируется; 5 — не может быть заблокирован; 6 — процесс может решить: позволить запись в терминал фоновым процессам без генерации этого сигнала или нет; 7 — в SVR3 и более ранних версиях системы назывался SIGCLD.

Важно отметить, что любое действие, в том числе и завершение работы, относится только к тому процессу, которому был доставлен сигнал. Такой подход требует, по крайней мере, чтобы процесс был назначен в текущий момент на выполнение. В загруженных системах для процесса, обладающего низким приоритетом, ожидание в очереди планировщика может занять некоторое значительное количество времени. Еще одна задержка может произой-

ти, если процесс окажется выгруженным, приостановленным или блокированным без возможности прерывания.

Ядро системы предупреждает процесс о наличии ожидающих его сигналов при помощи вызова функции issig(), который делается от имени процесса для проверки ожидающих сигналов. Вызов функции issig() происходит только в следующих случаях:

- до возвращения в режим задачи после системного вызова или прерывания;
- сразу перед блокированием на прерываемом событии;
- немедленно после пробуждения от прерываемого события.

Если функция issig() возвращает значение TRUE, то ядро вызовет функцию psig() для диспетчеризации сигнала. Эта функция завершит процесс, создаст файл core (по необходимости) или же вызовет sendsig() для запуска обработчика, определенного в приложении. Функция sendsig() возвращает процесс в режим задачи, передает управление обработчику сигнала и указывает процессу на необходимость продолжения выполнения прерванного кода после завершения функционирования обработчика. Реализация этих функций сильно зависит от платформы, так как они должны манипулировать стеком приложения, а также сохранять, загружать и изменять контекст процесса.

Рис. 4.1. Обработка сигналов

Сигналы вырабатываются вследствие происхождения асинхронных событий, которые могут произойти после любой инструкции в области кода процесса. После завершения обработки сигнала процесс восстанавливает свое функционирование с того места, где его выполнение было прервано сигналом (рис. 4.1). Если сигнал приходит тогда, когда процесс находится в стадии выполнения системного вызова, ядро системы прерывает обработку вызова и возвращает ошибку EINTR. В ОС 4.2BSD был введен механизм автоматического рестарта системного вызова после сигнала (см. раздел 4.4.3). Система 4.3BSD предлагает вызов siginterrupt, отключающий эту возможность для конкретного сигнала.

4.2.2. Генерирование сигналов

Ядро системы вырабатывает сигналы для процессов в ответ на различные события, причиной возникновения которых может быть сам процесс, получаю-

щий сигнал, другой процесс, а также прерывания или внешние действия. Основными источниками сигналов являются:

- **исключительные состояния** (exceptions). После возникновения в процессе такого состояния (например, при попытке выполнения недопустимой инструкции) ядро системы уведомляет об этом процесс при помощи сигнала;
- **другие процессы** (other processes). Процесс может отправлять сигналы другому процессу или набору процессов при помощи системных вызовов kill или sigsend. Процесс может послать сигнал даже самому себе;
- **прерывания от терминала** (terminal interrupts). При нажатии определенных комбинаций клавиш, например Ctrl+C или Ctrl+\, *текущему* (*интерактивному*) процессу терминала отправляется сигнал. Команда stty позволяет пользователю назначать каждому сигналу, создаваемому терминалом, определенные клавиши;
- **управление заданиями** (job control). Фоновые процессы, желающие произвести операции чтения или записи с терминалом, посылают сигналы управления заданиями. Командные интерпретаторы, поддерживающие управление заданиями, такие как csh или ksh, используют сигналы для управления текущими и фоновыми процессами. Когда процесс завершается или приостанавливается, ядро системы уведомляет об этом его родителя посредством сигнала;
- **квоты** (quotas). Если процесс превысит отведенные ему лимиты использования процессора или допустимого размера файла, ядро пошлет такому процессу сигнал;
- **уведомления** (notifications). Процессу может потребоваться уведомление о возникновении определенных событий, например готовности устройства на ввод-вывод. Ядро информирует процесс о таких событиях при помощи сигнала;
- **будильники** (alarms). Процесс может установить будильник на определенное время. Когда отведенное время истечет, ядро предупредит об этом процесс подачей сигнала тревоги.

Существуют три различных типа будильников, которые используют различные типы отсчетов. Так, отсчет ITIMER_REAL приводит к измерению реального времени и вырабатыванию будильником сигнала SIGALRM. При указании ITIMER_VIRTUAL измеряется виртуальное время, то есть тот промежуток, когда процесс находился в режиме задачи, и подается сигнал SIGVTALRM. С ITIMER_PROF ведется отсчет общего времени, используемого процессом как в режиме задачи, так и в режиме ядра, при этом генерируется сигнал SIGPROF.

Реализация будильников и отсчетов различается у разных производителей операционных систем.

4.2.3. Типичные примеры возникновения сигналов

Разберем несколько примеров генерирования и доставки сигналов. Представьте, что пользователь нажимает комбинацию клавиш Ctrl+C на своем терминале. Это действие приводит к возникновению прерывания терминала (точно так же, как и ввод любого иного символа с клавиатуры). Драйвер терминала распознает комбинацию как символы, по вводу которых вырабатывается определенный сигнал, после чего отправляет созданный им сигнал SIGINT текущему процессу данного терминала (если текущее *задание* состоит из более чем одного процесса, драйвер пошлет сигнал каждому его процессу). Когда этот процесс будет выбран планировщиком на выполнение, то он увидит сигнал, пытающийся вернуться в режим задачи после переключения контекста. Иногда интерактивный процесс уже является *текущим* во время возникновения прерывания. В таком случае обработчик прерываний прервет его выполнение и отправит процессу сигнал[1]. После возврата из прерывания процесс осуществит проверку на наличие ожидающих сигналов и обнаружит этот сигнал.

Однако *исключительные состояния* (или *исключения*) обычно приводят к возникновению синхронных сигналов. Причиной их часто являются ошибки в программе (например, попытка деления на ноль, недопустимые инструкции и т. д.), которые будут возникать в том же самом месте программы в случае, если она будет запущена с тождественными условиями (то есть если повторяется та же ветвь кода с теми же данными). Когда в программе происходит исключение, оно является причиной возникновения *ловушки* в режиме ядра. Обработчик ловушки, находящийся внутри ядра, распознает исключительное состояние и отправляет соответствующий сигнал текущему процессу. Перед возвратом в режим задачи обработчик ловушки вызывает функцию issig() для получения процессом сигнала.

Возможна ситуация, когда процесс одновременно ожидают несколько сигналов. В таком случае все сигналы будут обрабатываться по одному. Сигнал также может прийти в момент выполнения обработчика другого сигнала, что может стать причиной наложения обработчиков. В большинстве реализаций систем UNIX приложение имеет право запрашивать ядро о выборочной блокировке определенных сигналов перед запуском определенного обработчика (см. подробнее в разделе 4.4.3). Такой подход позволяет исключить или контролировать наложение обработчиков сигналов.

---

[1] На многопроцессорных системах процесс может оказаться функционирующим на другом процессоре. В таком случае обработчик должен создать специальное межпроцессорное прерывание для доставки сигнала его приемнику.

4.2.4. Спящие процессы и сигналы

Что происходит в случае, если спящий процесс получит сигнал? Должен ли он быть разбужен для обработки этого сигнала или сигнал будет ожидать того момента, когда процесс выйдет из режима сна?

Ответ на эти вопросы зависит от того, что стало причиной перехода процесса в режим сна. Если процесс спит в ожидании возникновения такого события, как завершение ввода-вывода с диска, то есть события, которое может наступить вскоре, имеет смысл немного подождать с доставкой сигналов. С другой стороны, если процесс ожидает ввода символа с клавиатуры, то такое ожидание может продолжаться очень долго. Нам необходимо определиться, как прерывать такие процессы сигналами.

Система UNIX поддерживает два вида сна: прерываемый и непрерываемый. Процесс, находящийся в состоянии сна до события, которое может произойти в ближайшее время (например, завершение дискового ввода-вывода), пребывает в непрерываемом сне и не может быть побеспокоен поступающими сигналами. Процесс, ожидающий такого события, как ввод-вывод терминала, который может не произойти в течение продолжительного времени, находится в прерываемом состоянии сна и будет разбужен посланным ему сигналом.

Если сигнал предназначается для процесса, находящегося в непрерываемом сне, то такой сигнал будет помечен как ожидающий, и больше никаких действий со стороны этого сигнала по отношению к процессу происходить не будет. Процесс не будет уведомлен о сигнале и после выхода из состояния сна до тех пор, пока он не окажется в состоянии возврата в режим задачи или не будет блокирован по прерываемому событию.

Если процесс находится в состоянии, когда он будет заблокирован по прерываемому событию, то перед этой процедурой он проверит наличие сигналов. Если таковые будут найдены, процесс обработает их и прервет выполнение системного вызова. Если сигнал будет создан уже после того, как процесс был блокирован, ядро системы разбудит такой процесс. Пробуждение и начало работы процесса может быть вызвано двумя причинами: либо произошло ожидаемое событие, либо его сон был прерван сигналом, поэтому в первую очередь процесс вызовет функцию issig() и проверит, есть ли для него сигналы. В случае когда сигнал ожидает процесс, после вызова issig() всегда происходит еще один вызов psig(), как это показано ниже:

```
if (issig())
  psig();
```

4.3. Ненадежные сигналы

Изначальная реализация сигналов (в системах SVR2 и более ранних версиях) была ненадежной и малоэффективной [2]. Эта реализация придерживается базовой модели, описанной в предыдущем разделе, и обладает рядом недостатков.

Наиболее важной проблемой является надежность доставки сигналов. Обработчики сигналов не являются постоянно установленными и не маскируют повторения одного и того же сигнала. Представьте, что программист установил обработчик на определенный сигнал. После возникновения этого сигнала ядро системы сбросит действия, определенные для него, на установки по умолчанию перед тем, как вызовет обработчик. Программа, желающая обработать все повторные сигналы, должна каждый раз переустанавливать обработчик, как это показано в листинге 4.1.

Листинг 4.1. Переустановка обработчика сигналов

```
void sigint_handler (sig)
int sig;
{
   signal (SIGINT, sigint_handler); /* переустановка обработчика */
        /* обработка сигнала*/
}
main()
{
   signal (SIGINT, sigint_handler); /* установка обработчика */

}
```

Однако этот подход приводит к состоянию состязательности. Представьте, что пользователь дважды быстро нажимает комбинацию клавиш Ctrl+C. Первое нажатие влечет за собой возникновение сигнала SIGINT, действие которого сбрасывается на принятое по умолчанию, после чего запустится обработчик. Если повторное нажатие произошло до переустановки обработчика, ядро системы выполнит действие по умолчанию и завершит процесс. Такой подход приводит к существованию определенного промежутка времени между запуском и переустановкой обработчика, в течение которого сигнал не может быть перехвачен. По этой причине говорят, что ранние реализации UNIX имеют *ненадежные сигналы*.

Существует также и проблема производительности, относящаяся к спящим процессам. В ранних реализациях вся информация, относящаяся к диспозиции сигналов[1], сохраняется в массиве u_signal[ ], который содержит одну запись о каждом типе сигналов. Массив располагается в области u. Элемент этого массива содержит адрес обработчика, определенного в приложении, а также параметр SIG_DFL, в котором регламентируется действие по умолчанию, или используется SIG_IGN для указания на необходимость игнорирования сигнала.

Так как ядро может считывать данные только из области u текущего процесса, оно не знает, каким образом другой процесс должен распорядиться сигналом. Более того, если ядру системы необходимо послать сигнал про-

[1] То есть поведению при их получении. — *Прим. ред.*

цессу, находящемуся в состоянии прерываемого сна, оно не может знать, игнорирует процесс такой сигнал или нет. Таким образом оно отправит сигнал и разбудит тем самым процесс, предполагая, что тот обработает сигнал. Если процесс обнаружит, что он разбужен сигналом, который им игнорируется, он просто снова заснет. Такие совершенно излишние пробуждения приводят в результате к совершенно ненужным переключениям контекста и потере времени на обработку сигналов. Очевидно, что лучшим вариантом представляется распознавание сигналов ядром и сброс тех из них, которые должны игнорироваться, без необходимости участия в этом процесса.

В завершение скажем, что система SVR2 не обладает средствами временного блокирования сигналов для задержки их доставки до тех пор, пока они не будут разблокированы. Эта система также не имеет поддержки управления заданиями, при которой группы процессов могут быть приостановлены и возобновлены для того, чтобы получить доступ к терминалу.

4.4. Надежные сигналы

Проблемы, озвученные в предыдущем разделе, были впервые решены в системе 4.2BSD, в которой был представлен управляющий механизм надежных и гибких в обработке сигналов. В ОС 4.3BSD были сделаны дополнительные усовершенствования, но базовые средства остались неизменными. Между тем компания AT&T представила собственную версию надежных сигналов в своей системе SVR3 [1]. Эта версия была несовместима с интерфейсом BSD и оказалась не настолько развитой. Разработчики сохранили в реализации SVR3 совместимость с исходным механизмом сигналов, который был в SVR2. В обеих системах, 4.2BSD и SVR3, была предпринята попытка решения одних и те же проблем разными способами. В итоге каждая из этих ОС обладает собственным набором системных вызовов, используемых для доступа к средствам управления сигналами. Эти вызовы имеют как различные имена, так и отличающуюся друг от друга семантику.

Стандарт POSIX.1 явился попыткой навести порядок в имеющемся хаосе, определив стандартный набор функций, которые должны быть реализованы во всех системах, претендующих на совместимость с ним. Функции согласно этому стандарту могут быть реализованы как в виде системных вызовов, так и в виде библиотечных процедур. На основе требований POSIX в системе SVR4 был представлен новый интерфейс, который удовлетворял стандарту POSIX и оказался совместим с BSD и со всеми предыдущими версиями UNIX, созданными ранее компанией AT&T.

Этот раздел начинается с описания основных возможностей механизма надежных сигналов. Затем вы увидите краткое описание интерфейсов систем SVR3 и 4.3BSD, а в конце раздела вы познакомитесь с подробным изложением сигнального интерфейса ОС SVR4.

4.4.1. Основные возможности

Все реализации механизма надежных сигналов обладают некоторыми общими возможностями, перечисленными ниже.

- **Постоянно установленные обработчики** (persistent handlers). Обработчики сигналов остаются установленными даже после возникновения сигнала и не требуют дополнительных переустановок. Такой подход защищает от существования временного интервала между запуском обработчика сигнала и его переустановкой, во время которого повторно поступивший сигнал может завершить процесс.
- **Маскирование** (masking). Сигнал может быть временно маскирован (слова «блокирован» и «маскирован» являются синонимами и взаимозаменяемы при разговоре о сигналах). Если вырабатывается сигнал, который уже блокирован процессом, ядро системы будет помнить о этом и не станет посылать такой сигнал процессу незамедлительно. Сигнал будет переправлен и обработан после того, как процесс разблокирует его. Такой подход дает возможность программисту защитить критические области кода от прерывания его выполнения при возникновении определенных сигналов.
- **Спящие процессы** (sleeping process). Некоторая информация о диспозиции сигналов в процессе является видимой для ядра (посредством хранения данных в структуре `proc` вместо области `u`), даже если процесс не находится в текущий момент на выполнении. Следовательно, если ядро генерирует сигнал для процесса, находящегося в прерываемом сне, но тот игнорирует или блокирует данный сигнал, то ядро не станет будить такой процесс.
- **Разблокирование и ожидание** (unblock and wait). Системный вызов `pause` блокирует процесс до тех пор, пока ему не будет доставлен сигнал. Механизм надежных сигналов предлагает еще один вызов, `sigpause`, который атомарно демаскирует сигнал и блокирует процесс до тех пор, пока тот не получит такой сигнал. Если демаскированный сигнал уже находится в ожидании, то произойдет немедленный возврат из этого системного вызова.

4.4.2. Сигналы в системе SVR3

Система SVR3 поддерживает все возможности, описанные в предыдущем разделе. Однако реализация механизма сигналов в этой ОС имеет некоторые недостатки. Проиллюстрируем это на примере с использованием системного вызова `sigpause`.

Представьте, что процессом объявлен обработчик, перехватывающий сигнал `SIGQUIT` и устанавливающий глобальный флаг при его захвате. Процесс

единожды проверяет флаг и, если тот не установлен, ждет его установки. Проверка и последующее ожидание вместе представляют собой *критический участок* кода: если сигнал будет доставлен после проведения проверки, но до начала ожидания, он будет потерян[1] и, следовательно, процесс может ожидать сигнал вечно. Таким образом, процессу необходимо маскировать сигнал SIGQUIT на время проверки флага. Но если процесс войдет в режим ожидания с маскированным сигналом, то в таком случае сигнал никогда не будет доставлен. Следовательно, нам необходим некий атомарный вызов, который бы демаскировал сигнал и блокировал процесс в ожидании. Эту функцию обеспечивает системный вызов sigpause. Пример кода, работающего в SVR3, представлен в листинге 4.2.

Листинг 4.2. Использование sigpause для ожидания сигнала

```
int sig_received = 0;
void handler(int sig)
{
      sig_received++;
}
main()
{
      sigset(SIGQUIT, handler);
      ...
/* ждем сигнала, если он уже не находится в режиме ожидания*/
      sighold(SIGQUIT);
      while (sig_received==0) /* сигнал еще не доставлен */
         sigpause(SIGINT);[2]
      /* сигнал прибыл, обработка*/
      ...
}
```

Пример показывает некоторые возможности системы SVR3 по обработке сигналов. Вызовы sighold и sigrelse позволяют блокировать и разблокировать сигнал. Вызов sigpause атомарно деблокирует сигнал и переводит процесс в состояние сна до тех пор, пока тот не получит сигнал, который им не игнорируется и не заблокирован. Системный вызов sigset определяет постоянный обработчик, не сбрасываемый в действие по умолчанию после возникновения сигнала. Старый вызов signal остался в системе для обратной совместимости. Обработчики, задаваемые при помощи этого вызова, не являются постоянными.

Такой интерфейс обладает несколькими недостатками [9]. Важно то, что системные вызовы sighold, sigrelse и sigpause могут работать только с одним

[1] Так как и флаг установится уже после его проверки. — *Прим. ред.*

[2] В строке, видимо, ошибка, и вместо SIGINT должно быть SIGQUIT, иначе незачем вызывать sighold на SIGQUIT, ибо это приведет к блокированию обработчика SIGQUIT и, следовательно, флаг никогда не изменит своего значения, что, в свою очередь, не даст выйти из цикла. — *Прим. ред.*

сигналом в один момент времени. Не существует способа атомарного блокирования или деблокирования нескольких сигналов одновременно. Если обработчик, показанный в листинге 4.2, будет использован несколькими сигналами сразу, не существует приемлемого способа программно выделить критическую область. Мы можем блокировать сигналы по одному в один момент времени, но вызов sigpause не сумеет атомарно разблокировать все эти сигналы и далее перевести процесс в состояние ожидания.

В системе SVR3 также отсутствует управление заданиями и такие возможности, как автоматический перезапуск системных вызовов. Такие возможности (наряду с некоторыми другими) представлены в ОС 4BSD.

4.4.3. Механизм сигналов в BSD

Впервые механизм надежных сигналов был представлен в ОС 4.2BSD. Возможности, предлагаемые механизмом сигналов в системе BSD [7], являются более развитыми, чем аналогичные возможности, представленные в ОС SVR3. В большинстве системных вызовов одним из входных аргументов передается 32-битовая маска сигналов, биты которой отображают, с какими из сигналов будет оперировать функция (по одному биту на каждый сигнал). Такой подход позволяет одному системному вызову работать сразу с несколькими сигналами. Вызов sigsetmask используется для указания набора блокируемых сигналов. Вызов sigblock добавляет в этот набор один или несколько дополнительных сигналов. Реализация вызова sigpause в системе BSD атомарно устанавливает новую маску блокируемых сигналов и переводит процесс в состояние сна до прихода сигнала.

Системный вызов sigvec заменил собою signal. Точно так же, как и signal, sigvec устанавливает обработчик для одного из сигналов. Дополнительно вызов sigvec может задавать маску, ассоциируемую с этим сигналом. После вырабатывания такого сигнала ядро системы перед вызовом обработчика установит новую маску блокируемых сигналов, являющуюся объединением текущей маски, маски, заданной в sigvec, и текущего сигнала.

Таким образом, обработчик всегда запускается, когда текущий сигнал блокирован, — следовательно, повторяющийся сигнал не будет доставлен до тех пор, пока обработчик не завершит свою работу. Такое устройство сигнального механизма наиболее соответствует типичным сценариям вызова обработчиков сигналов. Блокирование дополнительных сигналов, вырабатываемых во время функционирования обработчика, является весьма необходимой функцией, так как сами по себе обработчики сигналов обычно являются критическими участками кода. После возврата из обработчика происходит восстановление маски блокированных сигналов в ее предыдущее значение.

Еще одной важной возможностью является обработка сигналов в отдельном стеке. Представьте, что процесс управляет своим собственным стеком. Он может установить обработчик для сигнала SIGSEGV, вырабатываемого при

переполнении стека. В обычной ситуации обработчик запустится с тем же, уже переполненным стеком, что приведет к вырабатыванию еще одного сигнала SIGSEGV. Если обработчик будет стартовать, используя при этом отдельный стек, то проблема не возникнет. Отдельный стек для сигнала также полезен и для нитевых библиотек прикладного уровня. Системный вызов sigstack задает отдельный стек, который будет использоваться обработчиком сигнала. Ответственность за правильное указание размера такого стека лежит на разработчике, поскольку ядро системы ничего не знает о его границах.

В ОС BSD представлено несколько дополнительных сигналов, в том числе специально выделенных для *управления заданиями*[1]. Задание — это группа связанных между собой процессов, обычно формирующих единый конвейер. Пользователь может выполнять несколько заданий одновременно из одного сеанса терминала, но только один из них будет текущим. Текущему заданию позволено писать в терминал и считывать с него. Фоновым заданиям, пытающимся получить доступ к терминалу, посылаются сигналы, которые обычно приостанавливают процесс. Командные интерпретаторы Korn shell (ksh) и C shell (csh) [6] используют сигналы управления заданиями для манипулирования заданиями, посылая эти сигналы текущим и фоновым заданиям, приостанавливая и возобновляя их работу. Более подробно об управлении заданиями будет рассказано в разделе 4.9.1.

В заключение отметим, что система 4BSD позволяет автоматически перезапускать медленные системные вызовы, выполнение которых было прервано сигналами. Медленные вызовы включают в себя функции read и write, осуществляющие чтение и запись на символьные устройства, а также сетевые соединения, каналы, вызовы wait, waitpid, ioctl. Если один из этих вызовов прерывается сигналом, происходит его автоматический перезапуск после возврата из обработчика сигнала вместо прерывания работы с ошибкой EINTR. В системе 4.3BSD добавлен вызов siginterrupt, позволяющий выборочно разрешить или запретить такую возможность для каждого сигнала отдельно.

Интерфейс сигналов системы BSD является весьма гибким и мощным. Его основным недостатком остается несовместимость с оригинальным интерфейсом систем корпорации AT&T (и даже с вариантом, представленным в SVR3, хотя эта система была создана позже). Это дало возможность сторонним производителям предлагать различные библиотечные интерфейсы, которые пытались удовлетворить приверженцев обеих ветвей генеалогического дерева UNIX. Позже в системе SVR4 был представлен интерфейс, совместимый со стандартом POSIX и при этом обладающий совместимостью с предыдущими реализациями System V и семантикой, принятой в ОС BSD.

[1] Поддержка управления заданиями впервые появилась в 4.1BSD.

4.5. Сигналы в SVR4

Операционная система SVR4 предлагает набор системных вызовов [11], которые обеспечивают универсальные возможности обработки сигналов как ОС SVR3, так и BSD, а также поддерживается устаревший механизм ненадежных сигналов. Ниже представлены основные функции SVR4 для работы с сигналами.

- sigprocmask(how, setp, osetp);
- Аргумент setp используется для изменения маски блокируемых сигналов. Если how имеет значение SIG_BLOCK, то маска setp объединяется операцией ИЛИ с существующей. Если how определено как SIG_UNBLOCK, то сигналы, заданные в setp, деблокируются по существующей маске блокированных сигналов. Если how равняется SIG_SETMASK, то происходит замена текущей маски на набор, определенный в setp. При возврате из функции osetp содержит значение маски перед модификацией.
- sigaltstack(stack, old_stack);
- Задает новый стек stack для обработки сигналов. Альтернативный стек (если он требуется) необходимо определять перед установкой обработчика. Остальные обработчики используют стек, заданный по умолчанию. После возврата из функции в переменной old_stack содержится указатель на предыдущий альтернативный стек.
- sigsuspend(sigmask);
- Устанавливает маску блокируемых сигналов в значение sigmask и переводит процесс в состояние сна до тех пор, пока этому процессу не будет отправлен сигнал, который не игнорируется и не заблокирован. Если такой сигнал был послан и при изменении маски он будет разблокирован, произойдет немедленный выход из функции.
- sigpending(setp);
- Возвращает в setp набор сигналов, ожидающих процесс. Вызов не производит никаких изменений в состоянии сигналов и используется только для получения информации.
- sigsendset(procset, sig);
- Расширенная версия kill. Посылает сигнал sig набору процессов, заданных в procset.
- sigaction(signo, act, oact);
- Определяет обработчик для сигнала signo. Является аналогом вызова sigvec в ОС BSD. Аргумент act указывает на структуру sigaction, содержащую диспозицию сигналов (SIG_IGN, SIG_DFL или адрес обработчи-

ка), маску, ассоциированную с сигналом (аналогичную маске вызова sigvec), а также один или несколько следующих флагов:

SA_NOCLDSTOP	Не генерировать сигнал SIGCHLD, когда процесс-потомок приостановлен
SA_RESTART	Автоматический рестарт системного вызова при прерывании его сигналом
SA_ONSTACK	Обработка сигнала с альтернативным стеком, если такой стек был указан через sigaltstack
SA_NOCLDWAIT	Используется только с SIGCHLD. Просит систему не создавать процессы-зомби, если потомки вызывающего процесса завершают свою работу. Если процесс далее вызовет wait, то он будет находиться в режиме ожидания до тех пор, пока не завершат работу все его потомки
SA_SIGINFO	Обеспечивает дополнительную информацию для обработчика сигнала. Используется для обработки аппаратных исключений и т. д.
SA_NODEFER	Позволяет не блокировать автоматически сигнал в течение выполнения его обработчика
SA_RESETHAND	Сбрасывает действия на заданные по умолчанию перед вызовом обработчика

- Флаги SA_NODEFER и SA_RESETHAND используются для совместимости с изначальной реализацией механизма ненадежных сигналов. Во всех случаях переменная oact возвращает данные, установленные перед вызовом sigaction.
- *Интерфейс совместимости*
- Для обеспечения совместимости с предыдущими версиями система SVR4 также поддерживает вызовы signal, sigset, sighold, sigrelse, sigignore и sigpause. Системы, не требующие совместимости на бинарном уровне, могут реализовывать эти вызовы в виде библиотечных функций.

Все перечисленные системные вызовы (кроме указанных в последнем пункте списка) полностью удовлетворяют стандарту POSIX.1 по имени, передаваемым параметрам и семантике.

4.6. Реализация сигналов

Для эффективной реализации сигналов ядру необходимо содержать некоторое состояние в области u и в структуре proc. В этом разделе описывается реализация сигналов в системе SVR4, которая отличается от аналогичного набора ОС BSD именами некоторых переменных и функций. Область u содержит информацию, требующуюся для правильного запуска обработчиков сигналов, которую составляют нижеперечисленные поля области u.

u_signal[]	Вектор обработчиков для каждого сигнала
u_sigmask[]	Маски сигналов, ассоциированные с каждым обработчиком
u_sigaltstack	Указатель на альтернативный стек сигнала
u_sigonstack	Маска сигналов, обрабатываемых с альтернативным стеком
u_oldsig	Набор обработчиков, который должен имитировать устаревший механизм ненадежных сигналов

Структура proc содержит определенные поля, относящиеся к созданию и отправке сигналов, в том числе:

p_cursig	Текущий сигнал, в данный момент обрабатываемый
p_sig	Маска ожидающих сигналов
p_hold	Маска блокируемых сигналов
p_ignore	Маска игнорируемых сигналов

Рассмотрим далее реализации на уровне ядра различных функций, относящихся к доставке сигналов.

4.6.1. Генерация сигналов

После вырабатывания сигнала ядро проверяет структуру proc процесса, которому этот сигнал предназначен. Если сигнал необходимо проигнорировать, то ядро на этом завершит обработку, не предпринимая никаких действий. В иных случаях он будет добавлен в набор ожидающих сигналов, расположенный в поле p_cursig[1]. Так как p_cursig[2] является всего лишь битовой маской, где каждому сигналу отводится только один бит, ядро не может записать в нее несколько повторных экземпляров одного сигнала. Следовательно, процесс будет знать только об одном приходе ожидающего сигнала.

Если процесс находится в прерываемом сне и сигнал не заблокирован, то ядро системы разбудит такой процесс для получения сигнала. Более того, сигналы управления заданиями, такие как SIGSTOP или SIGCONT, напрямую приостанавливают или продолжают функционирование процесса без проведения их доставки.

4.6.2. Доставка и обработка

Процесс ведет проверку на наличие сигналов при помощи функции issig(), вызываемой после обработки системного вызова или прерывания, но перед возвращением из режима ядра. Вызов issig() производится также и в случаях

[1] Здесь, видимо, ошибка, и имеется в виду все-таки поле p_sig. — *Прим. ред.*

[2] Опять же, должно быть, не p_cursig, а p_sig. — *Прим. ред.*

перехода в режим прерываемого сна или выхода из него (то есть пробуждения). Функция issig() считывает установленные биты в поле p_cursig[1]. Если какой-либо бит установлен, то эта функция проверяет p_hold на предмет существования блокировки этого сигнала. Если сигнал не блокируется, то issig() сохраняет номер сигнала в p_sig[2] и возвращает значение TRUE.

Когда сигнал является ожидающим, ядро вызывает для его обработки psig(). Функция psig() проверяет информацию области u, относящуюся к этому сигналу. Если не задан ни один обработчик, произойдет действие по умолчанию, обычно это завершение процесса. Если необходимо вызвать обработчик, то происходит изменение маски блокируемых сигналов p_hold, в которую добавляется текущий сигнал, а также любой другой сигнал, ассоциируемый с ним в маске u_sigmask. Текущий сигнал не добавляется в маску в том случае, если для его обработчика был установлен флаг SA_NODEFER. Точно так же, если был установлен флаг SA_RESETHAND, действия, задаваемые в массиве u_signal[ ], сбрасываются на установленные по умолчанию.

Последним этапом действий является вызов sendsig(), производимый функцией psig(), который заставляет процесс вернуться в режим задачи и передать управление обработчику. Вызов sendsig() гарантирует, что после завершения работы обработчика процесс восстановит свое выполнение с того места программы, на котором он был прерван сигналом. Если используется альтернативный стек, вызов sendsig() загрузит обработчик с указанным стеком. Реализация функции sendsig() является машинно-зависимой, так как ей необходимо знать о деталях работы со стеком и о манипуляциях с контекстом.

4.7. Исключительные состояния

Исключительные[3] состояния (исключения) возникают, когда программа оказывается в необычной ситуации, и происходит это, как правило, вследствие ошибки. Например, попытка обращения по несуществующему адресу в памяти или деление на ноль повлекут за собой исключение. При возникновении исключения срабатывает ловушка в ядре, которая вырабатывает сигнал, уведомляющий процесс о возникшем исключении.

В системе UNIX для оповещения процесса об исключениях служат сигналы. Тип вырабатываемого сигнала зависит от природы исключения. Например, попытка обращения по несуществующему адресу может повлечь за собой генерацию сигнала SIGSEGV. Если в приложении указан для него обработчик, то ядро системы запустит его. Если нет, то выполняется действие по умолчанию (обычно это завершение работы процесса). Такой подход позволяет каждой

[1] Все-таки, наверное, в маске ожидающих сигналов p_sig. — *Прим. ред.*

[2] Вот тут, видимо, должно быть поле p_cursig. — *Прим. ред.*

[3] Этот раздел описывает только аппаратные исключения. Их не следует путать с программными исключениями, поддерживаемыми различными языками программирования, такими как C++.

программе устанавливать свои собственные обработчики исключений. Некоторые языки программирования (например, Ада) обладают встроенными механизмами обработки исключений. Обработчики могут быть реализованы в библиотеках языка.

Исключения часто используются отладчиками. При отладке (или трассировке) программы вырабатывают исключения в точках останова, а также по завершении выполнения системного вызова exec. Отладчик должен перехватывать эти исключения для контроля над программой. Отладчику также может понадобиться перехватывать и другие выбранные им исключения и сигналы, вырабатываемые отлаживаемой программой. Системный вызов ptrace системы UNIX разрешает такой перехват (более подробно о его работе можно прочесть в разделе 6.2.4).

Существует ряд недостатков в том способе, которым UNIX обрабатывает исключения. Во-первых, *обработчик исключения запускается в том же контексте, в котором это исключение произошло.* Это означает, что обработчик не имеет доступа к полному контексту регистров, существовавшему во время возникновения исключения. Когда происходит исключение, ядро системы передает лишь некоторый объем его контекста обработчику. Этот объем зависит от конкретной реализации UNIX, а также от аппаратной платформы, на которой работает система. Вообще говоря, одна нить должна работать с двумя контекстами — с контекстом обработчика и тем контекстом, в котором произошло исключение.

Во-вторых, *механизм сигналов изначально разрабатывался для однонитевых процессов.* Системы UNIX, поддерживающие многонитевые процессы, столкнулись с определенными сложностями в адаптации подобной схемы сигналов. И последний недостаток, о котором необходимо упомянуть, это ограничение системного вызова ptrace, который *позволяет отладчику, основанному на нем, контролировать только своих непосредственных потомков.*

4.8. Обработка исключительных состояний в Mach

Ограничения механизмов обработки исключений в UNIX стали причиной разработки нового унифицированного средства, представленного в системе Mach [4]. Целью разработчиков ОС стало создание такого механизма, который, с одной стороны, оставался бы совместимым с традиционной UNIX на бинарном уровне, а с другой — поддерживал многонитевые приложения. Это средство стало не только частью описываемой системы, но и частью ОС OSF/1, основанной на Mach.

Разработчики системы Mach отказались от идеи запуска обработчика в том же контексте, в котором произошло исключение. В традиционной UNIX вариант одного контекста применялся только потому, что обработчику требовался доступ и функционирование в том же адресном пространстве, где про-

изошло исключение. Так как система Mach является многонитевой, обработчик может стартовать как отдельная нить той же задачи. (Нити и задачи ОС Mach подробнее описаны в разделе 6.4. Вкратце *задача* объединяет набор ресурсов, в том числе и адресное пространство. *Нить* выполняется внутри задачи и представлена контекстом выполнения и контрольной точкой. Традиционный процесс UNIX в Mach можно интерпретировать как задачу, содержащую единственную нить.)

В системе Mach выделяются два понятия: нить-«жертва» (нить, в которой произошло исключение) и ее обработчик. На рис. 4.2 показано взаимодействие между ними. Нить-жертва *устанавливает* исключение, уведомляя ядро о его возникновении. Затем она *ожидает* окончания функционирования обработчика этого исключения. Обработчик *перехватывает* возникшее исключение, получая уведомление от ядра. Это уведомление содержит информацию о нити-жертве и о типе исключения. Затем происходит обработка исключения и его *очистка*, что дает возможность нити-жертве продолжить свое функционирование. Если обработка исключения не смогла завершиться успешно, то нить, в которой оно возникло, будет завершена.

Описанные взаимодействия в чем-то похожи на поток управления исключениями UNIX, отличаясь от него тем, что обработчик функционирует как отдельная нить. В результате операции *установки*, *ожидания*, *перехвата* и *очистки* составляют вместе *удаленный вызов процедуры*, реализованный в системе Mach через средства взаимодействия процессов (IPC, см. о них подробнее в разделе 6.4).

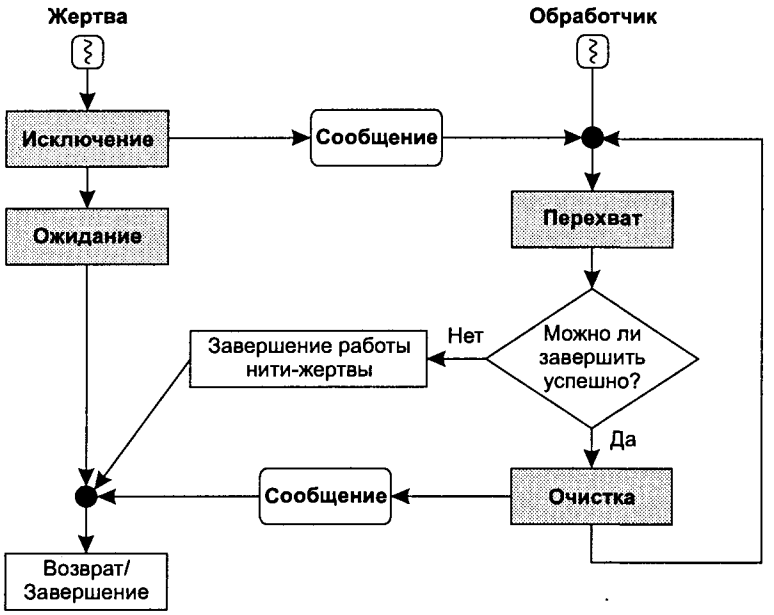


Рис. 4.2. Обработка исключительных состояний в системе Mach

При обработке каждого исключения создаются два сообщения. Когда нить-жертва устанавливает исключение, она посылает сообщение обработчику и затем ожидает ответа. Обработчик перехватывает это исключение сразу после получения сообщения и очищает его при помощи отправки ответного сообщения нити-жертве. После получения ответного сообщения нить-жертва может продолжить свою работу.

4.8.1. Порты исключительных состояний

В операционной системе Mach сообщения отправляются в определенный *порт*, который представляет собой защищенную очередь сообщений. Несколько задач могут обладать *правом на отправку* сообщений в конкретный порт, но только одна задача имеет право получать сообщения из порта. В системе Mach каждой задаче, а также каждой нити задачи назначается по одному порту исключений. Это обеспечивает два способа обработки исключений, которые соответствуют двум вариантам применения исключений: обработке ошибок и отладке.

Обработчики ошибок ассоциируются с нитями, так как ошибки обычно влияют только на ту нить, в которой они породили исключение. Таким образом, каждая нить может иметь отличный от других нитей обработчик ошибки. Порт обработчика регистрируется как порт исключений нити. При создании новой нити ее порт исключительных состояний инициализируется в NULL, что означает, что нить изначально не имеет обработчика исключений.

Отладчик прикрепляется к задаче посредством регистрации одного из своих портов в качестве порта исключительных состояний задачи. Отладчик запускается в виде отдельной задачи и обладает *правами на получение* сообщений, отправленных в этот порт. Каждая задача наследует порт исключительных состояний от своего родителя. Это позволяет отладчикам контролировать всех потомков отлаживаемой задачи.

Так как исключение может использовать как порт исключений нити, так и порт исключений задачи, необходимо найти способ разрешения конфликта. Для этого необходимо заметить, что порт исключений нити используется обработчиками ошибок и будет прозрачным для отладчиков. Например, обработчик может ответить на ошибку потери значимости числа с плавающей точкой нулем как результатом операции. Такие исключения, как правило, не интересуют отладчик, который в обычных случаях применяется для перехвата только неисправимых ошибок. Таким образом, если в приложении установлен обработчик ошибок, то Mach при возникновении ошибки в приложении отдаст предпочтение в ее обслуживании обработчику, нежели отладчику.

Когда происходит исключение, оно отправляется в порт исключений нити, если таковой существует. То есть те исключения, для которых определен обработчик, являются невидимыми для отладчика. Если установленный обработчик не сможет успешно очистить исключение, то он перенаправит

его в порт исключений задачи. (Так как обработчик является одной из нитей той же задачи, что и нить-жертва, он имеет право доступа к порту исключений задания.) Если ни один из обработчиков не исправит ошибку, то ядро системы завершит работу нити-жертвы.

4.8.2. Обработка ошибок

Когда нить-жертва устанавливает исключение, в порт исключений самой нити либо задачи посылается сообщение, содержащее адрес обратного порта, идентифицирующего нить и задачу, в которой произошло исключение, и тип исключения. После обработки исключения обработчик отправляет ответное сообщение обратно в указанный порт. Задача, в которой произошло исключение, обладает правами на получение сообщений из этого порта, и нить-жертва ожидает считывания из порта ответного сообщения. Когда сообщение приходит, нить-жертва получает его и восстанавливает свое выполнение в обычном режиме.

Так как обработчик и жертва являются нитями одной задачи, обработчик разделяет адресное пространство нити-жертвы. Он также может иметь доступ и к контексту регистров нити-жертвы, используя для этого вызовы thread_get_state и thread_set_state.

Система Mach поддерживает совместимость с UNIX, поэтому обработчики сигналов должны в ней запускаться в том же контексте, что и нить, в которой возникло исключение. Такой подход противоположен философии Mach, согласно которой для обработки ошибок запускается отдельная нить. В Mach эта разница была сглажена за счет использования инициируемого системой обработчика. Когда происходит исключение, для которого установлен обработчик UNIX-сигнала, в особый инициируемый системой обработчик посылается сообщение. Этот обработчик изменяет нить-жертву так, что обработчик сигнала исполняется, когда нить-жертва восстанавливает свое выполнение. Он очищает исключение, приведшее к возникновению сигнала. За коррекцию стека после завершения работы обработчика сигнала отвечает приложение.

4.8.3. Взаимодействие с отладчиком

Отладчик управляет задачей посредством регистрации порта, в котором он имеет такие же права на получение, что и порт исключений задачи. Когда в нити этой задачи возникает исключение, которое не может быть очищено ее обработчиком ошибок, ядро отправляет сообщение в этот порт, и отладчик получает это сообщение. Исключение приводит только к останову нити-жертвы, все остальные нити задачи продолжают работу. Однако отладчик может при необходимости приостановить функционирование всей задачи, используя вызов task_suspend.

ОС Mach располагает несколькими средствами, которые отладчик вправе использовать для управления задачей. Он может обращаться к адресному пространству нити-жертвы при помощи функции vm_read или vm_write, а также к ее контексту регистров, используя для этого функцию thread_get_state или thread_set_state. Отладчик также может приостанавливать или возобновлять работу приложения, а также завершить его функционирование при помощи функции task_terminate.

Механизм IPC (межпроцессное взаимодействие) в системе Mach является узло-независимым, то есть сообщения могут отправляться в порт как на той же самой машине, так и на удаленные хосты. Специальная прикладная задача-сервер под названием netmsgserver расширяет возможности IPC, делая этот механизм прозрачным при использовании через сеть. Сервер выделяет специальные прокси-порты для всех удаленных портов, принимает на них все сообщения, адресованные соответствующим удаленным портам, и затем пересылает эти сообщения по сети. Таким образом, весь механизм передачи сообщений делается для отправителя этих сообщений прозрачным[1]. Это дает возможность отладчику управлять задачей на любом узле сети точно так же, как и на локальном узле.

4.8.4. Анализ

Технология обработки исключений, реализованная в операционной системе Mach, разрешила многие проблемы, имеющиеся в традиционных вариантах UNIX. Она оказалась более гибкой и предоставила некоторые возможности, не поддерживаемые в других реализациях ОС. Перечислим некоторые важные преимущества описываемой технологии:

- преодоление ограничения на отладчик, по которому он управлял только своими непосредственными потомками. Теперь он может контролировать любое задание, если имеет соответствующие полномочия;
- отладчик может присоединяться к работающему заданию[2]. Для этого он регистрирует один из своих портов как порт исключений отлаживаемой задачи. Он также может отсоединиться от задания, устанавливая порт исключений задачи в его первоначальное значение. Порт исключений является всего лишь средством связи между отладчиком и задачей, ядро системы не содержит в себе каких-либо средств поддержки отладки;
- расширение действия средств IPC системы Mach на сеть позволяет создавать распределенные отладчики;

---

[1] Ведь сообщения отправляются в порт на локальной машине. — *Прим. ред.*

[2] На сегодняшний день большинство отладчиков используют файловую систему /proc, которая позволяет получать доступ к адресному пространству несвязанным процессам. Таким образом, отладчики имеют возможность легкого присоединения и отсоединения от процесса. Во времена разработки системы Mach такие средства были редкостью.

- наличие отдельной нити для обработчика обеспечивает чистое разделение контекста обработчика и нити-жертвы, при этом позволяя обработчику иметь полный доступ к контексту нити-жертвы;
- происходит корректная обработка многонитевых процессов. При возникновении исключения приостанавливается работа только одной нити, вызвавшей это исключение, в то время как остальные продолжают функционировать в обычном режиме. Если исключения произойдут сразу в нескольких нитях, то каждая из них создаст отдельное сообщение, и исключение каждой будет обрабатываться независимо от остальных.

4.9. Группы процессов и управление терминалом

В системе UNIX существует понятие *групп процессов*. Оно используется для управления доступом с терминала и поддержки сеансов входа в систему. Устройство и реализация таких средств в различных вариантах UNIX сильно отличаются друг от друга. Этот раздел начинается с описания общих концепций, после чего приводится анализ основных реализаций для конкретных ОС.

4.9.1. Общие положения

Группы процессов. Каждый процесс относится к определенной группе процессов, которая идентифицируется через *идентификатор группы процессов* (process group ID). Этот механизм используется ядром для проведения некоторых действий сразу над всеми процессами группы. Каждая группа может иметь лидера. *Лидер группы* — это процесс, имеющий PID, совпадающий с идентификатором группы процессов. Обычно процесс наследует идентификатор группы от своего предка, а все остальные процессы в группе являются потомками лидера.

Управляющий терминал. Любой процесс может обладать управляющим терминалом. Чаще всего это терминал входа в систему, от которого процесс был создан. Все процессы одной группы разделяют между собой один и тот же управляющий терминал.

Файл /dev/tty. С управляющим терминалом каждого процесса связан специальный файл /dev/tty. Драйвер устройства, связанный с этим файлом, перенаправляет все запросы на соответствующий терминал. Например, в системе 4.3BSD номер устройства управляющего терминала хранится в поле u_ttyd области u. Чтение из терминала реализовано следующим образом:

```
(*cdevsw[major(u.u_ttyd)].d_read) (u.u_ttyd, flags);
```

Следовательно, если два процесса относятся к различным сеансам входа в систему и они оба откроют файл /dev/tty, то результатом станет доступ к разным терминалам.

Управляющая группа. Каждый терминал ассоциируется с группой процессов. Такая группа, называемая управляющей группой терминала, идентифицируется при помощи поля t_pgrp структуры tty этого терминала[1]. *Сигналы, вырабатываемые при вводе с клавиатуры, такие как* SIGINT *или* SIGQUIT, *посылаются всем процессам управляющей группы терминала*, то есть всем процессам, чье поле p_pgrp в структуре proc равно значению t_pgrp структуры tty этого терминала.

Управление заданиями. Этот механизм (реализованный в системах 4BSD и SVR4) позволяет приостановить и возобновить работу группы процессов, а также управлять ее доступом к терминалу. Командные интерпретаторы, поддерживающие управление заданиями, такие как csh или ksh, распознают специальные управляющие символы (обычно Ctrl+Z) и команды, такие как fg и bg, для доступа к описываемым возможностям. Драйвер терминала обеспечивает дополнительное управление, посредством которого процессы, не входящие в управляющую группу терминала, защищены от чтения из терминала или записи в него.

Оригинальная реализация System V формирует группы процессов главным образом как представления сеансов входа в систему и не имеет средств управления заданиями. В системе 4BSD с каждой введенной командной строкой командного интерпретатора ассоциируется новая группа процессов (следовательно, все процессы, связанные между собой конвейером командного интерпретатора, относятся к одной группе). Таким образом, в этой операционной системе появилось понятие задания. В SVR4 произведена унификация этих различающихся друг от друга подходов при помощи понятия сеанса. В следующих разделах вы увидите описание всех трех технологий, перечисленных выше, а также анализ их преимуществ и недостатков.

4.9.2. Модель SVR3

В операционной системе SVR3 (и более ранних версиях ОС компании AT&T) группы процессов показывают характеристики сеанса входа в систему терминала. Технология доступа с терминала в SVR3 проиллюстрирована на рис. 4.3. Ниже перечислены ее основные характеристики.

Группы процессов. Каждый процесс во время создания вызовом fork наследует идентификатор группы от своего родителя. Единственный способ изменения группы процесса заключается в вызове setgrp, который переустанавливает

[1] Драйвер содержит отдельную структуру tty для каждого терминала.

группу делающего этот вызов процесса на значение, равное его PID. Таким образом, процесс становится лидером новой группы. Все потомки этого процесса, созданные при помощи fork, присоединятся к его группе.

Управляющий терминал. Терминал принадлежит своей управляющей группе. То есть если процесс создает новую группу, то он теряет свой управляющий терминал. После этого первый терминал, им открываемый (который еще не является управляющим), становится его управляющим терминалом. Значение поля t_pgrp структуры tty для этого терминала установится в значение поля p_pgrp структуры proc процесса. Все потомки процесса наследуют управляющий терминал от своего лидера. Не может существовать две группы процессов с одним управляющим терминалом.

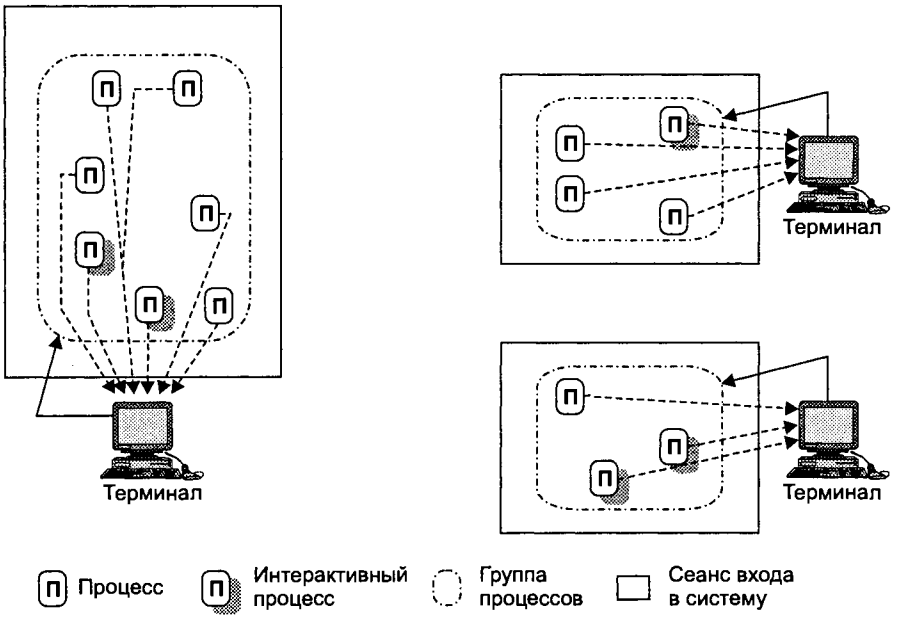


Рис. 4.3. Группы процессов в системе SVR3 UNIX

Типичный сценарий работы. Процесс init порождает процесс-потомок для каждого терминала, описанного в файле /etc/inittab. Потомок вызывает setgrp, становясь в результате этого лидером группы, после чего при помощи exec запускает программу getty, которая выводит приглашение на вход в систему и ждет ввода. Когда пользователь вводит свое регистрационное имя, программа getty посредством exec запускает программу login, которая запрашивает и верифицирует пароль, и затем стартует командный интерпретатор, назначенный[1] при входе в систему. Таким образом, командный интерпретатор,

[1] Конкретному пользователю. — *Прим. ред.*

запускаемый при входе в систему, является прямым потомком процесса init и лидером группы процессов. Обычно остальные процессы не создают собственных групп (исключение составляют системные демоны, запускаемые из сеанса входа в систему), — следовательно, все процессы, относящиеся к сеансу входа в систему, относятся к одной группе.

Доступ к терминалу. Не поддерживает управление заданиями. Все процессы, имеющие открытый терминал, обладают одинаковыми правами доступа к нему независимо от того, являются ли они фоновыми или интерактивными. Вывод таких процессов может производиться на терминал вперемежку. Если несколько процессов попытаются прочесть данные с терминала одновременно, то может оказаться неочевидным, какой из них «перехватит» ту или иную строку ввода.

Сигналы терминала. Такие сигналы, как SIGQUIT или SIGINT, создаваемые при нажатии определенных клавиш клавиатуры, посылаются всем процессам управляющей группы терминала, то есть обычно всем процессам сеанса входа в систему. Однако такие сигналы в большинстве случаев необходимы только интерактивным процессам. Поэтому командный интерпретатор при создании фонового процесса устанавливает для него игнорирование таких сигналов. Он также перенаправляет стандартный ввод таких процессов в /dev/null, что не позволяет им читать данные с терминала через его дескриптор (чтобы начать чтение с терминала, процесс может открыть другие дескрипторы).

Отсоединение от терминала. Терминал отсоединяется от его управляющей группы при установке поля t_pgrp своей структуры tty в ноль. Такое происходит, если ни один из процессов не связан более с данным терминалом или когда лидер группы завершает работу (обычно процесс, порождаемый при в входе в систему с данного терминала).

Завершение работы лидера группы. Лидер группы становится управляющим процессом своего терминала и отвечает за управление терминалом для всей группы. Когда лидер завершает работу, его управляющий терминал отсоединяется от группы (поле t_pgrp сбрасывается в ноль). Более того, всем остальным процессам группы оправляется сигнал SIGHUP, и значения их полей p_pgrp также сбрасываются в ноль, после чего эти процессы[1] больше не относятся ни к одной группе (становясь тем самым *осиротевшими*).

Реализация. Поле p_pgrp структуры proc содержит идентификатор группы процессов. Область u имеет два поля, относящиеся к терминалу: u_ttyp (указатель на структуру tty управляющего терминала) и u_ttyd (номер устройства управляющего терминала). Поле t_pgrp структуры tty содержит идентификатор группы процессов, управляющей терминалом.

---

[1] Оставшиеся «в живых», так как сигнал SIGHUP по умолчанию приведет к завершению получившего его процесса. — *Прим. ред.*

4.9.3. Ограничения

Реализация механизма групп процессов в системе SVR3 обладает несколькими ограничениями [8]:

- не существует способов, которыми группа процессов могла бы закрыть свой управляющий терминал и выделить себе другой;
- хотя группы процессов формируются уже после открытия сеансов входа в систему, не существует возможности сохранять такие сеансы после отсоединения их управляющего терминала. В идеальном варианте в системе должен присутствовать механизм, позволяющий сохранять сеансы так, чтобы к ним позже можно было присоединить другой терминал и восстановить сеанс в том состоянии, в котором он был сохранен;
- нет какого-либо приемлемого способа обработки «потери несущей» управляющим терминалом. Такой терминал выделен группе и в случае «потери несущей» может быть перекоммутирован на другую группу с иной реализацией;
- ядро не синхронизирует доступ к терминалу между различными процессами группы. Фоновые и интерактивные процессы могут читать или вести запись на терминал в произвольном порядке;
- по завершении работы лидера группы ядро системы рассылает сигнал SIGHUP всем процессам группы. При этом процессы, игнорирующие этот сигнал, могут продолжить осуществлять доступ к терминалу даже в том случае, если он переназначен уже другой группе. Это может привести к тому, что пользователь системы будет наблюдать неожиданный для него вывод таких процессов или, что хуже, процесс станет считывать данные, введенные уже новым пользователем. Последнее может стать причиной возникновения «дыр» в защите системы;
- если какой-то процесс, отличный от процесса, осуществляющего вход в систему, вызовет setgrp, то он будет отсоединен от управляющего терминала. При этом процесс вправе продолжать осуществлять доступ к терминалу через какие-либо существующие дескрипторы файлов. Однако такой процесс уже не управляется терминалом и он не может получить сигнал SIGHUP;
- в рассматриваемой системе отсутствуют средства управления заданиями, такие как возможность переключения процессов с интерактивного на фоновый и обратно;
- такие программы, как эмуляторы терминала, открывающие устройства, отличные от своего управляющего терминала, не имеют возможности получать уведомление о потере несущей от этих устройств.

В операционной системе 4BSD были решены некоторые из перечисленных проблем. Модель, реализованная в этой ОС, будет представлена в следующем разделе.

4.9.4. Группы и терминалы в системе 4.3BSD

В системе 4.3BSD группа процессов представляет собой *задание* (иногда называемое *задачей*, task) в рамках сеанса входа в систему. Задание — это набор связанных процессов, которые управляются как единый блок относительно доступа к терминалу. Основные принципы доступа к терминалу системы 4.3BSD продемонстрированы на рис. 4.4.

Группы процессов. Процесс наследует идентификатор группы от своего родителя. Процесс может изменить свой идентификатор группы или идентификатор группы любого другого процесса при помощи вызова setgrp (последнее зависит от полномочий: у всех этих процессов должен быть либо общий владелец, либо вызывающий процесс должен иметь права суперпользователя). Системный вызов 4.3BSD setgrp имеет два параметра: идентификатор процесса и новый присваиваемый ему идентификатор группы. Таким образом, в системе 4.3BSD процесс может освободить место лидера или присоединиться к любой другой группе. Более того, группа процессов вообще может не иметь лидера.

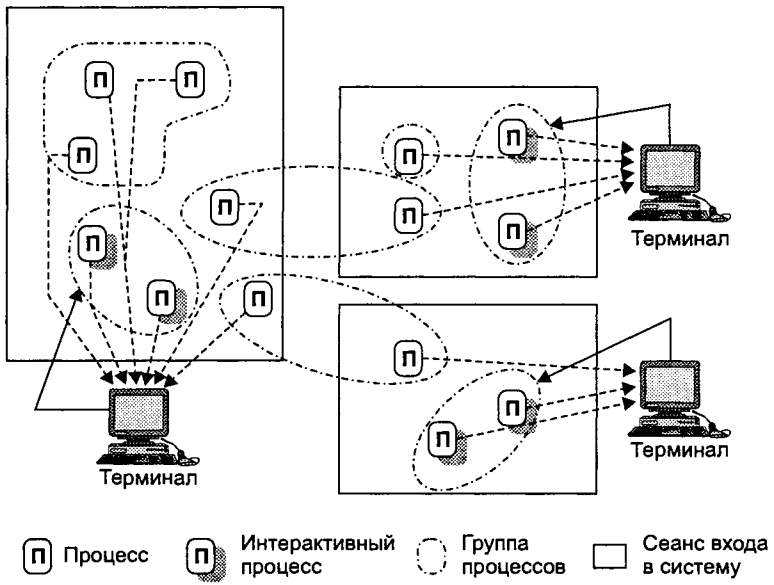


Рис. 4.4. Группы процессов в системе 4.3BSD UNIX

Задания. Командные интерпретаторы, поддерживающие управления заданиями, такие как csh, обычно создают новую группу процессов для каждой введенной командной строки, независимо от того, будут ли они выполняться в фоновом или текущем режиме. Таким образом, задание обыкновенно состоит из одного процесса или набора процессов, соединенных между

собой конвейерами[1]. Потомки этих процессов также будут являться членами группы.

Сеансы входа в систему. В системе 4.3BSD каждый сеанс входа в систему может создавать несколько групп процессов (или заданий), которые функционируют одновременно, разделяя между собой один и тот же терминал. Поле t_pgrp структуры tty терминала всегда содержит группу текущего выполняющегося задания.

Управляющие терминалы. Если процесс с идентификатором, группы равным нулю, открывает терминал, то такой терминал становится управляющим для данного процесса, а сам процесс присоединяется к текущей управляющей группе терминала (поле p_pgrp структуры proc процесса устанавливается равным полю t_pgrp структуры tty терминала). Если терминал в данный момент не является управляющим терминалом какой-либо группы, то процесс становится лидером (то есть поля p_pgrp структуры proc процесса и t_pgrp структуры tty терминала устанавливаются равными значению PID процесса). Прямые наследники init (то есть все командные интерпретаторы, установленные на вход в систему) первоначально обладают идентификатором группы, равным нулю. Установить идентификатор группы процесса в ноль может только суперпользователь.

Доступ к терминалу. Интерактивные процессы (то есть процессы, принадлежащие текущей управляющей группе терминала, полученной из поля t_pgrp структуры tty этого терминала) всегда обладают беспрепятственным доступом к терминалу. Если фоновый процесс попытается прочесть с терминала, драйвер пошлет сигнал SIGTTIN всем процессам, принадлежащим его группе. По умолчанию сигнал SIGTTIN приводит к приостановке работы получивших его процессов. Операция записи в терминал по умолчанию разрешена всем фоновым процессам. Система 4.3BSD предлагает настройку терминала (бит LTOSTOP, изменяемый при помощи вызова TIOCLSET ioctl), установка которой приводит к отправке сигнала SIGTTOU фоновому процессу, пытающемуся произвести запись в терминал. Задания, приостановленные сигналом SIGTTIN или SIGTTOU, могут продолжить работу после получения сигнала SIGCONT.

Управляющая группа. Процесс, имеющий доступ к терминалу на чтение, может осуществить вызов TIOSPGRP ioctl для изменения значения идентификатора управляющей группы терминала (поле t_pgrp структуры tty) на любое другое. Командный интерпретатор использует эту возможность системы для вызова процесса из фонового выполнения в интерактивный режим и наоборот. Например, пользователь может возобновить работу приостановленной

[1] В системе также существует возможность комбинирования двух или более не соединенных между собой процессов в одну группу при помощи ввода сразу нескольких команд в одной и той же строке. Такие команды заключаются в круглые скобки и отделяются друг от друга точкой с запятой: % (cc tanman.c; cp file1 file2; echo done >newfile).

группы процессов и назначить ее активной, сделав эту группу управляющей и отправив ей сигнал SIGCONT. Для этой цели в командных интерпретаторах csh и ksh предусмотрены команды fg и bg.

Закрытие терминала. Когда нет ни одного процесса, для которого данный терминал открыт, то такой терминал не связан с группой и его поле t_pgrp обнуляется. Это действие производится при помощи процедуры драйвера close, вызываемой в момент, когда последний дескриптор терминала закрывается.

Повторная инициализация линии терминала. ОС 4.3BSD обеспечивает системный вызов vhangup, который обычно используется процессом init для завершения текущего сеанса входа в систему и старта нового. Вызов просматривает таблицу открытых файлов, находит каждый элемент, относящийся к этому терминалу, и делает его неиспользуемым. Это достигается посредством удаления состояния «открыт» в элементах таблицы открытых файлов либо в тех реализациях, в которых поддерживается *интерфейс vnode* (см. раздел 8.6), изменением вектора vnodeops на такой набор функций, которые просто возвращают ошибку. Затем vhangup вызывает процедуру терминала close() и посылает сигнал SIGHUP управляющей группе этого терминала. Такой подход в ОС 4.3BSD является решением проблемы управления процессами, которые продолжают функционирование уже после завершения сеанса входа в систему.

4.9.5. Недостатки модели 4.3BSD

Несмотря на то что модель управления заданиями в 4.3 BSD является развитой и универсальной, она обладает рядом существенных недостатков, перечисленных ниже:

- не существует четкого представления сеанса входа в систему. Изначальный процесс входа в систему не является особенным и может даже не являться лидером группы. Обычно по завершении такого процесса сигнал SIGHUP не рассылается;
- не существует какого-то единственного процесса, ответственного за управление терминалом. Таким образом, состояние потери несущей передается всем процессам его текущей управляющей группы (при помощи сигнала SIGHUP), процессы которой могут даже игнорировать данный сигнал. Например, удаленный пользователь, работающий в системе через модемное соединение, останется в системе, даже если произойдет физическое отключение от линии связи;
- процесс может изменить управляющую группу терминала на любую другую, даже на несуществующую. Если позже будет создана группа с таким идентификатором, то она унаследует терминал и будет получать от него «незаслуженные» сигналы;
- программный интерфейс является несовместимым с интерфейсом System V.

Ясно, что нам необходим подход, при котором сохранялась бы концепция сеансов входа в систему и задач, выполняемых в таких сеансах. Последующий раздел посвящен описанию архитектуры сеансов операционной системы SVR4 и тому, как она решает эту проблему.

4.10. Архитектура сеансов в системе SVR4

Все ограничения моделей групп и терминалов в системах SVR3 и 4.3BSD могут быть отнесены к одной фундаментальной проблеме. Единое понятие группы процессов не в состоянии адекватно представлять сеансы входа в систему и задания, выполняемые в таких сеансах. В системе SVR3 неплохо реализовано управление поведением сеанса входа в систему, но она не поддерживает координацию заданий. Система 4.3BSD обладает развитыми средствами управления заданиями, но не умеет корректно изолировать друг от друга сеансы входа в систему.

В современных операционных системах, таких как SVR4 или 4.4BSD, эти проблемы были преодолены посредством представления сеансов и заданий раздельными, но взаимосвязанными между собой механизмами. Группа процессов определена единым заданием. Новый объект *сеанс* представляет собой сеанс входа в систему. В следующих разделах вы увидите описание архитектуры сеансов, представленной в ОС SVR4. Раздел 4.10.5 расскажет о реализации сеансов в 4.4BSD, которая схожа по функциональности с архитектурой, используемой в SVR4, но дополнительно обладает совместимостью со стандартами POSIX.

4.10.1. Задачи, поставленные перед разработчиками

Архитектура сеансов восполняет некоторые недостатки, имеющиеся в более ранних моделях. Основными целями новой архитектуры явились:

- поддержка на должном уровне как сеансов входа в систему, так и заданий;
- обеспечение управления заданиями, в стиле BSD;
- сохранение обратной совместимости с более ранними версиями System V;
- предоставление сеансам входа в систему возможности подключения и отключения нескольких управляющих терминалов в течение времени существования сеансов (конечно, в один момент времени они могут иметь только один управляющий терминал). Все эти изменения должны быть прозрачно распространены на все процессы сеанса;
- реализация лидера сеанса (процесса, создающего сеанс входа в систему), ответственного за обеспечение его целостности и безопасности;

- предоставление доступа к терминалу, основываясь исключительно на правах доступа к файлу. В частности, если процесс благополучно открыл терминал, то он может иметь к нему доступ на все то время, пока тот открыт;
- устранение несовместимости с предыдущими реализациями. Например, в ОС SVR3 существует определенная нестыковка в случае, если лидер группы порождает потомка при помощи fork до того, как назначит управляющий терминал. Такой процесс-потомок будет получать сигналы (SIGINT и т. д.) от этого терминала, но не будет иметь к нему доступа через /dev/tty.

4.10.2. Сеансы и группы процессов

Архитектура сеансов операционной системы SVR4 продемонстрирована на рис. 4.5 [12]. Каждый процесс относится как к определенному сеансу, так и к группе. Точно так же каждый управляющий терминал связан с сеансом и активной группой процессов (то есть управляющей группой терминала). Сеанс играет роль группы процессов системы SVR3, а лидер сесанса является ответственным за управление сеансом входа в систему и его изоляцию от других сеансов. Правом назначения или освобождения контролируемого терминала обладает только лидер сеанса.

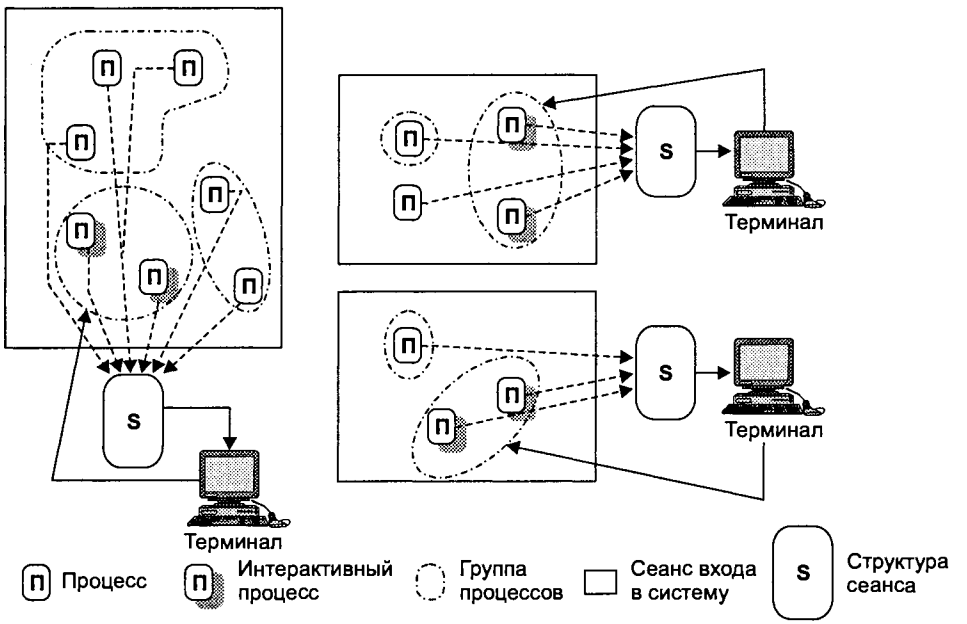


Рис. 4.5. Архитектура сеансов системы SVR4

Процесс создает новый сеанс при помощи вызова setsid, который устанавливает идентификаторы сеанса и группы в значение собственного PID. Таким образом, вызов setsid делает вызвавший его процесс одновременно лидером группы и сеанса. Если процесс уже является лидером группы, то он не может стать лидером сеанса, и выполнение вызова завершится с ошибкой.

Группы процессов системы SVR4 обладают основными чертами групп ОС 4.3BSD и обычно представляют собой задание внутри сеанса входа в систему. Таким образом, в одном сеансе входа в систему одновременно могут функционировать несколько групп процессов. Одна из таких групп является текущей и обладает неограниченным доступом к терминалу (то есть является управляющей группой терминала). Так же как и в системе 4.3BSD, фоновые процессы, пытающиеся получить доступ к управляющему терминалу, получат сигналы SIGTTIN и SIGTTOU (последний для этого должен быть разрешен, как это было описано в разделе 4.9.4).

Процесс наследует свою группу от родителя и может изменить ее при помощи вызовов setpgid или setgrp. Вызов setgrp идентичен варианту, представленному в SVR3, и устанавливает значение группы в PID сделавшего этот вызов процесса, делая его таким образом лидером группы. Вызов setpgid схож с setgrp системы 4.3BSD, но добавляет в действие некоторые существенные ограничения. Синтаксис setpgid таков:

```
setpgid (pid, pgid);
```

Задача этого вызова состоит в изменении группы процесса, указанного в pid, на значение, определенное в pgid. Если pgid равняется нулю, то идентификатор группы процесса устанавливается в значение, равное pid, что делает процесс лидером группы. Однако, как уже говорилось, в реализации setpgid существует несколько важных ограничений. Так, процесс, над которым производится действие, должен сам вызывать эту функцию, либо это должен делать один из его потомков, еще не сделавший exec. Вызывающий процесс и процесс, на который направлено действие, также должны относиться к одному и тому же сеансу. Если значение pgid не совпадает с PID целевого процесса (или равно нулю, что дает одинаковый эффект), то оно должно быть равным одному из существующих идентификаторов группы внутри того же сеанса.

По указанным выше причинам процессы имеют возможность перемещаться из одной группы в другую в течение продолжительности сеанса. Единственным способом покинуть сеанс для процесса является вызов setsid, открывающий новый сеанс с этим процессом как единственным ее членом. Процесс, являющийся лидером группы, может оставить свое лидерство, переместив себя в другую группу. Однако этот процесс не может создать новый сеанс до тех пор, пока его PID является идентификатором группы других процессов (то есть если группа, в которой он оставил лидерство, не пуста). Такой подход защищает от возникновения ситуации, когда группа процессов обладает тем же идентификатором, что и сеанс, в который эта группа не входит.

Точно так же текущая (управляющая) группа может быть изменена только процессом сеанса, который управляет терминалом, и только на группу, которая реально существует в сеансе. Эта возможность используется командными процессорами, поддерживающими управление заданиями для перевода заданий в интерактивный и фоновый режимы работы.

4.10.3. Структуры данных

На рис. 4.6 показаны структуры данных, используемые для управления сеансами и группами процессов. Вызов setsid выделяет новую структуру сеанса и сбрасывает поля p_sessp и p_pgidp структуры proc. Изначально сеанс не имеет управляющего терминала.

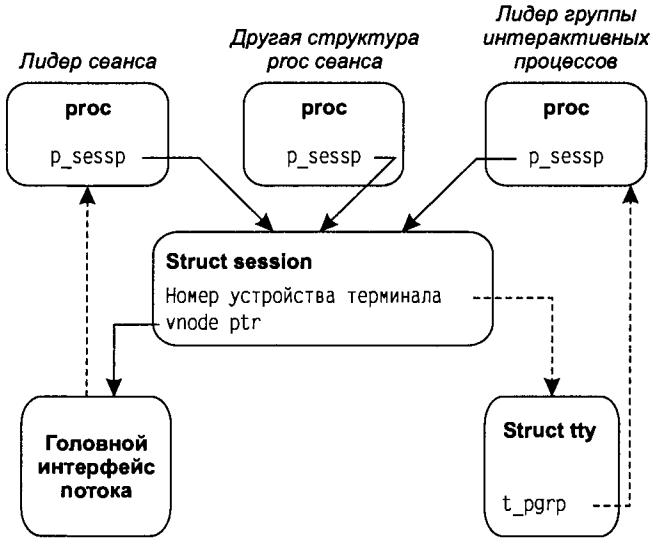


Рис. 4.6. Структуры данных управления сеансом в SVR4 UNIX

Когда лидер сеанса впервые открывает терминал (после того как он станет лидером), этот терминал станет управляющим терминалом сеанса, если только вызывающий процесс не передаст флаг O_NOCTTY в соответствующий вызов при открытии этого терминала. Структура сеанса инициализируется указателем на vnode этого терминала, а vnode, в свою очередь, указывает на головной интерфейс потока (stream head) для устройства.

Процессы-потомки лидера сеанса наследуют значение указателя в поле p_sessp структуры proc, благодаря чему могут постоянно следить за изменениями объекта сеанса. Поэтому процессы-потомки наследуют управляющий терминал, даже если тот был открыт уже после того, как эти процессы были созданы.

4.10.4. Управляющие терминалы

Файл /dev/tty снова выступает в качестве псевдонима управляющего терминала. Драйвер разрешает любые вызовы в /dev/tty посредством просмотра указателя на сеанс в структуре proc и, используя его, получает дескриптор vnode управляющего терминала. Если управляющий терминал освобождается, то ядро системы устанавливает указатель на vnode объекта сеанса в NULL, после чего все попытки доступа к /dev/tty закончатся неудачно. Если процесс открыл напрямую определенный терминал (в противоположность открытию /dev/tty), то он сможет продолжить доступ к нему даже после того, как терминал перестанет быть связанным с текущим сеансом входа в систему.

Когда пользователь входит в систему, процесс, обеспечивающий вход, производит следующие действия:

1. Вызывает setsid для того, чтобы стать лидером сеанса.
2. Закрывает stdin, stdout и stderr[1].
3. Вызывает open для открытия выбранного терминала. Так как этот терминал является первым из открытых лидером сеанса, то он становится управляющим терминалом для него. Вызов open возвращает дескриптор реального файла устройства терминала.
4. Дублирует и сохраняет полученный дескриптор, с тем чтобы не использовать stdin, stdout и stderr для ссылки на реальный файл устройства. После закрывает оригинальный дескриптор. Управляющий терминал остается открытым через дублированный дескриптор.
5. Открывает /dev/tty как stdin и дублирует его в stdout и stderr. Это действие эффективно открывает заново управляющий терминал через псевдоним устройства. Таким образом, лидер и все остальные процессы сеанса (которые наследуют эти дескрипторы) имеют доступ к терминалу только посредством /dev/tty (если только другой процесс прямо не откроет файл устройства терминала).
6. В конце происходит закрытие сохраненного дескриптора и удаление любых прямых контактов с управляющим терминалом.

Если драйвер терминала обнаружит разорванное соединение (например, потерю несущей при модемном подключении), то он пошлет сигнал SIGHUP только лидеру сеанса. Такой подход явно отличается от отправки сигнала текущей группе в системе 4.3BSD и оправки сигнала всем процессам управляющей группы (сеанса) в SVR3. При таком подходе лидер сеанса является доверенным процессом, и ожидается, что он произведет корректные действия при потере управляющего терминала.

Драйвер также посылает сигнал SIGSTP текущей группе процессов, если она не является группой лидера сеанса. Это защищает интерактивные процессы от получения неожиданных ошибок при попытке доступа к терминалу.

[1] Стандартные устройства ввода, вывода и вывода ошибок. — *Прим. ред.*

Управляющий терминал остается закрепленным за сеансом. Это дает возможность лидеру сеанса попытаться заново соединиться с терминалом после восстановления соединения.

Лидер сеанса может завершить соединение с текущим управляющим терминалом и открыть новый. Ядро системы установит указатель объекта vnode сеанса на указатель vnode нового терминала. В результате все процессы этого сеанса входа в систему будут прозрачно для них переключены на новый управляющий терминал. Такая косвенная связь, обеспеченная /dev/tty, облегчает решение задачи распространения такого изменения.

Когда лидер сеанса заканчивает свою работу, он завершает и сеанс входа в систему. Управляющий терминал освобождается при помощи установки указателя vnode сеанса в NULL. В результате ни один процесс этого сеанса не сможет иметь доступа к терминалу посредством /dev/tty (но они смогут продолжать осуществлять доступ при непосредственном открытии файла устройства терминала). Процессы текущей группы получают сигнал SIGHUP. Все прямые потомки существующего процесса будут унаследованы процессом init.

4.10.5. Реализация сеансов в 4.4BSD

В архитектуре сеансов системы SVR4 адекватно представлены как сеанс входа в систему, так и задания, выполняющиеся в этом сеансе. В то же время она совместима со стандартом POSIX 1003.1 и ранними версиями System V. Реализация сеансов в операционных системах 4.4BSD и OSF/1 очень похожа на архитектуру SVR4 и обладает сравнимыми с ней возможностями. Различия между реализациями проявляются только в отдельных деталях.

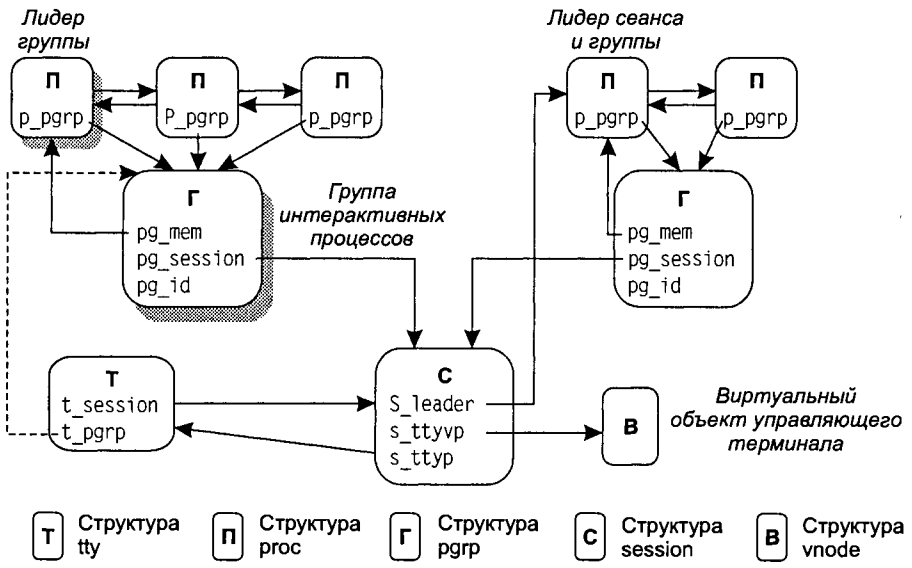


Рис. 4.7. Структуры данных управления сеансами в 4.4BSD UNIX

На рис. 4.7 для сравнения проиллюстрированы структуры данных, используемые в 4.4BSD [10]. Одним из важнейших отличий является то, что структура proc не имеет прямой ссылки на объект session. Вместо этого она ссылается на объект группы процессов (структура pgrp), который, в свою очередь, содержит указатель на структуру session.

4.11. Заключение

Появление стандарта POSIX 1003.1 помогло объединить различающиеся между собой и несовместимые реализации поддержки сигналов и управления терминалами. В результате интерфейсы оказались весьма удачными и в большой степени отвечающими ожиданиям типичных приложений и разработчиков.

4.12. Упражнения

Ответы на некоторые из перечисленных ниже вопросов могут быть различными в зависимости от используемого варианта UNIX. Отвечающий может выбрать для ответа на вопросы одну из наиболее близких для него реализаций системы.

1. Почему обработчики сигналов не сохраняются после выполнения системного вызова exec?
2. Почему сигнал SIGCHLD по умолчанию игнорируется?
3. Что происходит в случае возникновения сигнала для процесса, находящегося в стадии выполнения вызовов fork, exec или exit?
4. При каких условиях сигнал, отправленный kill, не завершит немедленно выполнение процесса?
5. В традиционных системах UNIX приоритет сна используется для двух целей: решения, будет ли процесс разбужен для приема этого сигнала, и определения приоритета процесса в расписании после выхода процесса из режима сна. Какие недостатки имеются в классической модели и как эти проблемы были решены в современных системах?
6. Какие существуют недостатки использования постоянно установленных обработчиков? Существуют ли какие-либо определенные сигналы, для обработки которых нецелесообразно применять постоянно установленные обработчики?
7. Чем отличаются между собой реализации вызова sigpause в системах 4.3BSD и SVR3? Опишите возможную ситуацию, при которой применение первого варианта представляется более удобным.
8. Чем предпочтительнее поддержка повторного старта прерванного системного вызова на уровне ядра, нежели перезапуск его приложением?

9. Что произойдет, если процесс получит несколько экземпляров одного и того же сигнала перед тем, как он обработает первый экземпляр этого сигнала? Будет ли в такой ситуации целесообразным применение иных сигнальных моделей?
10. Представьте, что процесс ожидает получения двух сигналов и что в нем были описаны обработчики для каждого из этих сигналов. Каким образом ядро системы может убедиться в том, что процесс обработает второй сигнал сразу же после обработки первого?
11. Что произойдет, если процесс получит сигнал во время обработки другого? Как в этом случае процесс может контролировать свои действия?
12. В каких случаях процессу целесообразно использовать флаг SA_NOCLDWAIT, определенный в системе SVR4? В каких случаях этого делать не следует?
13. Зачем обработчику исключения необходим полный контекст процесса, в котором исключение произошло?
14. Какой процесс может создать новую группу процессов: а) в системе 4.3BSD, б) в системе SVR4?
15. Какими преимуществами обладает архитектура сеансов SVR4 по сравнению со средствами работы с терминалами и управления заданиями системы 4.3BSD?
16. В [3] описывается менеджер сеансов прикладного уровня, использующийся для поддержки сеансов входа в систему. Насколько сходно это средство с архитектурой сеансов системы SVR4?
17. Что должно сделать ядро системы SVR4 в том случае, когда лидер сеанса освободит управляющий терминал?
18. Каким образом в системе SVR4 реализована поддержка переподключения сеанса к управляющему терминалу? В каких ситуациях применима такая возможность системы?

4.13. Дополнительная литература

1. American Telephone & Telegraph, «UNIX System V Release 3: Programmer's Reference Manual», 1986.
2. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
3. Bellovin, S. M., «The Session Tty Manager», Proceedings of the Summer 1988 USENIX Technical Conference, Jun. 1988.
4. Black, D. L., Golub, D. B., Hauth, K., Tevanian, A., and Sanzi, R., «The Mach Exception Handling Facility», CMU-CS-88-129, Computer Science Department, Carnegie Mellon University, Apr. 1988.

5. Institute for Electrical and Electronic Engineers, Information Technology – «Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) (C Language)», 1003.1–1990, Dec. 1990.
6. Joy, W., «An Introduction to the C Shell», Computer Science Division, University of California at Berkeley, Nov. 1980.
7. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3 BSD UNIX Operating System», Addison-Wesley, Reading, MA, 1989.
8. Leaner, D. C., «A System V Compatible Implementation of 4.2 BSD Job Control», Proceedings of the Summer 1986 USENIX Technical Conference, Jun. 1986, pp. 459–474.
9. Stevens, W. R., «UNIX Network Programming», Prentice-Hall, Englewood Cliffs, NJ, 1990.
10. Stevens, W. R., «Advanced Programming in the UNIX Environment», Addison-Wesley, Reading, MA, 1992.
11. UNIX Systems Laboratories, «Operating System API Reference: UNIX SVR4.2», UNIX Press, 1992.
12. Williams, T., «Session Management in System V Release 4», Proceedings of the Winter 1989 USENIX Technical Conference, Jan. 1989, pp. 365–375.

Глава 5
Планирование процессов

5.1. Введение

Центральный процессор наряду с памятью и терминалами является общим ресурсом, разделяемым между всеми процессами системы. Принятие решения о том, как должны использоваться ресурсы различными процессами, возложено на операционную систему. Планировщик является компонентом ОС, определяющим, какой из процессов должен выполняться в данный момент времени и как долго он может занимать процессор. Система UNIX изначально создавалась как ОС разделения времени, что означает возможность одновременного выполнения в ней нескольких процессов. Конечно, в определенной степени одновременное выполнение является лишь иллюзией (по крайней мере, в однопроцессорных системах), так как в один момент времени на одном процессоре может выполняться только что-то одно. В системах UNIX эмулируется одновременность работы при помощи чередования процессов на основе принципа разделения времени. Планировщик предоставляет процессор каждому процессу системы на небольшой период времени, после чего производит переключение на следующий процесс. Такой период называется *квантом времени* (quantum или slice).

В описании планировщика UNIX необходимо учитывать два важных аспекта. Первый из них касается *политики* — правил, используемых при принятии решения о том, какой из процессов следует назначить на выполнение и когда произвести переключение на выполнение другого процесса. Второй рассматриваемый аспект относится к *реализации*, представляющей собой набор структур данных и алгоритмов, используемых для проведения этих политик. Политика планирования должна по мере возможности удовлетворять различным требованиям: определенному времени реакции для интерактивных приложений, высокой производительности для фоновых заданий, недопущению полного необслуживания процессов и т. д. Попытка удовлетворения одновременно всех поставленных целей часто приводит к конфликтам, поэтому планировщик должен обеспечить наилучшее соотношение между ними. Также от него требуется эффективная реализация политики планирования при минимальных перегрузках системы.

На самом низком уровне планировщик заставляет процессор производить переключения от одного процесса к другому. Это действие называется *переключением контекста.* Ядро системы сохраняет аппаратный контекст текущего выполняющегося процесса в *блоке управления процессом* (process control block, PCB), который обычно является частью его области u. Контекст представляет собой «моментальный снимок» текущих значений регистров общего назначения, регистров управления памятью и других специальных регистров процесса. После завершения процедуры сохранения ядро системы загружает аппаратные регистры с контекстом следующего процесса, готовящегося к выполнению (контекст загружается из его блока PCB). Это действие приводит к тому, что CPU начинает выполнение нового процесса. Основной задачей планировщика является принятие решения о том, когда произвести переключение контекста и какой из процессов назначить на выполнение.

Переключения контекста являются весьма затратными операциями. Кроме сохранения копии регистров процесса, ядро системы должно выполнить еще множество архитектурно зависимых действий. На некоторых системах оно должно очистить данные, инструкции или буфер трансляции адресов для предупреждения некорректного доступа к памяти новым процессом (см. разделы 15.9–15.13). В результате в начале своей работы процесс тратит в виде накладных расходов несколько операций обращения к памяти. Это уменьшает его производительность, так как доступ к памяти является значительно более медленной процедурой, чем доступ к кэшу. Также стоит упомянуть о конвейерных архитектурах, таких как *процессоры с сокращенным набором команд* (Reduced Instruction Set Computers, RISC), в которых ядро перед переключением контекста должно очистить конвейер команд. Упомянутые факторы могут влиять не только на реализацию планировщика, но и на его политику.

Эта глава начинается описанием обработки прерываний таймера и заданий, основанных на его работе. Таймер является важным компонентом функционирования планировщика, так как последнему часто необходимо вытеснять выполняющийся процесс по окончанию выделенного ему кванта времени. Остальная часть главы рассказывает об устройстве различных планировщиков и о том, как их функционирование влияет на работу системы.

5.2. Обработка прерываний таймера

На каждой UNIX-машине существует аппаратный таймер, который вырабатывает прерывание в системе через фиксированные промежутки времени. В некоторых машинах операционная система должна увеличивать значение времени, отсчитываемого каждым прерыванием таймера, в других таймер это делает самостоятельно. Период времени между двумя последующими преры-

ваниями таймера называется *тиком процессора*, *тиком таймера* или просто *тиком*. Большинство компьютеров поддерживают переменные тиковые интервалы. В системах UNIX продолжительность тика составляет обычно 10 миллисекунд[1]. Во многих реализациях UNIX частота таймера (количество тиков в секунду) хранится в специальной константе HZ, которая обычно определена в файле param.h. Для тика продолжительностью 10 миллисекунд значение HZ будет равно 100. Функции ядра чаще всего измеряют время в количестве тиков, редко используя для этого секунды или миллисекунды.

Обработка прерываний сильно зависит от используемой системы. Этот раздел рассказывает о стандартной реализации, которую можно встретить во многих традиционных версиях UNIX. Обработчик прерываний таймера запускается в ответ на возникновение аппаратного прерывания таймера, являющегося вторым по приоритету событием в системе (после прерывания по сбою питания). Следовательно, обработчик должен запускаться как можно быстрее, а его время работы желательно сводить к минимуму. Обработчик прерываний таймера выполняет следующие задачи:

- ведет счет тиков аппаратного таймера по необходимости;
- обновляет статистику использования процессора текущим процессом;
- выполняет функции, относящиеся к работе планировщика, такие как пересчет приоритетов и действия, выполняющиеся по истечении выделенного кванта времени;
- посылает текущему процессу сигнал SIGXCPU, если тот превысил выделенную ему квоту использования процессора;
- обновляет часы и другие таймеры системы. Например, в SVR4 имеется переменная lbolt, хранящая количество тиков, отсчитанных с момента загрузки системы;
- обрабатывает отложенные вызовы (см. раздел 5.2.1);
- пробуждает в нужные моменты системные процессы, такие как swapper и pagedaemon;
- обрабатывает сигналы тревоги (см. раздел 5.2.2).

Некоторые из перечисленных задач не требуют выполнения на каждом тике. В большинстве систем UNIX определено понятие *основного тика*, который равен *n* тикам таймера (число *n* зависит от конкретного варианта системы). Планировщик выполняет некоторые из своих задач только с приходом основного тика. Например, в 4.3BSD пересчет приоритетов происходит на каждый четвертый тик, в то время как SVR4 обрабатывает сигналы тревоги и возобновляет по необходимости работу системных процессов с частотой один раз в секунду.

---

[1] Это число не является универсальным и зависит от конкретного варианта UNIX. Продолжительность тика также зависит от разрешения аппаратного таймера.

5.2.1. Отложенные вызовы

Отложенный вызов (callout) представляет собой запись функции, которую ядро системы должно будет вызвать через определенный промежуток времени. Например, в системе SVR4 любая подсистема ядра может зарегистрировать отложенный вызов следующим образом:

```
int to_ID = timeout (void (*fn)(), caddr_t arg, long delta);
```

где fn() — функция ядра, которую необходимо запустить, arg — аргумент, который следует передать fn(). delta — временной интервал, через который эта функция должна быть вызвана, выраженный в тиках процессора. Ядро выполняет функцию, определенную в отложенном вызове, в системном контексте. Следовательно, такая функция не должна переходить в режим ожидания или осуществлять доступ к контексту процесса. Возвращаемое значение to_ID может быть использовано для отмены выполнения отложенного вызова:

```
void untimeout (int to_ID);
```

Отложенные вызовы могут быть использованы для выполнения различных повторяющихся задач, таких как:

- повторная пересылка сетевых пакетов;
- некоторые функции планировщика и управления памятью;
- мониторинг устройств для предотвращения потери прерываний;
- периодический опрос устройств, не поддерживающих прерывания.

Отложенные вызовы считаются обычными процедурами ядра и не должны выполняться с приоритетами прерываний. Поэтому обработчик прерываний таймера не выполняет эти вызовы напрямую. На каждом тике обработчик прерываний таймера проверяет, не нужно ли начать выполнение отложенного вызова. Если он находит ожидающий вызов, то выставляет флаг, указывающий на необходимость запуска *обработчика отложенного вызова*. Система проверяет этот флаг при возврате в основной приоритет прерываний и, если тот установлен, запускает обработчик. Обработчик начнет выполнение каждого ожидаемого отложенного вызова. Следовательно, ожидаемый отложенный вызов будет выполнен с максимально возможной быстротой, но только после обработки всех ожидающих прерываний[1].

Ядро системы поддерживает список ожидающих отложенных вызовов. Организация такого списка влияет на производительность системы, так как он может содержать несколько вызовов. Поскольку список проверяется на каждом тике с высоким приоритетом прерывания, проверяющий алгоритм должен оптимизировать время проверки. Время, затрачиваемое на вставку

[1] Во многих реализациях UNIX обеспечена некоторая оптимизация механизма обработки прерываний для случая, когда никакие иные прерывания не находятся в ожидании завершения работы наиболее приоритетного обработчика. В такой ситуации обработчик таймера напрямую уменьшает приоритет прерывания и запускает обработчик отложенного вызова.

нового отложенного вызова в список, является менее критичным, так как вставка обычно происходит при более низких приоритетах и с меньшей частотой, чем одна операция на каждый тик.

Существует несколько способов реализации списка отложенных вызовов. Метод, используемый в 4.3BSD [12], сортирует список в порядке *возрастания времени запуска вызовов* (time to fire). Каждый элемент списка содержит разницу между временем своего запуска и запуска предыдущего отложенного вызова. Ядро системы уменьшает время первого элемента списка на каждом тике и запускает вызов после того, как это значение станет равным нулю. Если другие вызовы должны выполниться в это же время, то и они запускаются ядром. Описанная технология графически проиллюстрирована на рис. 5.1.

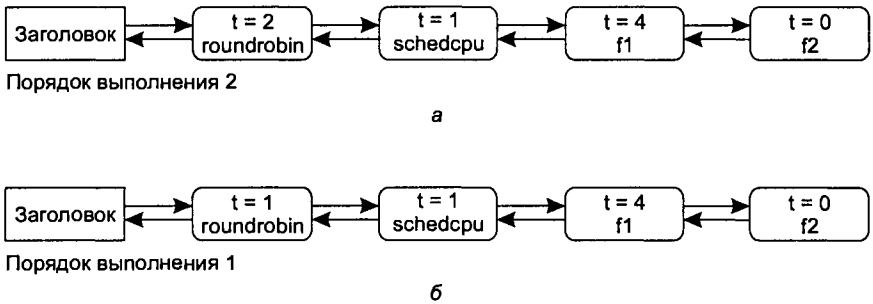


Рис. 5.1. Реализация отложенных вызовов в BSD UNIX: *а* — начальное состояние списка отложенных вызовов; *б* — состояние через один тик таймера

Еще одним применяемым методом является использование сходного сортированного списка, но хранится в нем абсолютное время запуска для каждого элемента. В таком случае на каждом тике ядро системы сравнивает текущее абсолютное время со временем первого элемента списка и запускает отложенный вызов, если значения совпадут.

Оба приведенных метода требуют организации сортированного списка, который может оказаться довольно большим и, следовательно, довольно затратным для системы. Альтернативное решение предполагает использование *карусели* (timing wheel), представляющей собой циклически замкнутый массив очередей отложенных вызовов. На каждом тике обработчик прерываний таймера смещает указатель к следующему элементу массива, циклически возвращаясь по достижении конца массива. Если в текущей очереди находятся отложенные вызовы, то происходит проверка их времени запуска. Новые вызовы добавляются в очередь, отстоящую на N элементов от текущей, где N – время до запуска отложенного вызова, выраженное в тиках[1].

[1] Скорее всего, N — это не просто смещение от текущего элемента, а остаток от деления времени запуска на длину массива; например, при длине массива в 100 элементов и времени запуска через 535 тиков N будет равняться 35. — *Прим. ред.*

В результате циклического планирования хэшируются отложенные вызовы на основе времени истечения их ожидания, то есть времени, через которое они должны отработать. Внутри каждой очереди отложенные вызовы могут храниться как в упорядоченном, так и в не сортированном виде. Сортировка сокращает время на обработку непустых очередей, но увеличивает затраты на внесение нового элемента в список. В работе [18] описываются способы улучшения продуктивности этого метода путем использования нескольких иерархически связанных временных колес с целью оптимизации производительности таймера.

5.2.2. Будильники

Процесс может запросить ядро системы послать ему сигнал через определенный промежуток времени, что похоже на работу обычного будильника. Существуют три типа *будильников*: реального времени, виртуального времени и профиля процесса. *Будильник реального времени* работает в действительном времени, по истечении заданного промежутка которого ядро посылает процессу сигнал SIGALRM. *Будильник профиля процесса* измеряет время работы процесса и использует сигнал SIGPROF. *Будильник виртуального времени* следит только за количеством времени работы процесса в режиме задачи и посылает сигнал SIGVTALRM.

В BSD UNIX существует системный вызов setitimer, который позволяет процессу запросить будильник любого типа, указывая интервал времени в микросекундах. Внутри системы этот интервал преобразуется в соответствующее ему количество тиков, так как именно тик является минимальной временной единицей, различаемой ядром. В System V для обращения к будильнику реального времени применяется системный вызов alarm. Задаваемый ему интервал времени должен выражаться в секундах. В систему SVR4 был добавлен системный вызов hrtsys, который обеспечивает таймерный интерфейс с высоким разрешением, позволяющим задавать временной интервал в микросекундах. Этот вызов позволяет реализовать совместимость с BSD посредством setitimer (а также getitimer, gettimeofday и settimeofday) в виде библиотечных функций.

Высокое разрешение будильников реального времени не предполагает высокой точности их работы. Предположим, что приложение использует будильник реального времени для воспроизведения звука по истечении 60 миллисекунд с момента запроса. После того как пройдет указанный промежуток времени, ядро пошлет сигнал SIGALRM вызывающему процессу. Однако процесс не сможет увидеть (а следовательно, и среагировать) на сигнал до тех пор, пока в очередной раз не будет назначен планировщиком на выполнение. Таким образом, перед началом обработки сигнала неизбежно возникает задержка, величина которой зависит от приоритета процесса и степени загруженности системы. Таймеры высокого разрешения полезны при использовании их

высокоприоритетными процессами, обладающими минимальными задержками при планировании. Но даже и эти процессы могут быть задержаны, если текущий процесс, выполняясь в режиме ядра, еще не достиг точки, с которой он может быть вытеснен. Более подробно описание таймеров реального времени см. в разделе 5.5.4.

Виртуальные будильники и будильники профиля не имеют вышеизложенной проблемы, так как они не привязаны к реальному времени. Точность этих будильников определяется другими факторами. Обработчик прерываний таймера измеряет число полных тиков работы текущего процесса, даже в том случае, если было использовано не целое количество тиков. Таким образом, измеряемый промежуток времени отражает количество прерываний таймера, произошедших в течение выполнения процесса. При большой продолжительности работы эта величина является достаточно точным индикатором. Однако для любого единичного будильника результат будет весьма неточен.

5.3. Цели, стоящие перед планировщиком

Планировщик должен по возможности наиболее справедливо распределять процессорное время между всеми процессами системы. Естественно также и то, что при увеличении общей загруженности системы каждый процесс получает меньшее количество процессорного времени и, следовательно, работает медленнее, чем на незагруженной системе. Планировщик должен следить за тем, чтобы система предоставляла приемлемую производительность каждому приложению при общей занятости системы в рамках нормы.

Обычно операционная система предоставляет возможность одновременного выполнения нескольких приложений. Эти приложения можно весьма фривольно категоризировать по следующим классам, в зависимости от их требований к планированию и производительности работы:

- **Интерактивные приложения.** Приложения типа командных интерпретаторов, редакторов и программ с графическим пользовательским интерфейсом, постоянно взаимодействующих с пользователями. Большую часть времени такие приложения находятся в ожидании действий пользователя, таких как ввод с клавиатуры, либо манипуляций мышью. После получения ввода приложение должно быстро его обработать, иначе пользователь будет наблюдать «торможение» системы. Следовательно, системе необходимо уменьшать среднее время между действием пользователя и реакцией приложения на него, так, чтобы пользователь не заметил задержки. Приемлемая величина задержек реакции на ввод с клавиатуры или движения мыши составляет 50–150 миллисекунд.
- **Пакетные приложения.** К ним относятся такие действия, как сборка программ или научные вычисления, то есть программы, не требующие взаимодействия с пользователем и часто выполняющиеся в фоновом

режиме. Для таких задач мерой эффективности планирования является время завершения их работы при функционировании среди других процессов, сравниваемое со временем их выполнения как единственной задачи системы.

- **Приложения реального времени.** Это достаточно широкий класс приложений, для которых время часто является весьма критичным. Хотя на сегодняшний день существует множество различных видов приложений реального времени, обладающих отличными друг от друга наборами требований, между ними существует множество сходств. Обычно все они требуют упреждающего поведения планировщика, с гарантированными границами времени реакции. Например, приложение работы с видео может выводить фиксированное количество *кадров в секунду* (frames per second, fps). Такое приложение может больше заботиться об уменьшении времени реакции системы, нежели просто запрашивать дополнительное процессорное время. Пользователи могут настраивать частоту кадров в диапазоне 10–30 fps со средней частотой в 20 fps.

На рабочей станции одновременно способны выполняться сразу несколько типов приложений. Планировщик должен пытаться сбалансировать требования каждого из них. Также он обязан следить за тем, чтобы функции ядра, такие как поддержка страничной памяти, обработка прерываний и управление процессами, могли работать именно тогда, когда это требуется.

В хорошо сбалансированной системе все приложения должны продолжать выполнение. Ни одно приложение не должно влиять на продолжение выполнения остальных, кроме случая, когда пользователь принудительно укажет на это. Более того, система обязана постоянно находиться в состоянии получения и обработки интерактивного пользовательского ввода, в противном случае пользователь не будет иметь возможности управлять системой.

Выбор политики планирования оказывает важное влияние на способность системы удовлетворять требованиям различных типов приложений. Следующий раздел описывает традиционный вариант планировщика (SVR3/4.3BSD), который поддерживает только интерактивные и пакетные задания. Остальная часть пятой главы посвящена планировщикам современных систем UNIX, поддерживающих приложения реального времени.

5.4. Планирование в традиционных системах UNIX

В этом разделе обсуждается традиционный алгоритм планирования, применяемый как в SVR3, так и в 4.3BSD. Эти ОС создавались как системы разделения времени, обладающие интерактивными средами для нескольких пользователей, которые могли одновременно запускать несколько пакетных

и интерактивных процессов. Цель политики планирования заключается в увеличении скорости реакции при интерактивном взаимодействии пользователя с системой, одновременно отслеживая протекание фоновых задач, защищая их от зависания из-за недостатка процессорного времени.

Механизм планирования в традиционных системах базируется на приоритетах. Каждый процесс обладает приоритетом планирования, изменяющимся с течением времени. Планировщик всегда выбирает процессы, обладающие более высоким приоритетом. Для диспетчеризации процессов с равным приоритетом применяется вытесняющее квантование времени. Изменение приоритетов процессов происходит динамически на основе количества используемого ими процессорного времени. Если какой-либо из высокоприоритетных процессов будет готов к выполнению, планировщик вытеснит ради него текущий процесс даже в том случае, если тот не израсходовал свой *квант времени*.

Традиционное ядро UNIX является строго невытесняющим. Если процесс выполняется в режиме ядра (например, в течение исполнения системного вызова или прерывания), то ядро не заставит такой процесс уступить процессор какому-либо высокоприоритетному процессу. Выполняющийся процесс может только добровольно освободить процессор в случае своего блокирования в ожидании ресурса, иначе он может быть вытеснен при переходе в режим задачи. Реализация ядра невытесняющим решает множество проблем синхронизации, связанных с доступом нескольких процессов к одним и тем же структурам данных ядра (см. раздел 2.5).

Следующие подразделы посвящены описанию устройства и реализации планировщика в системе 4.3BSD. Вариант, применяемый в SVR3, имеет лишь незначительные отличия во второстепенных деталях, таких как имена некоторых функций и переменных.

5.4.1. Приоритеты процессов

Приоритет процесса задается любым целым числом, лежащим в диапазоне от 0 до 127. Чем меньше такое число, тем выше приоритет. Приоритеты от 0 до 49 зарезервированы для ядра, следовательно, прикладные процессы могут обладать приоритетом в диапазоне 50–127. Структура proc содержит следующие поля, относящиеся к приоритетам:

p_pri	Текущий приоритет планирования
p_usrpri	Приоритет режима задачи
p_cpu	Результат последнего измерения использования процессора
p_nice	Фактор «любезности», устанавливаемый пользователем

Поля p_pri и p_usrpri применяются для различных целей. Планировщик использует p_pri для принятия решения о том, какой процесс направить на вы-

полнение. Когда процесс находится в режиме задачи, значение его p_pri идентично p_usrpri. Когда процесс просыпается после блокирования в системном вызове, его приоритет будет временно повышен для того, чтобы дать ему предпочтение для выполнения в режиме ядра. Следовательно, планировщик использует p_usrpri для хранения приоритета, который будет назначен процессу при возврате в режим задачи, а p_pri — для хранения временного приоритета для выполнения в режиме ядра.

Ядро системы связывает *приоритет сна* с событием или ожидаемым ресурсом, из-за которого процесс может заблокироваться. Приоритет сна является величиной, определяемой для ядра, и потому лежит в диапазоне 0–49. Например, значение приоритета сна для терминального ввода равно 28, в то время как для операций дискового ввода-вывода оно имеет значение 20. Когда замороженный процесс просыпается, ядро устанавливает значение его p_pri, равное приоритету сна события или ресурса[1]. Поскольку приоритеты ядра выше, чем приоритеты режима задачи, такие процессы будут назначены на выполнение раньше, чем другие, функционирующие в режиме задачи. Такой подход позволяет системным вызовам быстро завершать свою работу, что является желательным, так как процессы во время выполнения вызова могут занимать некоторые ключевые ресурсы системы, не позволяя пользоваться ими другим.

Когда процесс завершил выполнение системного вызова и находится в состоянии возврата в режим задачи, его приоритет сбрасывается обратно в значение текущего приоритета в режиме задачи. Измененный таким образом приоритет может оказаться ниже, чем приоритет какого-либо иного запущенного процесса; в этом случае ядро системы произведет переключение контекста.

Приоритет в режиме задачи зависит от двух факторов: «любезности» (nice) и последней измеренной величины использования процессора. *Степень любезности* (nice value) является числом в диапазоне от 0 до 39 со значением 20 по умолчанию. Увеличение значения приводит к уменьшению приоритета. Фоновым процессам автоматически задаются более высокие значения степени благоприятствия. Уменьшить эту величину для какого-либо процесса может только суперпользователь, поскольку при этом поднимется его приоритет. Степень любезности называется так потому, что одни пользователи могут быть поставлены в более выгодные условия другими пользователями посредством увеличения кем-либо из последних значения уровня любезности для своих менее важных процессов[2].

Системы разделения времени пытаются выделить процессорное время таким образом, чтобы конкурирующие процессы получили его примерно в рав-

---

[1] На котором он был заблокирован. — *Прим. ред.*

[2] Команда nice(1) принимает любые значения в диапазоне от –20 до 19 (отрицательные величины может задавать только суперпользователь). Это число используется как увеличение текущего значения фактора любезности.

ных количествах. Такой подход требует слежения за использованием процессора каждым из процессов. Поле p_cpu структуры proc содержит величину результата последнего сделанного измерения использования процессора процессом. При создании процесса значение этого поля инициализируется нулем. На каждом тике обработчик таймера увеличивает p_cpu на единицу для текущего процесса до максимального значения, равного 127. Более того, каждую секунду ядро системы вызывает процедуру schedcpu() (запускаемую через отложенный вызов), которая уменьшает значение p_cpu каждого процесса исходя из *фактора «полураспада»* (decay factor). В системе SVR3 используется фиксированное значение этого фактора, равное ½. В 4.3BSD для расчета фактора полураспада применяется следующая формула:

```
decay = (2*load_average)/(2*load_average + 1);
```

где load_average — это среднее количество процессов, находящихся в состоянии готовности к выполнению, за последнюю секунду. Процедура schedcpu() также пересчитывает приоритеты для режима задачи всех процессов по формуле

```
p_usrpri = PUSER + (p_cpu/4) + (2*p_nice);
```

где PUSER — базовый приоритет в режиме задачи, равный 50.

В результате, если процесс в последний раз[1] использовал большое количество процессорного времени, его p_cpu будет увеличен. Это приведет к росту значения p_usrpri и, следовательно, к понижению приоритета. Чем дольше процесс простаивает в очереди на выполнение, тем больше фактор полураспада уменьшает его p_cpu, что приводит к повышению его приоритета. Такая схема предотвращает зависание низкоприоритетных процессов по вине операционной системы. Ее применение предпочтительнее процессам, осуществляющим много операций ввода-вывода, в противоположность процессам, производящим много вычислений. Если процесс большинство времени выполнения тратит на ожидание ввода-вывода (например, командный интерпретатор или текстовый редактор), то он остается с высоким приоритетом и, таким образом, быстро получает процессор при необходимости. Вычислительные приложения, такие как компиляторы или редакторы связей, обычно обладают более высокими значениями p_cpu и работают на значительно более низких приоритетах.

Фактор использования процессора обеспечивает справедливость и равенство при планировании процессов в режиме разделения времени. Основная идея заключается в хранении приоритетов всех таких процессов примерно в том же диапазоне в течение некоторого периода времени. Приоритеты могут повышаться или понижаться в рамках этого диапазона в зависимости от того, сколько процессорного времени эти процессы получали в последний

[1] До вытеснения другим процессом. — *Прим. ред.*

раз. Если приоритеты будут меняться слишком медленно, процессы, начавшие работу с низким приоритетами, сохранят их в течение долгого периода времени, что приведет к их фактическому зависанию.

Фактор полураспада обеспечивает экспоненциально взвешенное среднее значение использования процессора в течение всего периода функционирования процесса. Формула, применяемая в системе SVR3, подсчитывает простое экспоненциальное среднее, что имеет побочный эффект, заключающийся в росте приоритетов при увеличении загрузки системы [2]. Рост происходит по причине того, что на сильно загруженной системе каждый процесс получает меньшее процессорное время. При этом величина использования процессора процессом остается низкой, поэтому фактор полураспада со временем еще дополнительно ее сокращает. В результате использование процессора не сильно влияет на приоритеты, и процессы, начавшие работу с более низкими приоритетами, простаивают в ожидании процессора непропорционально.

В системе 4.3BSD фактор полураспада зависит от загруженности системы. Если загрузка велика, он влияет несущественно. Следовательно, для процессов, получающих процессорное время, снижение приоритетов будет происходить быстро.

5.4.2. Реализация планировщика

Планировщик содержит массив qs, состоящий из 32 очередей выполнения (рис. 5.2). Каждая очередь соответствует четырем соседствующим приоритетам. Таким образом, очередь 0 используется для приоритетов 0–3, очередь 1 для приоритетов 4–7 и т. д. Каждая очередь содержит начало двунаправленного связанного списка структур proc. Глобальная переменная whichqs хранит битовую маску, в которой для каждой очереди зарезервирован один бит. Бит устанавливается, если в очереди имеется хотя бы один процесс. В очередях планировщика находятся только готовые к выполнению процессы.

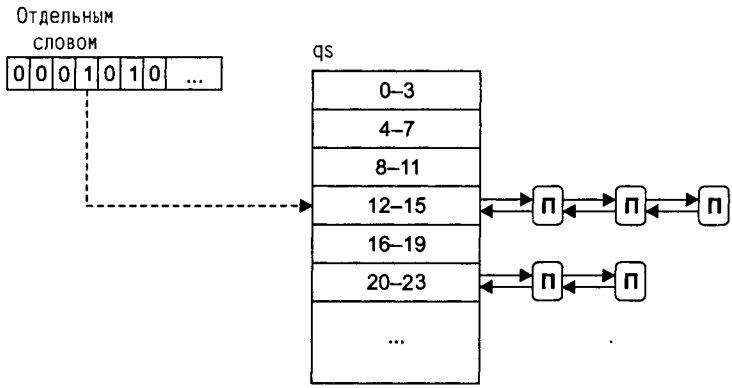


Рис. 5.2. Структуры данных планировщика в системе BSD

Использование массива упрощает задачу выбора процесса для выполнения. Процедура swtch(), производящая переключение контекста, проверяет whichqs и ищет в ней индекс первого установленного бита. Этот индекс используется для идентификации очереди планировщика, содержащей наиболее приоритетные готовые к выполнению процессы. Процедура swtch() удаляет процесс из начала очереди и переключает на него контекст выполнения. По выходу из процедуры swtch() вновь назначенный процесс продолжает выполнение.

При переключении контекста происходит сохранение состояния регистров текущего процесса (регистров общего назначения, счетчика команд, указателя стека и т. д.) в его блоке управления, являющемся частью области u, с последующей загрузкой регистров из сохраненного контекста очередного процесса. Поле p_addr структуры proc указывает на элементы таблицы страниц области u. Это поле использует процедура swtch() для нахождения блока PCB очередного процесса.

Так как машина VAX-11 стала основной платформой для ранних реализаций систем 4BSD и System V, ее архитектура оказала сильное влияние на реализацию планировщика. VAX имела две специальные инструкции — FFS (Find First Set, найти первый установленный) и FFC (Find First Clear, найти первый очищенный), применяемые для работы с 32-битовыми полями. Это послужило причиной размещения 128 приоритетов в 32 очередях. В VAX также имелись специальные инструкции (INSQHI и REMQHI) для атомарных операций вставки и удаления элементов из двунаправленных связанных списков, а также другие команды (LDPCTX и SVPCTX), загружающие и сохраняющие контекст процесса. Все это позволило VAX отрабатывать весь алгоритм планирования, используя при этом небольшой набор машинных инструкций.

5.4.3. Операции с очередью выполнения

Процесс, обладающий наивысшим приоритетом, запускается всегда, если только текущий процесс не выполняется в режиме ядра. Каждому процессу назначается квант времени фиксированного размера (в системе 4.3BSD это 100 миллисекунд). Это обстоятельство действует только при планировании тех процессов, которые находятся в одной очереди выполнения. Через каждые 100 миллисекунд ядро (через отложенный вызов) вызывает процедуру roundrobin() для постановки на выполнение очередного процесса из одной и той же очереди. Если в состоянии готовности к выполнению окажутся процессы с более высоким приоритетом, то они будут назначены на выполнение без ожидания вызова roundrobin(). *Если все остальные процессы, готовые к выполнению, находятся в очередях с более низким приоритетом, то текущий процесс продолжит выполнение даже после вырабатывания отведенного ему кванта времени.*

Процедура schedcpu() пересчитывает приоритет каждого процесса каждую секунду. Так как приоритет процесса, находящегося в очереди на выполнение, не может быть изменен, процедура schedcpu() извлекает процесс из очереди, меняет его приоритет и помещает его обратно, однако не обязательно в ту же очередь. Обработчик прерываний таймера пересчитывает приоритет текущего процесса через каждые четыре тика.

Существуют три ситуации, при которых возникает переключение контекста.

- Если текущий процесс блокируется в ожидании ресурса или завершает работу. При этом происходит свободное переключение контекста.
- Если в результате, полученном процедурой пересчета приоритетов, оказалось, что другой процесс обладает более высоким приоритетом по сравнению с текущим.
- Если текущий процесс или обработчик прерываний разбудил более приоритетный процесс.

Свободное переключение контекста является непосредственным, так как ядро системы напрямую вызывает процедуру swtch() из процедуры sleep() или exit(). События, препятствующие свободному переключению контекста, обычно происходят во время функционирования системы в режиме ядра, следовательно, текущий процесс не может быть вытеснен немедленно. Ядро устанавливает флаг runrun, указывающий на ожидание в очереди процесса с более высоким приоритетом. Перед возвратом текущего процесса в режим задачи ядро проверяет этот флаг. Если он окажется установленным, ядро передаст управление процедуре swtch(), которая инициирует переключение контекста.

5.4.4. Анализ

Классический алгоритм планирования является простым и эффективным. Он вполне приемлем для систем разделения времени как класса, со смешанным выполнением интерактивных и пакетных заданий. Динамический пересчет приоритетов защищает процессы от зависания из-за недостатка процессорного времени. Традиционный метод наиболее подходит для задач, производящих большой объем ввода-вывода и требующих небольших и нечастых «захватов» процессора.

Однако традиционный планировщик имеет ряд ограничений, которые делают его непригодным для использования с широким спектром коммерческих приложений:

- он не слишком хорошо умеет масштабировать: если общее количество процессов велико, он неэффективно пересчитывает приоритеты каждую секунду;
- не существует способов гарантированного предоставления части процессорного времени как ресурса определенной группе процессов или конкретному процессу;

- не существует никакого гарантированного времени реакции приложений, имеющих свойства приложений реального времени;
- приложения имеют скудные возможности для управления собственным приоритетом. Механизм, работающий через значение любезности, является слишком примитивным и не отвечающим поставленным требованиям;
- поскольку ядро системы является невытесняющим, высокоприоритетные готовые к выполнению процессы могут находиться в ожидании в течение довольно значительного интервала времени. Такое свойство системы называется *инверсией приоритетов*.

Современные системы UNIX применяются в самых различных областях деятельности. В частности, существует острая необходимость в планировщике, поддерживающем приложения реального времени, которым необходимо более предсказуемое поведение и ограниченное время реакции. Решение этой проблемы потребовало провести полную переработку планировщика. Все последующие разделы главы посвящены новым средствам планирования, реализованным в системах SVR4, Solaris 2.*x*, OSF/1 и в некоторых менее распространенных вариантах UNIX.

5.5. Планировщик в системе SVR4

В системе SVR4 был представлен полностью переработанный планировщик [1], в котором разработчики попытались улучшить традиционный метод планирования. Он оказался применимым для широкого спектра приложений за счет большей гибкости и управляемости. Ниже представлены основные качества новой архитектуры.

- Поддержка более широкого диапазона приложений, в том числе и требующих работы в режиме реального времени.
- Отделение политики планирования от механизма ее реализации.
- Предоставление приложениям больших возможностей по управлению своими приоритетами и планированию.
- Определение механизма планирования с хорошо описанным интерфейсом взаимодействия с ядром системы.
- Возможность добавления новых политик планирования как отдельных модулей, включая динамически загружаемые реализации планировщика.
- Ограничение на допустимую задержку реакции для критичных ко времени приложений.

Несмотря на то что некоторые усилия были направлены на поддержку приложений реального времени, данная архитектура оказалась, в общем, дос-

таточной для удовлетворения многих других требований планирования. Фундаментальным понятием архитектуры SVR4 явился *класс планирования*, определяющий политику планирования для всех процессов, относящихся к нему. По умолчанию в системе SVR4 поддерживаются два класса: класс разделения времени и класс реального времени.

Планировщик обеспечивает набор процедур, независимых от используемого класса, в которых реализованы общие службы, такие как переключение контекста, управление очередью выполнения и вытеснение. Он также определяет процедурный интерфейс для классо-зависимых функций, таких как расчет приоритета и наследование. Для каждого класса эти функции реализованы по-разному. Например, класс реального времени использует фиксированные приоритеты, в то время как в классе разделения времени приоритет процесса изменяется динамически, как реакция на определенные события.

Этот объектно-ориентированный подход похож на тот, который применяется в архитектуре vnode/vfs (см. раздел 8.6) и подсистеме памяти (см. раздел 14.3). В разделе 8.6.2 показаны основные концепции объектно-ориентированного подхода, используемого в современных системах UNIX. Согласно этому подходу, планировщик является *абстрактным базовым классом*, а каждый класс планирования становится его подклассом (или порожденным им классом).

5.5.1. Независимый от класса уровень

Независимый от класса уровень отвечает за переключение контекста, управление очередью выполнения и вытеснение. Высокоприоритетный процесс всегда получает процессор (кроме случаев невытесняемой обработки в ядре). Количество приоритетов увеличено до 160, и для каждого из них теперь используется отдельная очередь. В отличие от традиционной реализации больший номер приоритета соответствует более высокому приоритету. Однако назначение и пересчет приоритетов процессов совершается на классо-зависимом уровне.

На рис. 5.3 показаны структуры данных, используемые для управления очередью выполнения. Переменная dqactmap является битовой маской, показывающей, какая из очередей выполнения содержит по крайней мере один готовый к выполнению процесс. Для помещения процесса в очередь применяются вызовы setfrontdq() и setbackdq(), для удаления — dispdeq(). Эти функции могут быть вызваны как из основного кода ядра, так и из классо-зависимых процедур планировщика. Обычно очередной готовый к выполнению процесс помещается в конец своей очереди, в то время как процесс, вытесненный до окончания отведенного ему кванта времени, возвращается в начало очереди.

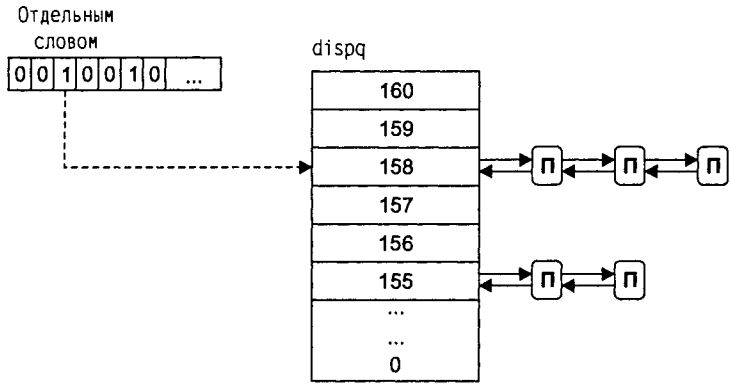


Рис. 5.3. Очереди отправки на выполнение в системе SVR4

Главное ограничение системы UNIX, относящееся к работе приложений реального времени, заключается в невытесняющей природе ядра. Для таких процессов необходимо, чтобы *задержки обслуживания планировщиком* (dispatch latency) (временные промежутки между моментом перехода процесса в состояние готовности к выполнению и началом его выполнения) были минимальны. Если процесс реального времени становится готовым к выполнению в то время, когда текущий процесс выполняет системный вызов, задержка перед переключением контекста может оказаться значительной.

Для решения этой проблемы в ядре системы SVR4 были определены несколько *точек вытеснения*. Эти точки являются определенными местами в коде ядра, в которых все структуры данных находятся в устойчивом состоянии, а ядро системы готово начать производство большого объема операций. Когда достигается одна из точек вытеснения, ядро проверяет флаг kprunrun, который указывает на готовность к выполнению процесса реального времени. Если флаг установлен, ядро системы вытеснит текущий процесс. Такой подход ограничивает время ожидания процесса реального времени перед тем, как он будет назначен на выполнение[1]. Макроопределение PREEMPT() проверяет флаг kprunrun и вызывает для вытеснения процесса процедуру preempt(). Ниже перечислены некоторые примеры точек вытеснения.

- В процедуре разбора имен путей lookuppn(): перед началом анализа каждого индивидуального компонента имени пути.
- В системном вызове open: перед созданием файла, если тот не существует.
- В подсистеме памяти: перед освобождением страниц памяти, занимаемых процессом.

[1] Этот код не является классо-зависимым, несмотря на явное упоминание процессов реального времени. Ядро только проверяет kprunrun для определения, должно ли оно вытеснить текущий процесс. На текущий момент только класс реального времени устанавливает этот флаг, однако в будущем может появиться и новый класс, также требующий вытеснения процессов, работающих в режиме ядра.

Флаг runrun используется точно так же, как в традиционных системах, и приводит к вытеснению процессов только при возврате их в режим задачи. Функция preempt() вызывает операцию CL_PREEMPT для выполнения классозависимых действий. Затем она вызывает swtch() для инициализации переключения контекста.

Функция swtch() вызывает pswtch() для проведения машинно-независимой части переключения контекста, после чего вызывает подпрограммы низкого уровня[1] для манипуляций с контекстом регистров, очистки буферов трансляции и т. д. Функция pswtch() сбрасывает флаги runrun и kprunrun, выбирает самый высокоприоритетный готовый к выполнению процесс и удаляет его из очереди отправки на выполнение. Эта функция также обновляет dqactmap и устанавливает состояние процесса в значение SONPROC (выполняемый на процессоре). В завершение она производит обновление регистров управления памяти, отображенных в области u, и карт трансляции адресов этого процесса.

5.5.2. Интерфейс с классами планирования

Вся классо-зависимая функциональность обеспечивается через общий интерфейс, реализация виртуальных функций (см. раздел 8.6.2) которого индивидуальна для каждого класса. Интерфейс определяет как семантику этих функций, так и связи, используемые в вызове определенной реализации для класса.

На рис. 5.4 показана реализация классо-зависимого интерфейса. Структура classfuncs является вектором указателей на функции, реализующие классо-зависимый интерфейс какого-либо класса. В глобальной таблице классов для каждого класса отводится по одному элементу. Такой элемент содержит имя класса, указатель на функцию инициализации и указатель на вектор classfuncs для этого класса.

Когда процесс создается, он наследует приоритет класса от своего родителя. Впоследствии процесс может быть переведен и в другой класс при помощи системного вызова priocntl, описанного в разделе 5.5.5. Структура proc имеет три поля, используемых классами планирования:

p_cid	Идентификатор класса, являющийся индексом в глобальной таблице классов
p_clfuncs	Указатель на вектор classfuncs для класса, к которому относится процесс. Указатель копируется из соответствующего элемента глобальной таблицы классов
p_clproc	Указатель на приватную классо-зависимую структуру данных

Набор макроопределений преобразует вызовы, сделанные через общий интерфейс функций, в соответствующие классо-зависимые функции. Например:

```
#define CS_SLEEP(procp, clprocp, ...) \
(*(procp)->p_clfuncs->cl_sleep)(clprocp, ...)
```

[1] Автор, видимо, имел в виду машинно-зависимые коды. — *Прим. ред.*

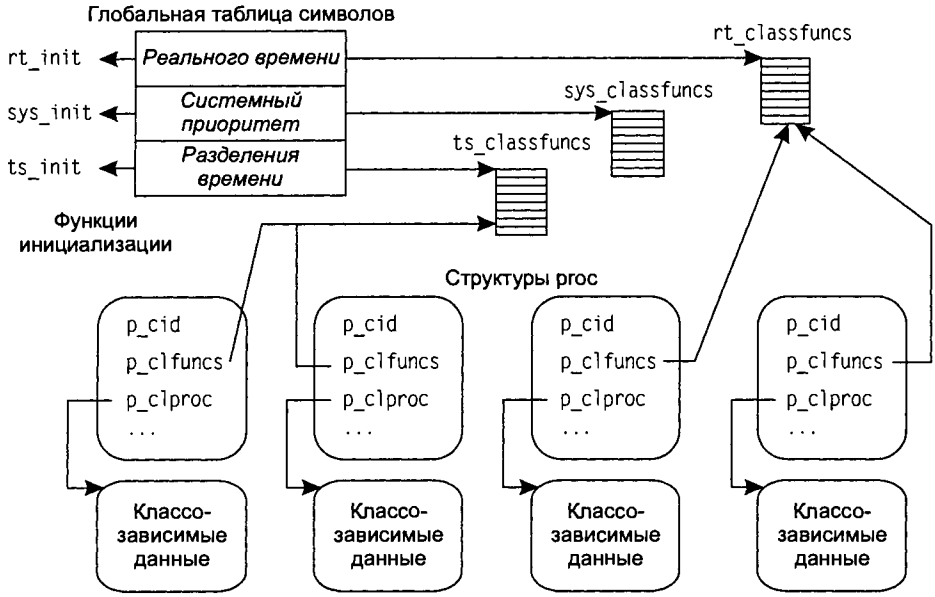


Рис. 5.4. Классо-зависимый интерфейс системы SVR4

Таким образом, классо-зависимые функции могут быть вызваны из классо-независимого кода, а также через системный вызов priocntl.

Класс планирования сам выбирает политику для расчета приоритетов и планирования процессов, относящихся к этому классу. Он определяет диапазон приоритетов для принадлежащих ему процессов, а также условия, при которых приоритет процесса может быть изменен. Класс также задает размер кванта времени работы процесса. Квант может быть как одинаковым для всех процессов, так и зависимым от приоритета процесса. Величина кванта может лежать в диапазоне от одного тика до бесконечности. Установка «бесконечного» кванта наиболее подходит для некоторых задач реального времени, которые должны находиться в режиме выполнения постоянно вплоть до завершения.

Точки входа классо-зависимого интерфейса включают в себя:

CL_TICK	Вызывается из обработчика прерываний таймера. Применяется для отслеживания квантов времени, пересчета приоритетов, обработки исчерпания кванта и т. д
CL_FORK, CL_FORKRET	Вызываются из fork. CL_FORK инициализирует определенную для класса структуру данных потомка. CL_FORKRET может устанавливать флаг runrun, позволяя процессу-потомку стартовать до своего предка
CL_ENTERCLASS, CL_EXITCLASS	Вызываются при перемещении процесса в класс планирования и из него. Отвечают за размещение и освобождение определенной для класса структуры данных соответственно

CL_SLEEP	Вызывается из sleep(). Может пересчитывать приоритет процесса
CL_WAKEUP	Вызывается из wakeprocs(). Помещает процесс в подходящую для него очередь выполнения, может устанавливать флаги runrun и kprunrun

Класс планирования сам решает, какие действия будет производить каждая функция, и в каждом классе эти функции могут быть определены по-своему. Такой подход придает общему планированию большую гибкость. Например, обработчик прерываний таймера традиционного планировщика производит подсчет каждого тика текущего процесса и пересчитывает его приоритет на каждом четвертом тике. В новой архитектуре обработчик вызывает процедуру CL_TICK, определенную для класса, которому принадлежит процесс. Процедура самостоятельно решает, как обрабатывать тики времени. Например, класс реального времени использует постоянные приоритеты и не требует их пересчета. Классо-зависимый код определяет момент, когда происходит исчерпание выделенного кванта времени, и устанавливает флаг runrun для инициализации процедуры переключения контекста.

По умолчанию все 160 поддерживаемых приоритетов разделяются на следующие три категории:

0–59	класс разделения времени
60–99	системные приоритеты
100–159	класс реального времени

Дальнейшие подразделы посвящены описанию реализации классов разделения времени и реального времени.

5.5.3. Класс разделения времени

Класс разделения времени является классом по умолчанию для процесса. Он изменяет приоритеты процесса динамически и реализует *карусельное планирование* для процессов одного приоритета. Класс использует для управления приоритетами процессов и квантами времени статическую *таблицу параметров планировщика*. Величина кванта времени, выделяемого конкретному процессу, зависит от его приоритета. В таблице параметров определяются кванты времени для каждого приоритета. По умолчанию чем меньше приоритет процесса, тем больше времени ему предоставляется для выполнения. Такой подход может показаться нелогичным, однако причиной его применения является тот факт, что низкоприоритетные процессы выполняются нечасто и должны получать больше времени, когда наступает их очередь выполняться.

Класс разделения времени использует планирование, управляемое по событиям [17]. Вместо пересчета приоритетов всех процессов каждую секунду

в системе SVR4 изменение приоритета процесса происходит в ответ на определенные события, относящиеся к этому процессу. Планировщик уменьшает приоритет процесса каждый раз, как он вырабатывает отведенный ему квант времени. С другой стороны, система увеличивает приоритет процесса, если тот блокируется по событию или находится в ожидании ресурса или если он долгое время не может израсходовать свой квант времени. Так как каждое событие обычно относится только к одному процессу, пересчет приоритета происходит очень быстро. В таблице параметров диспетчера определено, как различные события влияют на приоритет процесса.

Класс разделения времени использует для хранения классо-зависимых данных структуру tsproc. Эта структура включает следующие поля:

ts_timeleft	оставшееся время кванта
ts_cpupri	системную часть приоритета
ts_upri	прикладную часть приоритета (фактор «любезности»)
ts_umdpri	приоритет режима задачи (ts_cpupri + ts_upri, но не более чем 59)
ts_dispwait	число секунд таймера после начала кванта

Когда процесс возобновляет свою работу после сна, находясь в режиме ядра, его приоритет соответствует приоритету ядра и определяется условиями сна. Когда позже процесс возвращается в режим задачи, его приоритет восстанавливается из ts_umdpri. Приоритет режима задачи является суммой ts_cpupri и ts_upri, но ограничивается диапазоном значений от 0 до 59. Значение ts_upri лежит в диапазоне от −20 до +19 и по умолчанию равно 0. Значение этого поля может быть изменено вызовом priocntl, однако права на его увеличение даны только суперпользователю системы. Значение ts_cpupri выбирается из таблицы параметров диспетчера, как это будет описано ниже.

Таблица параметров содержит по одному элементу для каждого приоритета класса. Хотя в системе SVR4 каждый класс обладает такой таблицей (плюс еще одна для системных приоритетов), каждая из этих таблиц имеет различный формат. Она не является обязательной структурой для каждого класса, существует также возможность создания нового класса без таблицы. Для класса разделения времени каждый элемент таблицы состоит из следующих полей:

ts_globpri	Глобальный приоритет элемента (для класса разделения времени это то же самое, что индекс таблицы)
ts_quantum	Квант времени для этого приоритета
ts_tqexp	Новое значение ts_cpupri, устанавливаемое при исчерпании кванта времени
ts_slpret	Новое значение ts_cpupri, устанавливаемое при возврате в режим задачи после сна

ts_maxwait	Количество секунд ожидания до исчерпания кванта времени перед применением ts_lwait
ts_lwait	Используется вместо ts_tqexp, если процесс при исчерпании своего кванта затратил времени больше, чем значение ts_maxwait

Таблица может применяться двумя способами. Она может быть проиндексирована по текущим значениям поля ts_cpupri для получения доступа к полям ts_tqexp, ts_slpret и ts_lwait, так как эти поля предоставляют новое значение ts_cpupri, основанное на его прежнем значении. Таблица может быть проиндексирована по полю ts_umdpri для получения доступа к ts_globpri, ts_quantum и ts_maxwait, поскольку именно эти поля отвечают за общие приоритеты планирования.

Таблица 5.1 является примером типичной таблицы параметров диспетчера для класса разделения времени. Для того чтобы понять, как она может быть использована, представим процесс, обладающий ts_upri=14 и ts_cpupri=1. Значения его глобального приоритета (ts_globpri) и ts_umdpri одинаковы и равны 15[1]. Если процесс исчерпает выделенный ему квант времени, то его ts_cpupri будет установлено в значение 0 (следовательно, его ts_umdpri будет равно 14). Однако если процессу необходимо более 5 секунд для того, чтобы использовать отведенный ему квант времени, его ts_cpupri будет установлено в значение 11 (следовательно, ts_umdpri будет равным 25).

Предположим, что перед тем, как исчерпать отведенный ему квант времени, процесс произведет системный вызов, и это приведет к необходимости его блокирования в ожидании ресурса. Когда процесс возобновит работу и, в конечном счете, перейдет в режим задачи, его ts_cpupri установится в 11 (из колонки ts_slpret[2]), а поле ts_umdpri станет равным 25, в зависимости от того, сколько времени понадобится процессу на исчерпание отведенного ему кванта времени.

Таблица 5.1. Таблица параметров диспетчера для класса разделения времени

Индекс	globpri	Quantum	tqexp	slpret	maxwait	Lwait
0	0	100	0	10	5	10
1	1	100	0	11	5	11
...	...	...	...	...	...	...
15	15	80	7	25	5	25
...	...	...	...	...	...	...
40	40	20	30	50	5	50
...	...	...	...	...	...	...
59	59	10	49	59	5	59

[1] По всей видимости, здесь должно быть ts_globpri=1. — *Прим. ред.*

[2] Точнее, из Slpret. — *Прим. ред.*

5.5.4. Класс реального времени

Класс реального времени использует приоритеты в диапазоне 100–159. Эти приоритеты являются более высокими, чем у процессов, принадлежащих классу разделения времени, даже в случае выполнения их в режиме ядра. Это означает, что процессы реального времени будут выбраны для выполнения перед любым процессом, выполняющимся в режиме ядра. Предположим, что текущий процесс работает в режиме ядра в то время, когда процесс из класса реального времени становится готовым к выполнению. Ядро не может сразу же вытеснить текущий процесс, так как это способно повлечь нестабильное состояние системы. Процесс реального времени должен ожидать момента, когда текущий процесс начнет переключаться в режим задачи или достигнет одной из точек вытеснения ядра. В класс реального времени могут войти только процессы суперпользователя, для этого они используют вызов priocntl, в параметрах которого указываются необходимый приоритет и квант времени.

Процессы реального времени характеризуются постоянным приоритетом и постоянным выделяемым квантом. Единственным способом изменения этих параметров является применение вызова prionctl. Таблица параметров диспетчера для класса реального времени является очень простой, так как в ней хранятся только значения квантов времени для каждого приоритета, принятые по умолчанию. Они используются в том случае, если при вхождении в класс реального времени процесс не указывает квант самостоятельно. Для более низких приоритетов по умолчанию назначаются более продолжительные кванты времени. Классо-зависимые данные процесса реального времени хранятся в структуре rtproc, куда входят текущий квант, время, оставшееся до исчерпания кванта, и текущий приоритет.

Процессы реального времени требуют ограничения времени задержек обслуживания планировщиком и времени реакции. Оба этих понятия продемонстрированы на рис. 5.5. Задержка обслуживания планировщиком — это интервал времени между тем моментом, когда процесс становится готовым к выполнению, и тем, когда он начнет свою работу. Время реакции — промежуток времени между возникновением события, требующего реакции от процесса, и моментом начала обработки процессом этого события. Оба этих параметра должны иметь определенную верхнюю границу, лежащую в разумных пределах.

Время реакции является суммой времени, требуемого для обработки события обработчиком, времени задержки обслуживания планировщиком и времени, необходимого для проведения обработки события самим процессом реального времени. Величина задержки обслуживания планировщиком во многом зависит от ядра. Ядра традиционных систем не в состоянии обеспечить приемлемых границ этой задержки, так как они являются невытесняющими, вследствие чего готовый к выполнению процесс класса реального времени может ждать довольно долго, если текущий процесс «застрянет» в некоторой

замысловатой процедуре ядра. Измерения показали, что время выполнения некоторых участков кода ядра может доходить до нескольких миллисекунд, что совершенно неприемлемо для приложений реального времени.

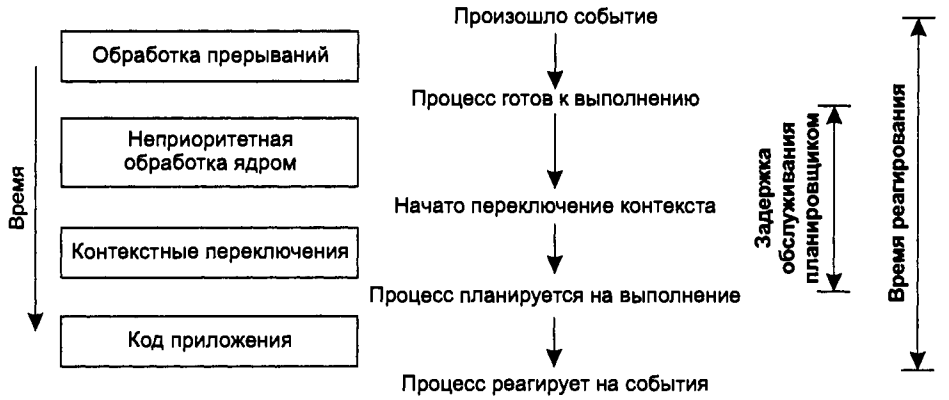


Рис. 5.5. Время реакции и задержки обслуживания планировщиком

В системе SVR4 для разделения длинных алгоритмов ядра на более короткие и ограниченные блоки выполнения применяются точки вытеснения. Когда процесс реального времени становится готовым к выполнению, процедура rt_wakeup(), которая обрабатывает классо-зависимую часть процедуры пробуждения, устанавливает определенный в ядре флаг kprunrun. Когда текущий процесс (предположительно выполняющийся в режиме ядра) достигает точки вытеснения, ядро проверяет флаг и инициирует переключение контекста для процесса реального времени. Таким образом, время ожидания ограничено временем выполнения максимально длинного участка кода между двумя точками вытеснения, что является удовлетворительным решением для многих задач.

В завершение необходимо упомянуть о том, что соблюдение границ задержки может быть гарантировано только в том случае, если процесс реального времени является самым высокоприоритетным процессом системы. Если в течение любого периода своего ожидания высокоприоритетный процесс становится готовым к выполнению, то он будет назначен на выполнение в первую очередь. После того как процесс получит процессор в свое распоряжение, пересчет задержки начнется с нуля.

5.5.5. Системный вызов priocntl

Системный вызов priocntl предлагает ряд возможностей для управления приоритетами и планированием процесса. Для него определен набор подкоманд, которые могут быть использованы для различных операций, таких как:

- изменение приоритета класса процесса;
- установка значения ts_upri для процессов класса разделения времени;

- сброс в значение по умолчанию приоритета и кванта времени для процесса реального времени;
- получение текущего значения ряда параметров планирования.

Большинство из этих операций может производить только суперпользователь, следовательно, они недоступны большинству приложений. В системе SVR4 также имеется системный вызов priocntlset, производящий те же операции, но над набором связанных чем-либо процессов, например:

- всех процессов системы;
- всех процессов группы или сеанса;
- всех процессов в определенном классе планирования;
- всех процессов, принадлежащих определенному пользователю;
- всех процессов, имеющих одного и того же родителя.

5.5.6. Анализ

В системе SVR4 традиционный планировщик был заменен новым планировщиком, отличным по устройству и по принципам работы. Новый планировщик обеспечивает гибкий подход, позволяющий добавлять новые классы планирования в систему. Производители программного обеспечения получили возможность «скроить» свой планировщик, который бы удовлетворял их приложениям. Применение таблиц параметров диспетчера расширило возможности системного администратора, поскольку появилась способность контролировать поведение системы путем изменения табличных установок и последующей пересборки ядра.

В традиционных системах UNIX пересчет приоритетов процессов ведется каждую секунду. Это может отнимать довольно много времени, если в системе имеется большое количество процессов. Следовательно, такой алгоритм не обладает достаточными возможностями по масштабированию в системах, обслуживающих тысячи процессов. В системе SVR4 класс разделения времени изменяет приоритеты процессов, основываясь на событиях, относящихся к этим процессам. Так как обычно событие относится лишь к одному процессу, применяемый алгоритм обладает высоким быстродействием и масштабируемостью.

Событийно управляемое планирование наиболее подходит для заданий, производящих в основном операции ввода-вывода, и интерактивных заданий, нежели для заданий, ограниченных только обработкой на процессоре. Такой подход имеет несколько недостатков. Пользователи, чьи интерактивные задания также требуют больших объемов вычислений, не сумеют на подобной системе эффективно с ними работать, поскольку такие процессы не смогут вырабатывать достаточное количество увеличивающих приоритет событий для более активного использования процессора. В дополнение к это-

му связанные с событиями операции по оптимальному повышению или понижению приоритетов зависят от общей загруженности системы и характеристик задач, работающих в текущий момент времени. Таким образом, для повышения эффективности системы и скорости реакции эти величины могут часто перенастраиваться.

Добавление какого-либо класса планирования не требует доступа к исходным кодам ядра. Для этой цели разработчик должен произвести следующие последовательные действия.

1. Разработать реализацию каждой классо-зависимой функции нового класса планирования.
2. Инициализировать вектор classfuncs указателями на эти функции.
3. Разработать функцию инициализации, выполняющую различные действия (например, выделить память для внутренних структур данных и т. д.) при установке нового класса.
4. Добавить запись для этого класса в таблицу классов в основном файле конфигурации, находящемся обычно в подкаталоге master.d каталога сборки ядра. Эта запись должна содержать указатели на функцию инициализации и на вектор classfuncs.
5. Произвести пересборку ядра.

Существенным ограничением является то, что в системе SVR4 не обеспечено сколько-нибудь приемлемого способа для перевода процессов класса разделения времени в иной класс. Вызов priocntl может быть в распоряжении только суперпользователя. Был бы очень полезен механизм, который умел бы отображать определенные идентификаторы пользователей или программы в класс, отличный от принятого по умолчанию.

Несмотря на то что средства поддержки класса реального времени явились важным шагом, они оказались далеки до соответствия желаемым возможностям. Например, система не поддерживает планирование по крайнему сроку (см. раздел 5.9.2). Код между двумя точками вытеснения все равно остается слишком длинным для многих приложений, критичных ко времени. Более того, настоящие системы реального времени должны обладать полностью вытесняющим ядром. Некоторые из этих аспектов были реализованы в системе Solaris 2.*x*, в которой были расширены возможности планировщика SVR4. Мы расскажем о них в следующем разделе главы.

Основной проблемой планировщика SVR4 является чрезвычайная сложность более-менее качественной настройки системы при работе со смешанными наборами приложений. В книге [13] описывается эксперимент, при котором в системе одновременно запускались три различных приложения: интерактивный сеанс с клавиатурным вводом данных, пакетное задание и программа для просмотра видео. Это оказалось весьма сложным предприятием, поскольку программа, принимающая клавиатурный ввод, и программа для просмотра видео требовали взаимодействия с X-server.

Для решения этой задачи авторы книги изменяли приоритеты и классы планирования всех четырех процессов (трех приложений и X-server). *Выяснилось, что найти комбинацию параметров, обеспечивающих приемлемое протекание работы всех приложений, очень трудно.* Например, вполне логичным решением представляется размещение в классе реального времени только программы воспроизведения видео. На деле же оказалось, что даже программа работы с видео не может нормально функционировать. Вся проблема заключалась в X-server, который не получал достаточного количества процессорного времени. Тогда авторы разместили в классе реального времени еще и X-server, в результате чего они добились нормальной скорости воспроизведения видео (после корректного подбора относительных приоритетов), но при этом функционирование интерактивного приложения и пакетного задания застопорилось, а система перестала отвечать на нажатия клавиш и движения мыши.

Возможно, что после длительных экспериментов есть шансы подобрать правильное сочетание приоритетов для данного набора приложений. Однако эти установки будут работать корректно только при использовании именно этого набора программ. В реальности же загрузка системы меняется постоянно, а подстраивать ее вручную при старте каждого нового приложения просто неразумно.

5.6. Расширенные возможности планирования системы Solaris 2.x

В системе Solaris 2.*x* были расширены возможности архитектуры планирования SVR4 сразу в нескольких направлениях [9]. Solaris является многонитевой, симметрично многопроцессорной операционной системой, следовательно, ее планировщик должен поддерживать все эти особенности. Кроме этого разработчики произвели оптимизацию системы в целях уменьшения задержек обслуживания планировщиком для высокоприоритетных, критичных ко времени процессов. В результате получился планировщик, более пригодный для приложений реального времени.

5.6.1. Вытесняющее ядро

Точки вытеснения в ядре SVR4 являются наилучшим компромиссным решением, позволяющим ограничить задержки, что требуется для процессов реального времени. Ядро системы Solaris 2.*x* является полностью вытесняющим, что позволяет гарантировать быстроту его реакции. Это явилось радикальным изменением ядра UNIX и имело далеко идущие последствия. Большинство глобальных структур ядра необходимо защищать при помощи соответствующих объектов синхронизации, таких как семафоры или *взаимные исключения* (mutual exclusion locks, mutex). Сделать ядро вытесняющим — очень сложная

задача, однако решение этой проблемы является необходимым требованием для многопроцессорных операционных систем.

Другим подобным изменением в системе Solaris 2.*x* является реализация прерываний при помощи специальных нитей ядра, использующих стандартные средства синхронизации ядра и блокирующихся в ожидании ресурсов, если это необходимо (см. раздел 3.6.5). В результате в системе нечасто требуется повышать уровень прерываний для защиты критичных участков кода и она имеет лишь несколько невытесняемых сегментов кода. Таким образом, высокоприоритетные процессы могут быть назначены на выполнение сразу после того, как они станут готовыми к выполнению.

Нити прерываний всегда выполняются в системе с наивысшими приоритетами. ОС Solaris позволяет динамически загружать классы планирования. При этом приоритеты нитей прерываний пересчитываются для гарантии того, что они останутся с самыми высокими значениями. Если нити прерывания необходимо блокироваться в ожидании ресурса, то возобновить функционирование она сможет только на том же самом процессоре.

5.6.2. Многопроцессорная поддержка

Система поддерживает единую очередь диспетчеризации для всех процессоров. Тем не менее некоторые нити (например, нити прерываний) могут быть ограничены выполнением только на одном, определенном процессоре. Процессоры могут взаимодействовать друг с другом при помощи отправки *межпроцессорных прерываний*. Каждый процессор обладает следующим набором переменных планирования:

cpu_thread	текущая нить, выполняемая на этом процессоре
cpu_dispthread	нить, выбранная последней для запуска
cpu_idle	нить холостой работы
cpu_runrun	флаг вытеснения, используемый нитями класса разделения времени
cpu_kprunrun	флаг вытеснения, устанавливаемый нитями класса реального времени
cpu_chosen_level	Приоритет нити, собирающейся вытеснить текущую нить

Планирование в многопроцессорных средах показано на рис. 5.6. Событие на процессоре П1 делает нить Н6 (с приоритетом 130) готовой к выполнению. Ядро помещает Н6 в очередь отправки на выполнение и вызывает процедуру cpu_choose() для нахождения процессора, на котором выполняется нить, имеющая наименьший приоритет (в данном случае это — П3). Так как приоритет найденной нити окажется меньше, чем у Н6, процедура cpu_choose() пометит процессор, на котором эта нить выполняется, для вытеснения, установит его значение переменной cpu_chosen_level равным приоритету Н6 (130) и пошлет ему межпроцессорное прерывание. Предположим, что в это время, то есть до того как процессор П3 обработает прерывание и вытеснит нить Н3, другой

процессор, например П2, обрабатывает событие, которое сделает нить Н7 с приоритетом 115 готовой к выполнению. Теперь процедура cpu_choose() проверит значение cpu_chosen_level процессора П3 и обнаружит его равным 130. Это приведет к тому, что процедура выявит, что на данном процессоре выполняется нить с более высоким приоритетом. Следовательно, процедура cpu_choose() оставит Н7 в очереди отправки на выполнение, предупреждая конфликт.

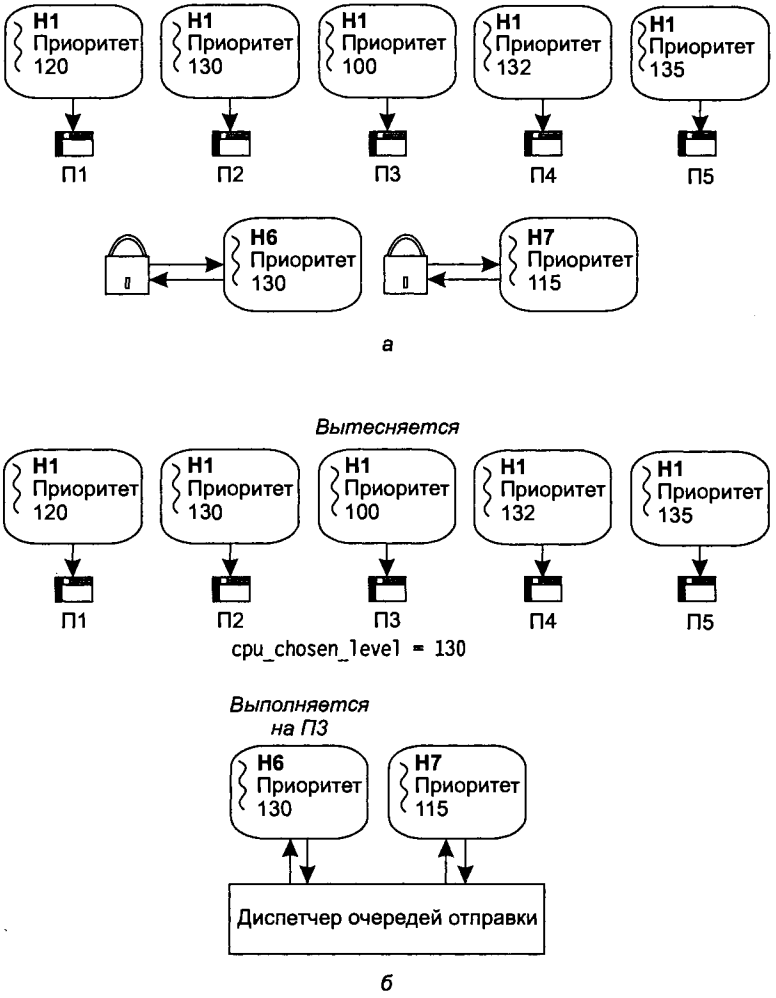


Рис. 5.6. Многопроцессорное планирование в системе Solaris 2.*x*: *а* — начальное состояние; *б* — после того как Н6 и Н7 станут готовы к выполнению

Существуют определенные ситуации, когда низкоприоритетная нить может заблокировать более высокоприоритетную нить на длительный период времени. Причиной возникновения подобных ситуаций является либо *скры-*

тое планирование, либо *инверсия приоритетов*. В системе Solaris устранены подобные проблемы. Как это сделано, описано в следующих разделах главы.

5.6.3. Скрытое планирование

Ядро системы часто совершает некоторые действия асинхронно от имени нити. Ядро планирует эту работу без учета приоритета нити, для которой ее выполняет. Это называется *скрытым планированием*. Примерами таких действий ядра являются процедуры обслуживания STREAMS (см. раздел 17.4) и отложенные вызовы.

В операционной системе SVR4, например, перед каждым возвратом процесса в режим задачи ядро вызывает процедуру runqueues() для проверки существования ожидающих *запросов на обслуживание*. Если таковые существуют, то ядро обрабатывает эти запросы при помощи вызова процедуры service соответствующего модуля STREAMS. Таким образом, подобные запросы обслуживаются *текущим процессом* (тем самым, который собирается возвратиться в режим задачи), хотя они относятся к совсем другим процессам. Если приоритет процесса, от чьего имени был сделан запрос, ниже, чем у текущего, запрос будет обработан с неверным приоритетом. В результате нормальная работа текущего процесса прерывается выполнением низкоприоритетной задачи.

В Solaris эта проблема решается посредством перевода функционирования STREAMS на уровень нитей ядра, которые всегда обладают приоритетом меньшим, чем у нитей режима реального времени. Однако такое решение порождает новую проблему: некоторые запросы STREAMS могут быть инициированы нитями реального времени. Поскольку эти запросы также обслуживаются нитями ядра, они обрабатываются с более низким приоритетом, чем должны. Эта проблема не решается без кардинальных изменений в семантике обработки нитей и остается возможным препятствием для функционирования режима реального времени на требуемом уровне.

Также существует проблема, связанная с обработкой отложенных вызовов (см. раздел 5.2.1). В системе UNIX все отложенные вызовы обслуживаются с самым низким уровнем приоритета прерываний, являющимся, однако, выше любого приоритета реального времени. Если такой вызов будет произведен низкоприоритетной нитью, то его обслуживание может задержать постановку на выполнение высокоприоритетной нити. Измерения производительности, проводимые на ранних версиях SunOS, показали, что для обработки очереди отложенных вызовов система способна потратить до 5 миллисекунд.

Для решения означенной проблемы разработчики Solaris переложили обработку таких вызовов на *нить отложенных вызовов*, которая выполняется с максимальным системным приоритетом, который, вместе с тем, ниже любого приоритета реального времени. Вызовы, произведенные процессами реального времени, обрабатываются отдельно и обладают низшими приоритетами прерываний. Это гарантирует своевременное выполнение отложенных вызовов, критичных ко времени.

5.6.4. Инверсия приоритетов

Проблема *инверсии приоритетов* была впервые описана в работе [10] и относится к ситуации, когда низкоприоритетный процесс удерживает ресурс, необходимый процессу с более высоким приоритетом. Таким образом, он блокирует работу более высокоприоритетного процесса. Существует несколько разновидностей сформулированной проблемы. Рассмотрим их на примерах (рис. 5.7).

В простейшем случае нить H1 удерживает ресурс P, который необходим более высокоприоритетной нити H2. Последняя будет ожидать до тех пор, пока H1 не освободит ресурс. Усложним сценарий, добавив нить H3, приоритет которой меньше, чем у H2, но больше, чем у H1 (рис. 5.7, *а*). Предположим также, что H2 и H3 являются нитями реального времени. Поскольку нить H2 заблокирована, то H3 на данный момент является наиболее высокоприоритетной и готовой к выполнению нитью, и поэтому именно она вытеснит нить H1 (рис. 5.7, *б*). В результате нить H2 останется блокированной до тех пор, пока нить H3 либо завершит работу, либо заблокируется, и только после этого нить H1 начнет функционировать и освободит ресурс.

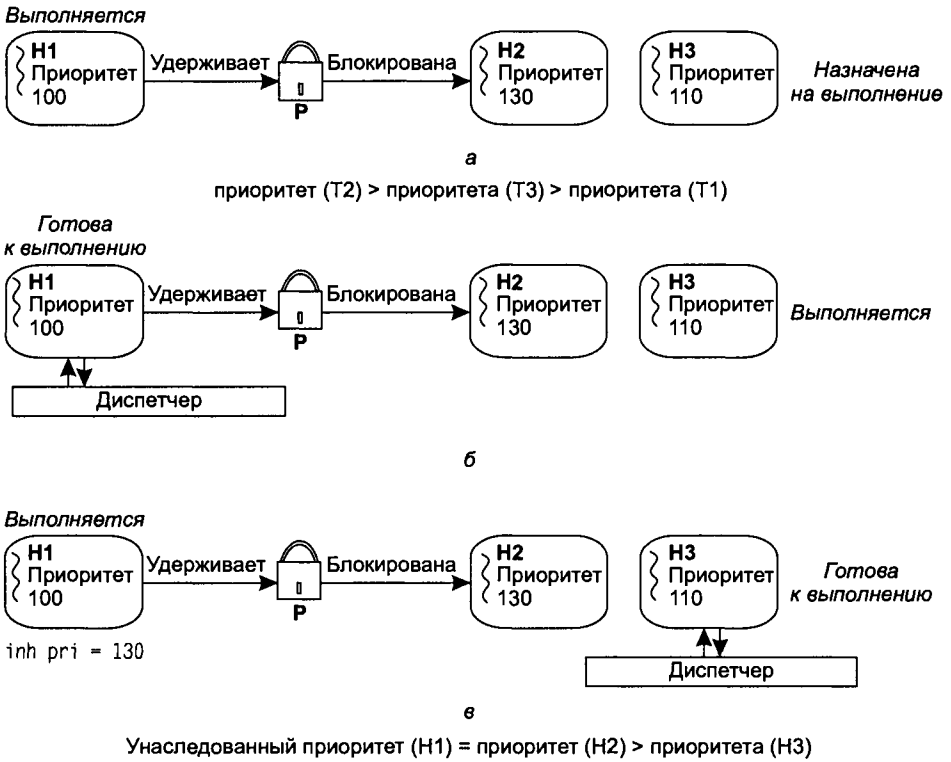


Рис. 5.7. Простейший случай инверсии приоритетов: *а* — начальная ситуация, *б* — без наследования приоритетов, *в* — с наследованием приоритетов

Описываемая проблема может быть решена при использовании метода *наследования приоритетов* или *временной передачи приоритетов*. Когда высокоприоритетная нить блокируется в ожидании ресурса, она временно передает свой приоритет менее приоритетной нити, которая в данный момент удерживает этот ресурс. Таким образом, в приведенном выше примере нить H1 унаследует приоритет нити H2 и теперь не сможет быть вытеснена нитью H3 (рис. 5.7, *в*). Когда нить H1 освободит ресурс, ее приоритет снова вернется в оригинальное значение, что позволит H2 вытеснить ее. Нить H3 будет назначена на выполнение только после того, как H1 освободит ресурс, а H2 закончит работу и отдаст процессор.

Наследование приоритетов должно быть переходным. На рис. 5.8 нить H4 блокируется в ожидании ресурса, удерживаемого нитью H5, которая, в свою очередь, блокирована на ресурсе, удерживаемом нитью H6. Если приоритет нити H4 выше, чем у H5 и H6, то нить H6 должна унаследовать приоритет нити H4 через нить H5. В противном случае нить H7, обладающая приоритетом большим, чем у H5 и H6, но меньшим, чем у H4, вытеснит нить H6 и послужит причиной инверсии приоритетов в отношении H4. Таким образом, унаследованный приоритет нити должен равняться приоритету высокоприоритетной нити, непосредственно или косвенно ожидающей данный ресурс.

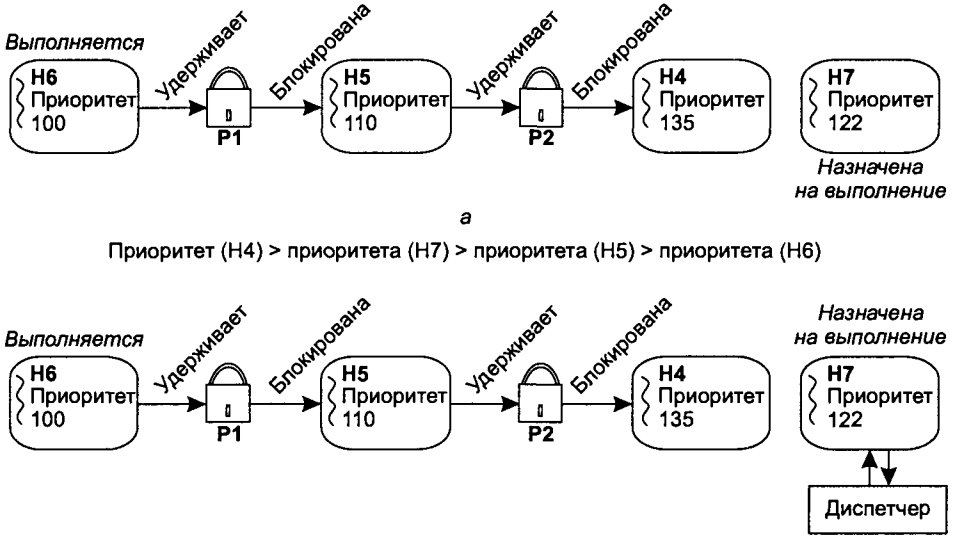


Рис. 5.8. Переходная инверсия приоритетов: *а* — начальная ситуация, *б* — при переходящем наследовании

Для реализации наследования приоритетов ядро системы Solaris должно содержать дополнительную информацию о занятых объектах. Необходимо

идентифицировать, какая нить в данный момент назначена владельцем каждого занятого объекта, а также те объекты, в ожидании которых находится каждая блокированная нить. Так как наследование является переходным, ядро должно иметь возможность просматривать все объекты и блокированные нити в *цепочке синхронизации*, начинающейся от каждого данного объекта. О том, как реализовано наследование приоритетов в системе Solaris, будет рассказано в следующем подразделе.

5.6.5. Реализация наследования приоритетов

Каждая нить обладает двумя приоритетами: *глобальным приоритетом*, определяемым классом планирования, и *унаследованным приоритетом*, зависящим от взаимодействия нити с объектами синхронизации. Унаследованный приоритет обычно равен нулю до тех пор, пока нити не будет передан чей-либо приоритет. Приоритет планирования нити выше, чем ее глобальный и унаследованный приоритеты.

Когда нить должна блокироваться в ожидании ресурса, она вызывает функцию pi_willto() для передачи своего приоритета всем нитям, которые прямо или косвенно блокируют ресурс. Так как наследование является переходным, функция pi_willto() передает унаследованный приоритет вызывающей нити[1]. Функция pi_willto() просматривает цепочку синхронизации этой нити, начиная с объекта, на котором нить заблокирована напрямую. Этот объект содержит указатель на свою *нить-владельца*, удерживающую в текущий момент защелку. Если приоритет планирования нити-владельца ниже, чем наследуемый приоритет вызывающей нити, то нить-владелец ресурса унаследует более высокое значение приоритета. Если нить-владелец объекта заблокируется в ожидании другого ресурса, ее структура нити будет содержать указатель на соответствующий объект синхронизации. Функция pi_willto(), следуя по этому указателю, передаст приоритет нити-владельцу объекта и т. д. Цепочка синхронизации закончится тогда, когда достигнет незаблокированной нити или объекта, приоритет которого не был инвертирован[2].

Разберем пример, приведенный на рис. 5.9. Нить H6 является текущей и обладает глобальным приоритетом, равным 110. Она желает заполучить ресурс P4, удерживаемый нитью H5. Ядро вызывает функцию pi_willto(), которая просматривает цепочку синхронизации, начинающуюся от P4, производя при этом описанные ниже действия.

---

[1] В том случае, видимо, когда унаследованный приоритет выше глобального, то есть функция вызывается нитью, находящейся не в начале цепочки. — *Прим. ред.*

[2] Приведенный алгоритм известен под названием *вычисления транзитивного замыкания* (computation of transitive closure), а просматриваемая цепочка представляет собой *прямой ациклический граф*. (Матрица транзитивного замыкания составляется при помощи последовательного удаления узлов, встречающихся дважды.) — *Прим. ред.*

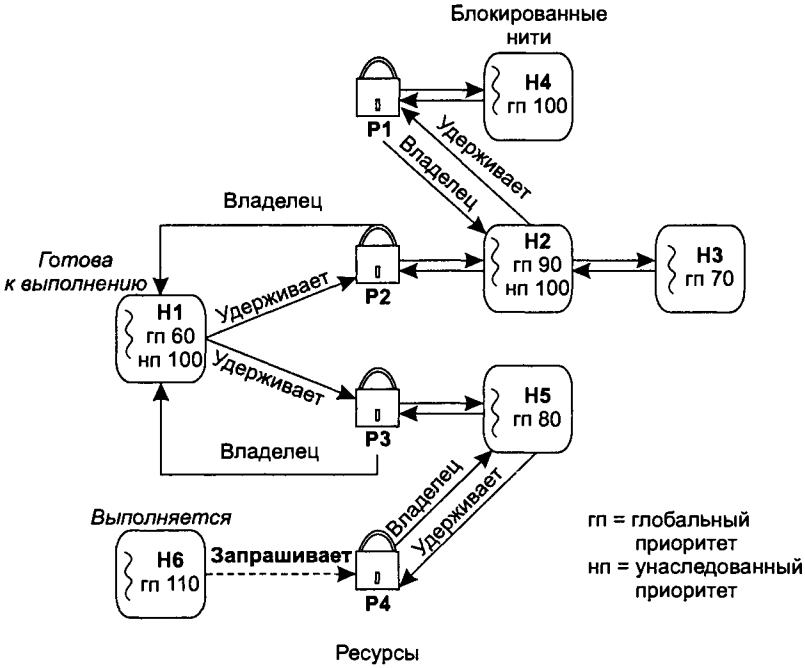


Рис. 5.9. Просмотр цепочки синхронизации

1. Нить H5, имеющая глобальный приоритет 80, но без унаследованного приоритета, является владельцем ресурса P4. Так как значение ее приоритета ниже 110, то ей устанавливается унаследованный приоритет, равный 110.
2. Нить H5 блокирована в ожидании ресурса P3, владельцем которого является H1. Нить H1 имеет глобальный приоритет 60, но ее унаследованный приоритет равен 100 (через ресурс P2). Так как это число меньше 110, функция увеличит наследуемый приоритет нити H1 до 110.
3. Поскольку нить H1 не блокирована на каком-либо ресурсе, просмотр цепочки завершается, функция возвращает управление.

После возврата из pi_willto() ядро блокирует H6 и выбирает другую нить для выполнения. Так как приоритет H1 был только что поднят до значения 110, скорее всего именно она будет немедленно назначена на выполнение. На рис. 5.10 показана ситуация после переключения контекста.

Когда нить освобождает объект, она сбрасывает унаследованный приоритет при помощи вызова pi_waive(). В некоторых случаях (например, в предыдущем примере) нить может удерживать несколько объектов. Тогда ее унаследованный приоритет будет равняться максимальному значению из всех приоритетов, наследуемых от этих объектов. Когда нить освобождает какой-либо объ-

ект, ее приоритет пересчитывается на основе оставшихся удерживаемых ею объектов. Такие сокращения наследованного приоритета могут привести к тому, что приоритет этой нити станет меньше, чем приоритет другой, готовой к выполнению нити, которая в конечном счете вытеснит первую.

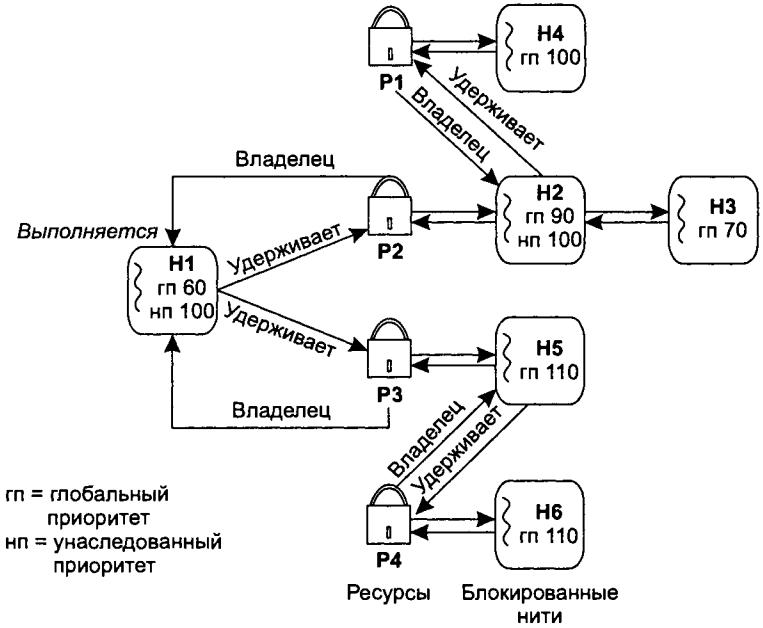


Рис. 5.10. Наследование приоритетов

5.6.6. Ограничения наследования приоритетов

Наследование приоритетов следует использовать только в случаях, когда мы знаем, какая из нитей собирается освободить ресурс. Это возможно, когда ресурс удерживается единственной известной нам нитью. В системе Solaris 2.*x* поддерживается четыре типа объектов синхронизации: взаимные исключения (mutex), семафоры, условные переменные и защелки чтения/записи (более подробно эти объекты будут рассмотрены в главе 7). При применении взаимных исключений владелец ресурса всегда известен[1]. Однако при использовании семафоров или условных переменных владелец обычно не определяется, следовательно, наследование приоритетов не используется. Это является неприятным моментом, так как условные переменные часто применяются в сочетании со взаимными исключениями, реализуя тем самым высокоуровневые конструкции синхронизации, а некоторые из последних имеют определяемых владельцев.

[1] Владельца можно определить через функцию mutex_owned. — *Прим. ред.*

Если защелка чтения/записи закрывается для записи, то владелец один, и он известен[1]. Однако ресурс может удерживаться сразу несколькими читающими нитями. Пишущая нить должна блокироваться до тех пор, пока все текущие читающие нити не освободят объект. В этом случае объект не имеет единственного владельца, а хранить указатели на всех владельцев непрактично. В системе Solaris для решения подобной проблемы предусмотрено определение *обладателя записи* (owner-of-record), которым является первая читающая нить, получившая защелку и переключившая тем самым ее в режим разрешения доступа «только на чтение». Если высокоприоритетная пишущая нить блокируется в ожидании этого объекта, нить владельца записи унаследует ее приоритет. Когда нить владельца записи освободит объект, возможна ситуация, что какие-либо другие неопределенные читающие нити еще удерживают защелку в режиме чтения. Эти нити не могут унаследовать приоритет пишущей нити. Таким образом, предлагаемое решение является ограниченным, но остается удобным, так как во многих ситуациях удержание защелки производится всего лишь одной читающей нитью.

Наследование приоритетов уменьшает время блокирования высокоприоритетного процесса, находящегося в ожидании ресурсов, удерживаемых низкоприоритетными процессами. Однако в самом худшем случае величина задержки по-прежнему остается достаточно большой, не удовлетворяя требования многих приложений реального времени. Одной из причин этого является тот факт, что цепочка блокирования способна достигать весьма больших размеров. Другой причиной может послужить наличие в высокоприоритетном процессе нескольких критических участков кода, на каждом из которых может происходить блокирование, что в сумме может привести к значительным задержкам. Этой проблеме уделяется большое внимание среди разработчиков. Появились ее альтернативные решения, например *ceiling-протокол* (ceiling protocol, см. [16]). Протокол контролирует захват ресурсов процессами для гарантии того, что высокоприоритетный процесс блокируется в ожидании ресурса, удерживаемого низкоприоритетным процессом, не более чем один раз при каждой активации. Хотя такой подход ограничивает задержки блокировки для высокоприоритетных процессов, он заставляет низкоприоритетные процессы чаще блокироваться. Он также требует априорного знания обо всех процессах системы и их требованиях относительно ресурсов. Описанные недостатки сужают область применимости протокола, оправдывая его только для небольшого круга приложений.

5.6.7. Турникеты

Ядро содержит сотни объектов синхронизации, по одному для каждой структуры данных, которую необходимо защищать отдельно. Такие объекты должны хранить огромные объемы информации, например очереди нитей, на них

[1] Тот, кто пишет. — *Прим. ред.*

блокированных. Содержание большой структуры данных для каждого объекта является расточительным, поскольку хотя в ядре находятся и сотни таких объектов, но только несколько из них используются в один конкретный момент. В системе Solaris применяется эффективное решение этой проблемы при помощи *турникетов*. Объект синхронизации содержит указатель на турникет, в котором находятся все данные, необходимые для манипулирования объектом, например очереди блокированных нитей и указатель на нить, владеющую ресурсом в текущий момент (рис. 5.11). Турникеты выделяются динамически из пула, который растет в размере по мере увеличения количества нитей в системе. Турникет предоставляется первой нити, которой необходимо заблокировать объект. Когда более не остается блокированных на объекте нитей, турникет освобождается и возвращается обратно в пул.

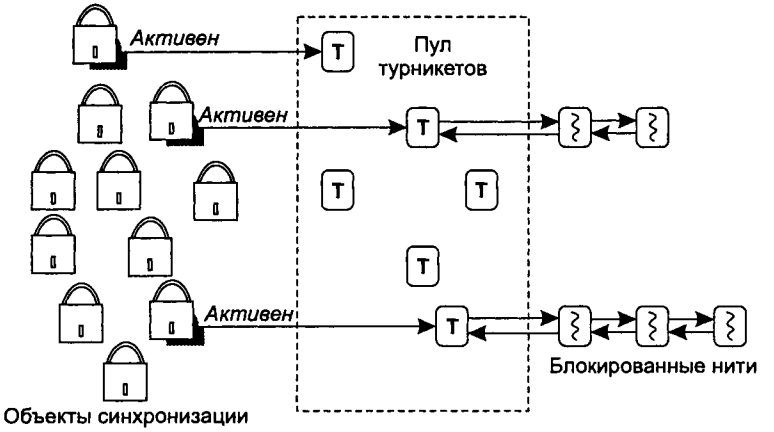


Рис. 5.11. Турникеты

В традиционных системах UNIX ядро связывает с каждым ресурсом или событием, на котором процесс может заблокироваться, особый *канал сна* (см. раздел 7.2.3). Канал обычно представляет собой адрес, связанный с ресурсом или событием. Ядро хэширует процесс в очередь сна, основанную на этом канале ожидания. Поскольку разные каналы ожидания могут отображаться в одну и ту же очередь сна, время, затрачиваемое на ее просмотр, ограничивается только общим количеством нитей системы. В ОС Solaris 2.*x* этот механизм заменен турникетами, которые ограничивают очередь сна количеством нитей, блокированных на конкретном ресурсе, что дает более разумные границы времени, затрачиваемого на обработку очереди.

Нити внутри турникета выстроены в порядке их приоритетов. Объекты синхронизации поддерживают два способа разблокировки: *сигнал*[1], который

[1] Функциональность этих сигналов не имеет никакого отношения к традиционным сигналам системы UNIX. Так как в UNIX принято использовать терминологию из нескольких источников, некоторые термины системы обладают несколькими значениями.

пробуждает определенную спящую нить, или *широковещательное сообщение* (broadcast), которое будит все нити, блокированные в ожидании ресурса. В системе Solaris сигнал будит наиболее высокоприоритетную нить очереди.

5.6.8. Анализ

В системе Solaris 2.*x* представлена сложная среда поддержки многонитевой обработки и обработки в режиме реального времени — как для однопроцессорных, так и для многопроцессорных систем. В Solaris были устранены некоторые недостатки, присущие механизму планирования в SVR4. Измерения, произведенные на Sparcstation 1, показали, что задержки обслуживания планировщиком в большинстве случаев не превышали 2 миллисекунд. Такое значение является следствием полностью вытесняющего ядра и наследования приоритетов.

Несмотря на то что в Solaris имеются средства, подходящие для большинства приложений реального времени, она является главным образом операционной системой общего назначения. Система, создаваемая специально для приложений реального времени, должна поддерживать такие дополнительные возможности, как групповое планирование процессоров, планирование по крайним срокам или планирование, основанное на приоритетах операций устройств ввода-вывода. Эти возможности будут описаны в разделе 5.9.

Рассмотрим несколько алгоритмов планирования, применяемых в коммерческих и экспериментальных вариантах системы UNIX.

5.7. Планирование в системе Mach

Mach является многонитевой операционной системой, поддерживающей многопроцессорные системы. Она разработана для применения в любых типах компьютеров — от однопроцессорных машин до массивно-параллельных систем, содержащих сотни процессоров и разделяющих между собой единое адресное пространство. Из этого следует, что планировщик ОС Mach должен подходить для всех возможных целей применения [2].

Основные понятия системы — задачи и нити — были описаны в разделе 3.7. Нить является главной единицей планирования и система планирует выполнение нитей безотносительно задачи, к которой они относятся. Такой подход снижает производительность, поскольку переключение контекста между нитями одной задачи происходит намного быстрее по отношению к переключению нитей, относящихся к различным задачам (из-за того, что в первом случае не нужно производить изменение карт управления памятью). Однако политика, которая поощряет переключение нитей внутри процесса, может противоречить целям баланса загрузки и использования системы. Более того, различия производительности двух типов контекстного переключения могут оказаться малозначительными в зависимости от применяемого аппаратного обеспечения и запущенных приложений.

Каждая нить наследует базовый приоритет планирования от задачи, к которой она относится. Этот приоритет сочетается с фактором использования процессора, который хранится и ведется отдельно для каждой нити. Система Mach уменьшает степень использования процессора для каждой нити посредством умножения ее на 5/8 каждой секунды простаивания нити. Алгоритм снижения является распределенным. Каждая нить отслеживает уровень собственного использования процессора и пересчитывает его при пробуждении после блокирования. Обработчик прерываний таймера регулирует фактор использования текущей нити. Для предупреждения зависания (из-за недостатка процессорного времени) низкоприоритетных нитей, продолжающих оставаться в очереди на выполнение без возможности пересчета своих приоритетов, каждые 2 секунды запускается внутренняя нить ядра, которая пересчитывает приоритеты всех готовых к выполнению нитей.

Назначенная на выполнение нить выполняется в течение определенного кванта времени. По исчерпании этого промежутка она может быть вытеснена другой нитью, обладающей равным или большим приоритетом. Перед тем как израсходуется начальный квант у текущей нити, ее приоритет может быть понижен по отношению к другим готовым к выполнению нитям. Однако в системе Mach такие изменения не приведут к переключению контекста раньше, чем нить исчерпает свой квант. Благодаря этой особенности сокращается общее число переключений контекста, что прямо сказывается на равновесии в использовании системы. Текущая нить может быть вытеснена, если более высокоприоритетная нить станет готовой к выполнению, даже если квант текущей нити не был до конца выработан.

В системе Mach поддерживается *автоматическое планирование* (handoff scheduling), посредством которого нить может напрямую передавать процессор другой нити без поиска очередей выполнения. Подсистема межпроцессного взаимодействия (IPC) использует эту технологию для передачи сообщений: если нить уже находится в режиме ожидания сообщения, посылающая нить передает ей процессор непосредственно. Такой подход увеличивает производительность работы вызовов IPC.

5.7.1. Поддержка нескольких процессоров

Как уже говорилось, система Mach функционирует на самых различных аппаратных архитектурах, от небольших персональных компьютеров до машин, имеющих сотни процессоров. Планировщик системы обладает несколькими средствами, позволяющими эффективно управлять процессорами системы.

Для вытеснения в Mach не применяются межпроцессорные прерывания. Предположим, что некоторое событие на одном из процессоров привело к появлению готовой к выполнению нити, имеющей приоритет больший, чем у другой нити, выполняющейся на другом процессоре. Нить с меньшим приоритетом не будет вытеснена до тех пор, пока другой процессор обрабатыва-

ет прерывание таймера или иное событие, относящееся к планированию. Отсутствие межпроцессорного вытеснения не оказывает отрицательного влияния на режим разделения времени, но оно, тем не менее, может сказываться на эффективности реакции системы по отношению к приложениям реального времени.

Система Mach позволяет пользователям управлять выделением процессоров различным задачам путем создания *наборов процессоров*, каждый из которых может содержать 0 и более процессоров. Каждый процессор относится к одному из таких наборов, и он также может быть перемещен из одного набора в другой. Каждой задаче или нити назначается определенный набор процессоров, который может быть изменен в любой момент времени. Однако такой возможностью обладают только привилегированные задачи, которым разрешается назначать процессоры, задачи и нити наборам процессоров.

Нить может выполняться на одном из процессоров из выделенного ей набора. Назначение задачи набору процессоров делает его назначаемым по умолчанию всем новым нитям этой задачи. Задача наследует набор от своего предка, а *первоначальной задаче* (initial task) предоставляется *набор процессоров, принятый по умолчанию*. Исходно такой набор состоит из всех процессоров системы, и на этом наборе выполняются внутренние нити ядра и демоны.

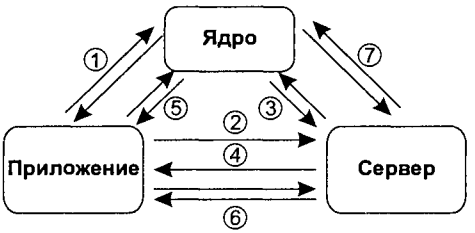


Рис. 5.12. Выделение процессоров в Mach

Выделение процессоров может быть выполнено при помощи программы-сервера прикладного уровня (выполняющейся как привилегированное задание), которая и определяет политику присвоения. На рис. 5.12 показана типичная схема взаимодействий между приложением, сервером и ядром. Приложение выделяет набор процессоров всем своим нитям. Сервер назначает процессоры для этого набора. Последовательность действий при этом описана ниже.

1. Приложение запрашивает ядро о выделении для себя набора процессоров.
2. Приложение запрашивает сервер о процессорах для этого набора.
3. Сервер запрашивает ядро о назначении процессоров набору.
4. Сервер отвечает приложению, подтверждая, что процессоры были выделены.

5. Приложение запрашивает ядро о назначении своих нитей этому набору.
6. Приложение использует процессоры и уведомляет сервер, когда завершает свою работу.
7. Сервер переназначает процессоры.

Такой подход позволяет добиться потрясающей гибкости в управлении процессорами, особенно для массивно-параллельных систем с большим количеством процессоров. Например, существует возможность назначить их некоторое количество одной задаче или группе задач, гарантируя тем самым предоставление им части доступных ресурсов независимо от общей загруженности системы. В крайних случаях приложение может добиваться выделения по одному процессору для каждой своей нити. Такой подход называется *групповым планированием* (gang scheduling).

Применение группового планирования удобно для приложений, требующих *барьерной синхронизации*. Такие приложения создают несколько нитей, которые функционируют независимо друг от друга до достижения некой точки синхронизации, называемой барьером. Каждая нить, достигнувшая барьера, должна ожидать, пока к нему приблизятся остальные. После того как все нити сведут свою работу к этой точке, приложение может запустить некоторый однонитевый код и затем создать другой набор нитей, повторяющих действия предыдущего набора.

Для того чтобы подобные приложения функционировали оптимально, задержки при достижении барьера должны быть минимальными. Это требует, чтобы все нити достигали барьера примерно в одно и то же время. Групповое планирование позволяет приложению запустить нити одновременно, предоставив каждой из них отдельный процессор. Описанный подход сводит к минимуму задержки при достижении барьера.

Групповое планирование также подходит для мелкомодульных приложений, нити которых часто взаимодействуют между собой. В таких приложениях вытеснение одной из нитей способно привести к блокированию других нитей, требующих взаимодействия с ней. Недостатком предоставления каждой нити отдельного процессора является то, что при блокировании нити процессор не может быть использован.

Процессоры системы могут быть неодинаковыми. Некоторые из них могут оказаться более быстрыми, чем другие, некоторые могут содержать блок вычислений с плавающей точкой и т. д. Возможность составления процессоров в наборы позволяет использовать определенные процессоры для выполнения свойственной им задачи. Например, процессоры с блоком вычислений с плавающей точкой логично назначать только нитям, которым необходима максимальная производительность при работе с вещественными числами.

Кроме всего перечисленного, нить может быть временно ограничена определенным процессором. Такая возможность поддерживается, в первую очередь, для совместимости части кода системы Mach с UNIX в области непа-

раллельной (небезопасной в многопроцессорной системе) обработки. Эта часть кода выполняется на одном процессоре, называемом *главным процессором* (master processor). Каждый процессор имеет локальную очередь выполнения, а каждый набор — глобальную, используемую совместно всеми процессорами набора. Сначала процессоры проверяют свои локальные очереди, отдавая таким образом предпочтение ограниченным данным процессором нитям (или неограниченным использованием данного процессора нитям, обладающим высокими приоритетами). Такое решение привнесло максимальную производительность для непараллельного кода UNIX, защищая от эффекта «бутылочного горлышка».

5.8. Планировщик реального времени Digital UNIX

Планировщик операционной системы Digital UNIX поддерживает приложения, выполняющиеся как в режиме разделения времени, так и в режиме реального времени [5]. Он совместим с интерфейсом POSIX 1003.1b [8], определяющим программное расширение реального времени. Несмотря на факт, что Digital UNIX произошла от ОС Mach, ее планировщик был полностью переработан. Он поддерживает следующие классы планирования:

- SHED_OTHER, или класс разделения времени.
- SHED_FIFO, или класс FIFO («первым вошел, первым вышел»).
- SHED_RR, или класс карусельного обслуживания.

Для установки класса планирования и приоритета процесса приложение может вызвать sched_setscheduler. По умолчанию используется класс разделения времени, в котором приоритеты изменяются динамически на основе величины любезности и уровня использования процессора. Оставшиеся два класса используют постоянные приоритеты. Процессы, задействующие политику класса SHED_FIFO, не имеют определенного кванта времени и выполняются до тех пор, пока сами не освободят процессор или не будут вытеснены более высокоприоритетным процессом. Классам разделения времени и карусельного обслуживания определен квант времени, который влияет на диспетчеризацию процессов, обладающих одинаковым приоритетом. Когда процессы, принадлежащие какому-либо из этих двух классов, исчерпывают выделенный им квант, они помещаются в конец списка процессов того же приоритета. В случае отсутствия готовых к выполнению процессов, обладающих одинаковым или более высоким приоритетом по сравнению с текущим, процесс будет продолжать свою работу.

Планировщик всегда выбирает для выполнения наиболее высокоприоритетный процесс. Каждый процесс имеет приоритет в диапазоне 0–63, где меньшие числа соответствуют меньшим приоритетам. Планировщик для каждого

приоритета содержит упорядоченную очередь и выбирает процесс из начала наивысшей непустой очереди. Когда блокированный процесс становится готовым к выполнению или текущий процесс освобождает процессор, то такие процессы обычно помещаются в конец очереди своего приоритета. Исключительным случаем является вытеснение процесса до того, как он исчерпает выделенный ему квант времени. В этом случае процесс возвращается в начало своей очереди, что позволяет ему завершить использование кванта времени до того, как начнут свое выполнение другие процессы, обладающие тем же приоритетом.

Приоритеты подразделяются на три перекрывающихся класса, которые позволяют увеличить гибкость. Назначение приоритетов процессам регулируется следующими правилами:

- Процессы класса разделения времени имеют приоритеты в диапазоне от 0 до 29. Для увеличения приоритетов выше 19 требуются привилегии суперпользователя.
- Пользователи управляют приоритетами процессов класса разделения времени посредством изменения величины их «любезности». Значение величины «любезности» находится в диапазоне от −20 до +20, где меньшие числа указывают на более высокий приоритет (для обратной совместимости). Отрицательные величины, соответствующие приоритетам диапазона 20–29, может задавать только суперпользователь системы.
- Фактор использования процессора уменьшает приоритеты процессов класса разделения времени в зависимости от полученного ими количества процессорного времени.
- Системные процессы обладают фиксированными приоритетами в диапазоне 20–31.
- Процессам с фиксированными приоритетами может быть назначен любой приоритет в диапазоне 0–63. Однако для назначения приоритетов выше 19 требуются привилегии суперпользователя. Процессы реального времени имеют приоритеты в диапазоне от 32 до 63, поскольку системные процессы не должны их вытеснять.

Вызов sched_setparam применяется для изменения приоритетов процессов классов FIFO и карусельного обслуживания. Вызов sched_yield перемещает вызвавший его процесс в конец очереди его приоритета, что эквивалентно освобождению процессора для любого готового к выполнению процесса, обладающего тем же приоритетом. Если таких процессов не обнаружено, процесс, вызывавший sched_yield, продолжит выполнение.

5.8.1. Поддержка нескольких процессоров

Система Digital UNIX позволяет эффективно использовать такое свойство систем, как многопроцессорность, посредством настройки своего планировщика для оптимизации переключений контекста и использования кэша [6].

В идеальном случае планировщик стремится выполнять высокоприоритетные готовые к выполнению нити на всех доступных процессорах. Такая политика требует от планировщика поддержания глобального набора очередей выполнения, разделяемого всеми процессорами. Это может стать причиной эффекта «бутылочного горлышка», когда все процессоры будут стремиться получить монопольный доступ (то есть блокировать доступ другим) к этим очередям. Более того, когда нить начинает свое выполнение, кэши процессора заполняются ее данными и инструкциями. И если на короткий промежуток времени эта нить будет вытеснена и затем снова назначена на выполнение, то по возможности она должна продолжаться на том же процессоре, так как при этом она получит определенный выигрыш из-за наличия в кэшах процессора ранее размещенных данных и инструкций.

Для согласования вышеуказанных моментов в системе Digital UNIX для нитей режима разделения времени используется политика *мягкого сродства* (soft affinity). Такие нити хранятся в локальных очередях выполнения процессора, следовательно, они вновь назначаются на выполнение на том же самом процессоре, что и раньше. Это также сокращает борьбу за очереди выполнения. Планировщик следит за очередями для каждого процессора и предотвращает возможный дисбаланс путем перемещения нитей из очереди более загруженного процессора в очередь менее занятого.

Нити с фиксированными приоритетами назначаются на выполнение из глобальной очереди, потому что они должны начать свое выполнение как можно быстрее. По возможности ядро отправляет их на тот же самый процессор, который использовался ими в предыдущий раз. В завершение следует также отметить, что система Digital UNIX обеспечивает вызов `bind_to_cpu`, который принуждает нить к выполнению только на определенном процессоре. Эта возможность полезна для кода, функционирование которого на более чем одном процессоре небезопасно.

Планировщик системы Digital UNIX предоставляет совместимый с POSIX интерфейс планирования реального времени. Однако он не обладает многими возможностями, имеющимися в Mach и SVR4. Он также не обеспечивает интерфейс для выделения наборов процессоров или механизм «ручного» планирования. Ядро Digital UNIX является невытесняющим и не имеет никаких средств для контроля за инверсией приоритетов.

5.9. Другие реализации планирования

Реализация алгоритмов планирования системы зависит, главным образом, от требований приложений, которые будут работать в данной ОС. В некоторых системах работают критичные ко времени программы и приложения реального времени, в других — крупные, работающие в режиме разделения времени приложения, в третьих — и те и другие. В иных системах выполняется

огромное количество процессов, для которых существующие методики планирования не соответствуют требованиям по масштабированию. Такое положение дел побудило к созданию ряда планировщиков, которые частично были реализованы в различных вариантах UNIX. Их описанию и посвящен этот раздел книги.

5.9.1. Планирование справедливого разделения

Планировщик справедливого разделения (fair-share) предоставляет фиксированный объем разделяемых ресурсов процессора каждой *разделяющей группе* процессов. Такие группы могут состоять из единичного процесса, всех процессов одного пользователя, всех процессов сеанса входа в систему и т. д. Выбор распределения процессорного времени между *разделяющими группами* доступен только суперпользователю системы. Ядро отслеживает использование процессора для выбора схемы его распределения. Если одна из групп не использовала выделенную ей часть ресурсов, то их оставшаяся часть обычно делится между остальными группами пропорционально изначально предоставленным им ресурсам.

Такой подход дает каждой разделяющей группе прогнозируемый объем процессорного времени, который не зависит от общей загруженности системы. Это особенно полезно в средах, где время вычисления является строго расписанным ресурсом, поскольку ресурсы могут выделяться пользователям фиксированными объемами. Это также пригодно для гарантии предоставления необходимых ресурсов критичным к ним приложениям в системах разделения времени. Одна из реализаций планировщика равного разделения описана в [7].

5.9.2. Планирование по крайнему сроку

Многие приложения реального времени должны отвечать на события в течение определенного ограниченного периода. Например, мультимедийный сервер может передавать видеокадры клиенту каждые 33 миллисекунд. Если сервер читает данные с диска, то он может установить крайний срок завершения операции чтения. Если в течение этого времени операция не успеет завершиться, кадр будет задержан. Механизм назначения крайних сроков также применим для запросов на ввод-вывод или вычислений. В последнем случае нить может запросить выделения определенного объема процессорного времени, которое должно быть ей предоставлено до окончания установленного крайнего срока.

Производительность описываемых видов приложений повысится при использовании технологии *планирования по крайнему сроку*. Основным принципом этого метода является динамическое изменение приоритетов, увеличение которых происходит при приближении к конечному сроку. Примером реализации подобного алгоритма является планировщик Ferranti-Originated

Real-Time Extensions to UNIX (FORTUNIX), описанный в [3]. Его алгоритм определяет четыре уровня приоритетов:

- Жесткие приоритеты реального времени для крайних сроков, которые всегда должны быть соблюдены при удовлетворении чего-либо.
- Мягкие приоритеты реального времени, при которых требования удовлетворить что-либо до наступления крайнего срока должны реализовываться с наибольшей вероятностью, однако при этом случаи неукладывания в граничные сроки возможны.
- Приоритеты разделения времени без определенных конечных сроков, но с требованием, чтобы были установленны разумные пределы времени реакции.
- Приоритеты пакетных заданий, для которых крайние сроки выражаются в часах (а не в миллисекундах).

Система осуществляет планирование процессов в соответствии с их уровнями приоритетов. Например, процесс реального времени с мягким приоритетом будет выполнен только в случае отсутствия готовых к выполнению процессов с жесткими приоритетами. Процессы классов 1, 2 и 4 выполняются в соответствии с их крайними сроками: процесс, обладающий наименьшим оставшимся временем, будет назначен на выполнение первым. Процессы перечисленных выше классов будут выполняться до тех пор, пока они не завершатся или не будут блокированы, если только не появится готовый к выполнению процесс из более старшего класса или того же класса, но с меньшим временем, оставшимся до достижения крайнего срока. Процессы режима разделения времени планируются на выполнение в традиционной манере UNIX на основе их приоритетов, зависящих от фактора любезности и использования процессора.

Планирование по крайнему сроку подходит для систем, в которых работают процессы с известными заранее требованиями по времени реакции. Подобная схема приоритетов применима для планирования запросов ввода-вывода к диску и т. д.

5.9.3. Трехуровневый планировщик

Планировщики UNIX не способны гарантированно обеспечить соблюдение требований приложений реального времени, но в то же время позволяют осуществлять произвольную смешанную рабочую нагрузку. Основной причиной недостатков является отсутствие *управления доступом* (admission control). Не существует ограничений на количество или типы приложений реального времени и задач общего назначения, запускаемых в системе. Система позволяет всем процессам состязаться за ресурсы, не контролируя этот процесс. Забота о том, чтобы в системе не возникало перегрузок, лежит фактически на ее пользователях.

В работе [14] описывается трехуровневый планировщик, применяющийся в системах реального времени для мультипротокольных файлов и медиа-серверов. Планировщик поддерживает три класса служб: изохронный, реального времени и общего применения. Изохронный класс подразумевает периодические действия, такие как передача видеокадров, через определенные интервалы с минимальным *дрожанием изображения*, или колебаниями. Класс реального времени требуется для апериодических задач, нуждающихся в ограниченной задержке обслуживания планировщиком. Последний класс служб общего назначения используется для низкоприоритетных фоновых задач. Планировщик гарантирует, что выполнение таких задач не будет оказывать влияния на работу изохронных нитей.

Несколько слов о том, как планировщик проводит политику управления доступом. Перед приемом нового видеопотока резервируются все необходимые для этого потока ресурсы, которые включают в себя часть процессорного времени, пропускную способность при операциях с диском и сетевым контроллером. Если серверу не удается зарезервировать ресурсы, то он не разрешит обработку запроса. Каждая служба реального времени за каждую свою активацию может обрабатывать ограниченное число *рабочих блоков*. Например, сетевой драйвер перед тем, как освободить процессор, вправе обработать лишь определенное количество входящих сообщений. Планировщик также по отдельности устанавливает фиксированный объем всех ресурсов для заданий общего характера, по отношению к которым не используется управление доступом. Такой подход защищает задания общего применения от зависания на сильно загруженных системах.

Для реализации описанной политики система планирует не только процессорное время, но и дисковые и сетевые операции. Запросам на ввод-вывод назначаются приоритеты на основе инициирующей их задачи. Обрабатываются в первую очередь высокоприоритетные запросы. Резервирование пропускной способности всего тракта из всех имеющихся ресурсов дает возможность гарантировать, что система будет удовлетворять требованиям всех допущенных видеопотоков. Такой подход также гарантирует продолжение выполнения низкоприоритетных задач даже в случае максимальной загруженности системы.

В трехуровневом алгоритме планирования общецелевые задания являются полностью вытесняемыми. Задания реального времени могут вытесняться изохронными заданиями только в четко определенных точках вытеснения, которые обычно находятся в конце выполнения каждого блока работы. Изохронные задачи используют алгоритм планирования *постоянного отношения* (rate-monotonic, описан в [11]). Каждая такая задача обладает определенным фиксированным приоритетом планирования, зависящим от ее периода. Чем меньше период, тем выше будет приоритет. Высокоприоритетная изохронная задача может вытеснить задачу с более низким приоритетом только в точке вытеснения. В работе [15] показано, что алгоритм постоянного отношения оптимален для планирования периодических задач с фиксированными приоритетами.

Еще одной проблемой серверов, работающих на традиционных системах UNIX, является их неспособность справляться с перенасыщением, вызванным большой нагрузкой. Системы UNIX производят основную часть обработки входящих сетевых запросов на уровне прерываний. Если входящий трафик слишком высок, система тратит больше времени на обработку прерываний, оставляя слишком малую его долю на обслуживание запросов. Если входная загрузка превышает определенный критический уровень, резко снижается пропускная способность сервера. Эта проблема известна под названием *петли приема* (receive livelock). Применение трехуровневого планировщика решает проблему путем перемещения всей обработки сетевых запросов на уровень задач реального времени, для которых ограничен объем обрабатываемого за один прием трафика. Если входящий трафик превысит критическую отметку, сервер отбросит избыточные запросы, что даст возможность продолжения обработки запросов, которые были приняты. Следовательно, после снижения трафика пропускная способность вновь приблизится к постоянной величине, вместо того чтобы снизиться.

5.10. Заключение

В этой главе были рассмотрены несколько различных архитектур планирования и показано, как они влияют на действия системы по отношению к различным типам приложений. Поскольку вычислительные системы применяются в самых различных областях деятельности, каждая из которых обладает определенным набором требований, то ни один планировщик не может идеально устраивать все системы. Планировщик ОС Solaris 2.*x* подходит для многих типов приложений и предоставляет механизм, позволяющий динамически добавлять новые классы планирования, удовлетворяющие требованиям каких-либо особых приложений. Он лишен некоторых возможностей, таких как поддержка ввода-вывода нитей реального времени или управляемое пользователем планирование работы с диском, однако он является более совершенным, чем традиционный планировщик UNIX. Другие методы планирования, которые были рассмотрены в этой главе, применимы для особых областей приложений, например для параллельной обработки или мультимедиа.

5.11. Упражнения

1. Почему отложенные вызовы не обрабатываются непосредственно обработчиком прерываний таймера?
2. В каких ситуациях для обработки отложенных вызовов использование временных колес будет более эффективным, чем алгоритм, применяемый в системе 4.3BSD?

3. В чем преимущества и недостатки использования в отложенных вызовах относительных интервалов времени по сравнению с абсолютными их значениями?
4. Почему в системе UNIX более предпочтительны процессы ввода-вывода, нежели вычислительные процессы?
5. В чем преимущества объектно-ориентированного интерфейса планировщика системы SVR4? В чем недостатки этой методики?
6. Почему значения slpret и lwait всегда больше, чем значения tgexp в каждой строке таблицы параметров диспетчера (см. табл. 5.1)?
7. Почему процессам реального времени даются приоритеты выше, чем процессам, работающим в режиме ядра? В чем недостатки такого распределения приоритетов?
8. Почему планирование по событиям наиболее удобно для приложений ввода-вывода и интерактивных задач?
9. В разделе 5.5.6 был упомянут эксперимент, описанный в [13]. Какой будет эффект в том случае, если X-server'у, программе для просмотра видео и интерактивному заданию будут назначены приоритеты класса реального времени, а пакетное задание получит приоритет класса разделения времени?
10. Предположим, что процесс освобождает ресурс, который ожидают сразу несколько других процессов. Что рациональнее: разбудить все ожидающие процессы или же только один из них? Если будить один процесс, то какой из них выбрать?
11. Групповое планирование подразумевает, что каждая нить выполняется на отдельном процессоре. Какие действия предпримет приложение, требующее барьерной синхронизации, в том случае, если доступных процессоров окажется меньше, чем готовых к выполнению нитей? Могут ли в такой ситуации нити, достигшие барьера, находиться в режиме активного ожидания прихода остальных?
12. Какие существуют методы поддержки приложений реального времени в системе Solaris 2.*x*? В каких случаях их применение неадекватно?
13. Почему планирование по крайнему сроку не подходит для традиционной операционной системы UNIX?
14. Перечислите характеристики процессов реального времени. Приведите несколько примеров периодических и непериодических приложений реального времени.
15. Для уменьшения времени реакции и задержек обслуживания планировщиком можно просто использовать более мощный процессор. Какие различия существуют между системами реального времени и высокопроизводительными системами? Может ли быть такое, что система

с меньшей общей производительностью окажется наиболее подходящей для приложений реального времени?

16. Какие существуют различия между жесткими и мягкими требованиями для классов реального времени?
17. Чем важна для систем реального времени поддержка управления доступом?

5.12. Дополнительная литература

1. American Telephone and Telegraph, «UNIX System V Release 4 Internals Students Guide», 1990.
2. Black, D. L., «Scheduling Support for Concurrency and Parallelism in the Mach Operating System», IEEE Computer, May 1990, pp. 35–43.
3. Bond, P. O., «Priority and Deadline Scheduling on Real-Time UNIX», Proceedings of the Autumn 1988 European UNIX Users' Group Conference, Oct. 1988, pp. 201–207.
4. Digital Equipment Corporation, «VAX Architecture Handbook», Digital Press, 1986.
5. Digital Equipment Corporation, «DEC OSF/1 Guide to Realtime Programming», Part No. AA–PS33C–TE, Aug. 1994.
6. Denham, J. M., Long, P., and Woodward, J. A., «DEC OSF/1 Version 3.0 Symmetric Multiprocessing Implementation», Digital Technical Journal, Vol. 6, No. 3, Summer 1994, pp. 29–54.
7. Henry, G. J., «The Fair Share Scheduler», AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Oct. 1984, pp. 1845–1857.
8. Institute for Electrical and Electronic Engineers, «POSIX P1003.4b, Real-Time Extensions for Portable Operating Systems», 1993.
9. Khanna, S., Sebree, M., and Zolnowsky, J., «Realtime Scheduling in SunOS 5.0», Proceedings of the Winter 1992 USENIX Technical Conference, Jan. 1992.
10. Lampson, B. W. and Redell, D. D., «Experiences with Processes and Monitors in Mesa», Communications of the ACM, Vol. 23, No. 2, Feb 1980, pp. 105–117.
11. Liu, C. L., and Layland, J. W., «Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment», Journal of the ACM, Vol. 20, No. 1, Jan. 1973, pp. 46–61.
12. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3 BSD UNIX Operating System», Addison-Wesley, Reading, MA, 1989.

13. Nieh, J., «SVR4 UNIX Scheduler Unacceptable for Multimedia Applications», Proceedings of the Fourth International Workshop on Network and Operating Support for Digital Audio and Video, 1993.
14. Ramakrishnan, K. K., Vaitzblit, L., Gray, C. G., Vahalia, U., Ting, D., Tzelnic, P., Glaser, S., and Duso, W. W., «Operating System Support for a Video-on-Demand File Server», Multimedia Systems, Vol. 3, No. 2, May 1995, pp. 53–65.
15. Sha, L., and Lehoczky, J. P., «Performance of Real-Time Bus Scheduling Algorithms», ACM Performance Evaluation Review, Special Issue, Vol. 14., No. 1, May 1986.
16. Sha, L., Rajkumar, R., and Lehoczky, J. P., «Priority Inheritance Protocols: An Approach to Real-Time Synchronization», IEEE Transactions on Computers, Vol. 39, No. 9, Sep. 1990, pp. 1175–1185.
17. Straathof, J. H., Thareja, A. K., and Agrawala, A. K., «UNIX Scheduling for Large Systems», Proceedings of the Winter 1986 USENIX Technical Conference, Jan. 1986.
18. Varghese, G., and Lauck, T., «Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility», Eleventh ACM Symposium on Operating Systems Principles, Nov. 1987, pp. 25–38.

Глава 6
Межпроцессное взаимодействие

6.1. Введение

В сложных программных средах часто применяется несколько взаимодействующих процессов, выполняющих взаимосвязанные действия. Такие процессы должны иметь возможность общаться и разделять между собой общие ресурсы и данные. Для того чтобы сделать это возможным, ядро системы должно поддерживать определенные механизмы. Такие механизмы получили название *межпроцессного взаимодействия* (interprocess communications, или IPC). В этой главе будут описаны средства IPC, представленные в основных вариантах системы UNIX.

Взаимодействие процессов производится для нескольких определенных целей:

- **Передача данных.** Одному процессу иногда необходимо передать данные другому процессу. Количество передаваемых данных может варьироваться от одного байта до нескольких мегабайтов.
- **Совместное использование информации.** Нескольким процессам необходимо обрабатывать разделяемые данные таким образом, чтобы их изменение одним из них сразу же становилось видимым остальным процессам, участвующим в их совместном использовании.
- **Уведомление о событиях.** Одному процессу иногда требуется уведомлять другой процесс (или набор процессов) о возникновении какого-либо события. Например, при завершении работы процесса об этом необходимо проинформировать всех его потомков. Получатель может предупреждаться асинхронно, в таком случае нужно прерывать нормальную работу. Альтернативным вариантом является ожидание получателем уведомления.
- **Совместное использование ресурсов.** Хотя ядро системы обычно предоставляет определенную семантику для выделения ресурсов, она может оказаться неподходящей для некоторых приложений. Набор взаимодействующих процессов иногда нуждается в определении собственного

протокола доступа к определенным ресурсам. Подобные правила обычно реализуются на основе схемы блокировки и синхронизации, которая строится поверх основного набора средств, предоставляемых ядром.

- **Управление процессами.** Некоторым процессам (например, отладчикам) необходимо полное управление выполнением других процессов. Контролирующий процесс может перехватывать все исключительные состояния и аппаратные прерывания целевого процесса и уведомляться обо всех изменениях его состояния.

В системах UNIX предлагаются различные механизмы IPC. Эта глава начинается с описания общего набора средств, которые имеются во всех реализациях UNIX. Это — сигналы, программные каналы и трассировка процессов. Затем мы расскажем о средствах, имеющихся в System V IPC. В конце главы будет рассмотрено взаимодействие между процессами на основе сообщений, поддерживаемое в системе Mach, в которой предлагается богатый набор средств, объединенных единой унифицированной структурой.

6.2. Универсальные средства IPC

Первая внешняя реализация системы UNIX поддерживала три различных средства, которые можно использовать для взаимодействия процессов: сигналы, каналы и трассировку процессов [14][1]. Все перечисленные механизмы являются общими для различных вариантов ОС UNIX. Сигналы и каналы уже были описаны ранее, в этой главе мы остановимся только на том, как их можно использовать для межпроцессного взаимодействия.

6.2.1. Сигналы

Обычно сигналы применяются для уведомления процессов о возникновении асинхронных событий. Изначально они были изобретены для обработки ошибок, но могут быть использованы и в качестве примитивных механизмов IPC. Современные системы UNIX распознают 31 и более различных сигналов. Большинство из них имеют определенные назначения, но по крайней мере два — `SIGUSR1` и `SIGUSR2` — вправе использоваться приложениями по их собственному усмотрению. Процесс может послать сигнал другому процессу (или процессам) при помощи системного вызова `kill` или `killpg`. Кроме этого, сигнал может быть сгенерирован ядром в ответ на различные события, происходящие в системе. Например, при нажатии комбинации клавиш `Ctrl+C` на терминале ядро посылает сигнал `SIGINT` приоритетному процессу.

[1] Первые системы UNIX от Bell Telephone Laboratories не поддерживали таких средств. Например, программные каналы были разработаны Д. Мак-Илроем и К. Томпсоном и впервые появились в Version 3 UNIX (см. [12]).

Каждый сигнал приводит к определенным действиям, по умолчанию это завершение процесса. Процесс может указать альтернативное действие, происходящее при получении любого сигнала, предоставив системе функцию обработки сигнала. При вырабатывании сигнала ядро прерывает выполнение процесса, который должен ответить на него запуском обработчика. После завершения обработки сигнала процесс может продолжить работу в обычном режиме.

При помощи сигналов процессы уведомляются о произошедших асинхронных событиях и реагируют на них. Однако сигналы могут также использоваться для синхронизации. Процесс может вызвать sigpause для ожидания прибытия сигнала. В первых реализациях системы UNIX совместное использование ресурсов и протоколы блокировки многих приложений базировались на сигналах.

Изначально сигналы разрабатывались для обработки ошибок, например ядро транслировало аппаратные ошибки, такие как *деление на ноль* или *неверные инструкции* в сигналы. Если процесс не имел собственного обработчика для таких ошибок, ядро завершало его работу.

При применении сигналов как механизма взаимодействия процессов существуют несколько ограничений, связанных с тем, что обработка сигналов является затратным действием. Сначала отправитель вызывает системную функцию, после чего ядро прерывает работу получателя и производит интенсивные действия над его стеком, так как ему нужно загрузить обработчик и позже продолжить выполнение прерванного процесса. Более того, сигналы обладают малой пропускной способностью. Во-первых, существует всего 31 сигнал (в системах SVR4 и 4.3BSD, некоторые реализации типа AIX поддерживают большее число сигналов). Во-вторых, сигналы могут нести только ограниченный объем информации. Не существует способа отправки дополнительных данных или входных аргументов при посылке сигналов, создаваемых пользователем[1]. Сигналы применимы для уведомления о событиях, но остаются недостаточными для более сложных взаимодействий.

Подробнее сигналы обсуждались в главе 4.

6.2.2. Каналы

В традиционных реализациях UNIX программные каналы (pipe)[2] представляют собой однонаправленный неструктурированный поток данных фиксированного максимального размера, работающий по принципу FIFO («первым вошел,

[1] Сигналы, генерируемые ядром в ответ на аппаратные сбои, возвращают дополнительную информацию через структуру siginfo, передаваемую обработчику.

[2] На неименованных программных каналах, иначе трубах, реализуется технология конвейеризации, почему их еще называют конвейерами. — *Прим. ред.*

первым вышел»)[1]. Отправители добавляют данные в конец канала, получатели извлекают их из его начала. После того как данные будут прочитаны, они сразу же удаляются из канала и больше недоступны другим получателям. Программные каналы представляют собой простейший механизм управления нитями. Процесс, осуществляющий попытку чтения из пустого канала, будет приостановлен до тех пор, пока в канале не появятся какие-либо данные. Точно так же, если процесс попытается записать в заполненный программный канал, то он будет приостановлен до того момента, пока иной процесс не прочтет данные из канала, тем самым освободив его.

Рис. 6.1. Движение потока данных по программному каналу

Для создания программного канала применяется системный вызов pipe, который возвращает два дескриптора файла: один для чтения и один для записи. Эти дескрипторы наследуются процессами-потомками, таким образом разделяя между собой доступ к файлу. Следовательно, в каждый программный канал могут записывать данные и читать из него несколько процессов (рис. 6.1). Каждый процесс вправе как читать, так и записывать информацию, а также производить оба действия сразу. Однако в большинстве случаев программный канал используется двумя процессами, на двух его концах. Операции ввода-вывода над каналами очень похожи на аналогичные операции над файлами. Они производятся через системные вызовы read и write с помощью дескриптора канала. Процесс иногда может и не знать о том, что работает с каналом, а не с обычным файлом.

Различные приложения, такие как командные интерпретаторы, управляют программными каналами, считая, что у тех имеется только один отправитель (записывающий данные) и один получатель (считывающий данные), тем самым образуется однонаправленный поток данных. Наиболее общим применением программного канала является перенаправление вывода одной программы на вход другой. Для этого пользователи объединяют две программы в конвейер при помощи оператора |.

---

[1] В традиционных системах, например в SVR2, программные каналы реализованы в файловой системе. Они используют поля блоков прямой адресации в индексных дескрипторах файлов (inode, см. раздел 9.2.2) для размещения блоков данных в канале. Это ограничивает размер конвейера до десяти блоков. В современных системах UNIX остается такое ограничение несмотря на то, что реализация каналов в них иная.

С точки зрения межпроцессного взаимодействия программные каналы являются эффективным способом передачи данных от одного процесса другому. Однако они обладают несколькими ограничениями:

- так как чтение данных из канала приводит к их удалению, он не может быть использован для широковещательной передачи информации нескольким адресатам;
- данные канала интерпретируются как поток байтов, границы сообщения заранее не известны. Если отправитель посылает через канал несколько объектов различной длины, получатель не имеет возможности определять, как много объектов ему было передано, он также не знает, где заканчивается один объект и начинается другой[1];
- когда данные из канала считываются несколькими процессами, отправитель не в состоянии передать данные какому-то определенному получателю. Точно так же при наличии нескольких отправителей не существует способа определения, какой из них отослал данные[2].

Реализация программных каналов может быть осуществлена различными способами. В традиционных вариантах (например, в SVR2) для этого применяются механизмы файловой системы и ассоциация *индексного дескриптора файла* (inode) и *элемента таблицы файлов* для каждого канала. Во многих реализациях системы, основанных на BSD, применяются сокеты. В ОС SVR4 используются двунаправленные каналы STREAMS, которые будут описаны подробнее в следующем разделе.

В ОС System V UNIX и других коммерческих вариантах UNIX представлены *файлы FIFO*, называемые также *именованными каналами* (named pipe). Они отличаются от обычных неименованных каналов способами создания и доступа. Пользователь создает файл FIFO при помощи системного вызова `mknod`, передавая ему имя файла и режим его создания. Поле режима описывает тип файла S_IFIFO и обычные правила доступа к нему. После этого процесс, обладающий соответствующими полномочиями, может открывать файл FIFO и производить с ним операции чтения и записи. Семантика чтения-записи для файлов FIFO похожа на аналогичные операции над программными неименованными каналами и будет более подробно описана в разделе 8.4.2. Файл FIFO существует до тех пор, пока не будет отсоединен принудительно, даже если у него не останется ни одного активного отправителя или получателя.

Файлы FIFO обладают несколькими преимуществами по сравнению с программными каналами. Такие файлы могут быть доступны любым процессам.

---

[1] Взаимодействующие приложения могут договориться о протоколе, описывающем границы пакета информации в каждом объекте.

[2] Еще раз повторим, что приложения могут определить соглашение по маркировке источника данных каждого объекта.

Они являются постоянными и, следовательно, могут быть использованы для хранения данных пользователей, уже не активных в системе. Они обладают определенным именем в пространстве имен файловой системы. Конечно, файлы FIFO имеют и некоторые недостатки. По завершении использования их необходимо удалять принудительно. Они менее защищены по сравнению с программными каналами, так как к ним может обращаться любой процесс, обладающий достаточными привилегиями. Программные каналы проще в создании и требуют меньшего количества ресурсов.

6.2.3. Каналы в системе SVR4

В системе SVR4 как базовое средство работы с сетями и реализации каналов и файлов FIFO применяются STREAMS (см. главу 17). Это дает возможность предоставления системой новых полезных средств конвейеризации[1]. В этом разделе будут описаны только новые возможности каналов. Детали их организации обсуждаются в разделе 17.9.

В системе SVR4 каналы двунаправлены. Вызов pipe возвращает, как и ранее, два дескриптора, однако теперь они оба открыты как для чтения, так и для записи. Синтаксис вызова (совпадающий с традиционным) приведен ниже:

```
status = pipe(int fildes[2]);
```

В SVR4 вызов pipe создает два независимых канала ввода-вывода, работающих по принципу «первым вошел, первым вышел» и представленных двумя дескрипторами. Данные, записанные в fildes[1], могут быть прочитаны из fildes[0], а данные, записанные в fildes[0], могут быть прочитаны из fildes[1]. Такой подход является очень удобным, поскольку многие приложения требуют от системы возможности двухсторонних коммуникаций, для чего в предыдущих версиях SVR необходимо было открывать два отдельных канала.

Операционная система SVR4 может позволить процессу присоединить к объекту файловой системы любой дескриптор файла STREAMS [10]. Приложение создает канал посредством вызова pipe и затем связывает его дескрипторы с именем файла при помощи конструкции

```
status = fattach(int fildes, char *path);
```

где path — путь к объекту файловой системы, обладателем которого является вызывающий процесс[2]. Такой объект может быть обычным файлом, каталогом или специальным файлом. Он не может быть точкой монтирования (то есть на него не должна монтироваться файловая система) или объектом удаленной файловой системы. Он также не присоединяется к другому дескрип-

[1] Новые средства представлены только для программных каналов. Реализация файлов FIFO в ОС SVR4 почти совпадает с традиционной.

[2] В ином случае вызывающий процесс должен являться привилегированным.

тору файла STREAMS. Существует возможность связать дескриптор с несколькими путями, ассоциируя с ними несколько имен.

После присоединения все последующие операции над path будут на самом деле производиться над файлом STREAMS до тех тор, пока дескриптор не будет отключен от path через вызов fdetach. Используя описанную возможность, процесс может создавать канал и затем дать доступ к нему другим процессам системы.

Пользователь может поместить модули STREAMS в канал или файл FIFO. Такие модули перехватывают проходящие через канал данные и производят над ними различные действия. Так как эти модули функционируют внутри ядра системы, они умеют выполнять действия, недоступные пользовательским приложениям. Только привилегированные пользователи обладают правом добавлять в систему модули, но всем пользователям доступна возможность помещения модулей в открытые ими потоки.

6.2.4. Трассировка процессов

Системный вызов ptrace предлагает базовый набор средств трассировки процессов. Обычно он применяется различными отладчиками, например sdb или dbx. При помощи ptrace процесс может управлять выполнением своего потомка. Он может контролировать и несколько потомков сразу, но такое редко применяется на практике. Синтаксис ptrace следующий:

```
ptrace (cmd, pid, addr, data);
```

где pid — идентификатор трассируемого процесса, addr — ссылка на его адресное пространство. Интерпретация аргумента data зависит от cmd. Аргумент cmd позволяет процессу-предку осуществлять следующие действия:

- записывать или считывать слово из адресного пространства потомка;
- записывать или считывать слово из области u потомка;
- считывать или писать в общецелевые регистры потомка;
- перехватывать определенные сигналы. Если перехватываемый сигнал генерируется для потомка, ядро приостановит его работу и уведомит его предка о возникновении события;
- устанавливать или удалять точки наблюдения в адресном пространстве потомка;
- возобновлять выполнение приостановленного потомка;
- осуществлять пошаговое выполнение потомка: прервать его работу, затем позволить ему выполнить одну инструкцию, после чего снова приостановить его функционирование;
- завершить работу потомка.

Одна из команд (cmd==0) зарезервирована для потомка. Она применяется для информирования ядра системы о том, что потомок будет подвергнут трассировке своим предком. После этого ядро устанавливает для него флаг traced (в структуре proc), который влияет на обработку потомком сигналов. Если сигнал генерируется для трассируемого процесса, ядро приостанавливает его выполнение и уведомляет об этом его процесс-предок при помощи сигнала SIGCHLD (вместо загрузки обработчика сигнала). Это позволяет родительскому процессу перехватить сигнал и произвести соответствующие действия. Флаг traced также изменяет выполнение системного вызова exec. Если потомок загружает новую программу, вызов exec генерирует для потомка сигнал SIGTRAP до того, как он возвратится в пользовательский режим. Это дает возможность родительскому процессу получить контроль над потомком перед началом выполнения потомка.

Обычно родительский процесс создает процесс-потомок, который в дальнейшем вызывает ptrace для того, чтобы предок имел возможность управлять им. Затем родительский процесс применяет вызов wait для ожидания события, изменяющего режим выполнения потомка. После возникновения такого события ядро системы будит родительский процесс. Величина, возвращаемая wait, указывает на то, что потомок был приостановлен быстрее, чем завершен, и передает информацию о событии, заставившем потомка приостановить работу. Затем родительский процесс может контролировать дочерний при помощи одной или нескольких команд ptrace.

Хотя появление системного вызова ptrace помогло создать многие отладчики, он, к сожалению, обладает несколькими недостатками и ограничениями:

- процесс умеет управлять выполнением только своих прямых потомков. Если трассируемый процесс производит вызов fork, то отладчик не сможет контролировать новые процессы и их потомков;
- вызов ptrace является весьма неэффективным, так как требует нескольких переключений контекста только для того, чтобы передать одно слово от потомка его предку. Контекстные переключения являются обязательными, поскольку отладчик не обладает прямым доступом к адресному пространству потомка;
- отладчик не в состоянии следить за процессом, который уже находится в стадии выполнения, так как потомку необходимо сначала вызвать ptrace для уведомления ядра системы о том, что им можно управлять;
- трассировка программы setuid приводит к возникновению проблем защиты, если эта программа затем произведет вызов exec. Пользователь может использовать отладчик для изменения адресного пространства процесса, тогда вызов exec приведет к загрузке оболочки вместо программы, которую он хотел выполнить. В результате пользователь загрузит командный интерпретатор с привилегиями суперпользователя. Для предотвращения возникновения такой ситуации в системах UNIX либо отключают трассировку программ setuid, либо запрещают действия setiud и setgid с последующим вызовом exec.

В течение длительного периода времени вызов ptrace оставался единственным инструментом программ-отладчиков. В современных системах UNIX, таких как SVR4 и Solaris, имеются более эффективные средства отладки, использующие файловую систему /proc [9], описываемую в разделе 9.11.2. Эти средства не обладают ограничениями, присущими ptrace, и предоставляют различные дополнительные возможности, такие как отладка независимых друг от друга процессов или подключение отладчика к выполняющемуся процессу. После появления технологии /proc многие отладчики были переписаны заново и теперь не используют для своей работы вызов ptrace.

6.3. System V IPC

Средства, описанные в предыдущих разделах, не удовлетворяют требованиям многих приложений по взаимодействию процессов. Появление ОС System V ознаменовалось значительными расширениями возможностей IPC. Разработчики системы представили три механизма: семафоры, очереди сообщений и разделяемую память. Все они известны под общим названием System V IPC [1]. Изначально они создавались для поддержки приложений обработки транзакций. Позже эти средства были взяты на вооружение большинством поставщиков систем UNIX, в том числе и теми, кто создавал ОС на основе BSD. В данном разделе будут описаны возможности этих механизмов и показано, как они были практически реализованы в системе UNIX.

6.3.1. Общие элементы

Все три механизма очень схожи друг с другом в отношении программного интерфейса и своей реализации. При описании их общих возможностей мы будем использовать термин «ресурс IPC» (или просто «ресурс») вместо указания на набор семафоров, очередь сообщений или область разделяемой памяти. Каждый ресурс IPC обладает набором атрибутов.

- **Ключ** (key). Поддерживаемое пользователем целое число, идентифицирующее конкретный экземпляр ресурса.
- **Создатель** (creator). Пользовательские и групповые идентификаторы процесса, создавшего ресурс.
- **Владелец** (owner). Пользовательские и групповые идентификаторы владельца ресурса. При создании ресурса его владелец и создатель одинаковы. Однако процесс, обладающими правами изменения владельца, может указать позже нового владельца ресурса. Такими правами обладают процессы создателя, текущего владельца и суперпользователя.
- **Права** (permissions). Права файловой системы на чтение/запись/выполнение для владельца, группы и других пользователей.

Процесс получает ресурс при помощи системных вызовов shmget, semget и msgget, передавая им ключ, необходимые флаги и другие аргументы, зависящие от используемого механизма. Разрешенными флагами являются IPC_CREAT и IPC_EXCL. Первый из них запрашивает ядро о создании ресурса, если таковой не существует. Флаг IPC_EXCL применяется совместно с IPC_CREAT и запрашивает ядро о возвращении ошибки, если требуемый ресурс уже существует. Если не указывается ни один из флагов, ядро ищет существующий ресурс с тем же ключом[1]. Если оно находит такой ресурс и если вызывающий процесс обладает правами доступа к нему, то ядро возвращает *идентификатор ресурса* (resourse ID), который может быть в дальнейшем использован для быстрого обнаружения ресурса.

Каждый механизм обладает управляющим системным вызовом (shmctl, semctl и msgctl), предоставляющим несколько различных команд. Эти команды включают в себя IPC_STAT и IPC_SET, которые применяются для получения и установки статусной информации (специфичной для каждого механизма), а также IPC_RMID для освобождения ресурса. Для управления семафорами поддерживаются дополнительные команды управления, которые используются в целях получения и установки переменных отдельных семафоров набора.

Каждый ресурс IPC должен освобождаться принудительно при помощи команды IPC_RMID. В противном случае ядро системы будет считать ресурс активным, даже если все использовавшие его процессы были завершены. Такое свойство ресурсов может оказаться весьма удобным. Например, процесс может записать данные в разделяемую область памяти или в очередь сообщений и затем завершить свою работу. Эти данные может запросить позже другой процесс. Ресурс IPC является постоянным, то есть годится для использования уже после завершения работы процесса, обращавшегося к нему.

Однако такой подход имеет и определенные недостатки, так как ядро системы не может определить, был ли ресурс оставлен активным для новых процессов или он был потерян случайно, например, если работа процесса была завершена принудительно и тот не успел освободить ресурс. В результате ядру системы необходимо продолжать поддерживать ресурс, имеющий неизвестное состояние. Если это случается слишком часто, система может такой ресурс выгрузить, так как он, по крайней мере, занимает определенный объем памяти, которому можно найти другое, более полезное применение.

Правами на выполнение команды IPC_RMID обладают только процессы, являющиеся создателем, текущим владельцем ресурса или обладающие привилегиями суперпользователя. Удаление ресурса влияет на все процессы, обращающиеся к нему в текущий момент, следовательно, ядро системы должно

---

[1] Если ключ является специальной переменной IPC_PRIVATE, ядро создаст новый ресурс. Этот ресурс недоступен при помощи других вызовов get (так как ядро будет каждый раз создавать еще один ресурс), и, следовательно, вызывающий процесс будет обладать исключительными правами на его владение. Владелец может использовать ресурс совместно со своими потомками, которые наследуют его через вызов fork.

удостовериться в том, что процессы постепенно и согласованно обрабатывают такое событие. Специфика предпринимаемых действий зависит от конкретного применяемого механизма и будет подробнее описана в следующих разделах.

Для реализации интерфейса каждый тип ресурса обладает собственной таблицей ресурса фиксированного размера. Размер таблицы является настраиваемым и ограничен общим количеством каждого типа ресурсов, одновременно поддерживаемых системой. Каждый элемент таблицы содержит общую структуру ipc_perm, а также данные, специфичные для определенного типа ресурса. Структура ipc_perm хранит общие атрибуты ресурса (ключ, идентификаторы создателя и владельца, привилегии), а также последовательность чисел, являющуюся счетчиком, который увеличивается при каждом новом использовании элемента таблицы.

При создании ресурса IPC пользователем ядро возвращает идентификатор ресурса, который вычисляется по формуле:

```
id = seq * table_size + index;
```

где seq — последовательность чисел для этого ресурса, table_size — размер таблицы ресурса и index — индекс ресурса в таблице. Эта формула гарантирует создание нового id, если элемент таблицы используется повторно, так как значение seq инкрементируется. Это защищает процессы от доступа к ресурсам, использующим устаревший идентификатор.

ПРИМЕЧАНИЕ

Термин «инкрементирование переменной» означает, что ее значение увеличивается на единицу. «Декремент» означает уменьшение значения на единицу. Эти термины взяты из языка C, где применяются операторы инкремента (++) и декремента (--)[1].

Пользователь передает id в качестве аргумента последующим системным вызовам, производящим действия над этим ресурсом. Ядро транслирует id для обнаружения ресурса в таблице по формуле

```
index = id % table_size;
```

6.3.2. Семафоры

Семафоры [6] — это объекты, находящиеся в диапазоне целых чисел, которые поддерживают две операции, P() и V()[2]. Примитив P() применяется для декремента значения семафора, если новое значение оказывается меньше нуля, то он блокируется. Операция V() используется для инкремента значения, при

[1] Впервые операторы автоувеличения и автоуменьшения появились в компиляторе языка B – предшественника C, они были введены К. Томпсоном для собственных нужд при работе с PDP-7. Позже популярность C и UNIX на PDP-10 обусловилась во многом этими режимами, как пишет Д. Ритчи в *The Development of the C Language. — Прим. ред.*

[2] Имена P() и V() происходят от голландских слов, означающих соответствующие операции.

этом если результат оказывается равным нулю или больше, то V()пробуждает нить или процесс. Эти операции являются неделимыми.

Семафоры можно применять для реализации различных протоколов синхронизации. Например, представьте, какие проблемы возникают при управлении исчисляемого ресурса (обладающего определенным числом элементов). Процесс пытается получить одну их составляющих ресурса и освободить ее после завершения использования. Ресурс может быть представлен семафором, который применяется для управления доступа к нему. Примитив P() применяется при каждой попытке запроса ресурса и уменьшает значение семафора при ее удачном завершении. Если переменная станет равна нулю (то есть свободных ресурсов больше нет), все последующие операции приведут к блокировке. При освобождении ресурса стартует операция V(), которая увеличивает значение семафора, что приводит к пробуждению заблокированного процесса.

Во многих системах UNIX семафоры применяются ядром для синхронизации внутренних операций. Они также используются в этом качестве и для пользовательских приложений. В ОС System V поддерживается наиболее общая версия семафоров. Системный вызов semget создает или запрашивает массив семафоров (верхнюю границу которого можно назначить). Синтаксис вызова приведен ниже:

```
semid = semget (key, count, flag);
```

где key — 32-разрядная переменная, передаваемая вызывающим процессом. Функция semget возвращает массив count семафоров, ассоциированных с ключом (key). Если с ключом не связано ни одного набора семафоров, то вызов будет возвращать ошибку до тех пор, пока не будет задан флаг IPC_CREAT, создающий новый набор семафоров. Если функции передается флаг IPC_EXCL, то semget возвращает ошибку в том случае, если набор семафоров для указанного ключа уже существует. Переменная semid применяется в последующих операциях над семафорами, она идентифицирует массив семафоров.

Системный вызов semop применяется для проведения операций над отдельными семафорами массива. Ее синтаксис таков:

```
status = semop (semid, sops, nsops);
```

где sops — указатель на элемент nsops массива структур sembuf. Каждая структура sembuf, как это будет показано ниже, представляет одну из операций над отдельным семафором набора.

```
struct sembuf {
   unsigned short sem_num;
   short sem_op;
   short sem_flg;
};
```

Здесь sem_num идентифицирует один из семафоров массива, а sem_op указывает на действия, которые следует с ним произвести. Значения переменной sem_op интерпретируются по следующим правилам:

sem_op > 0	Добавляет sem_op к текущему значению семафора. Результатом этой операции может стать пробуждение процесса, ожидавшего увеличения переменной
sem_op = 0	Блокировка до тех пор, пока значение семафора не станет равным нулю
sem_op < 0	Блокировка до момента, когда значение семафора станет равным или выше абсолютного значения sem_op, после чего производится вычитание sem_op от значения семафора. Если оно уже выше, чем абсолютная величина sem_op, вызывающий процесс не блокируется

Таким образом, используя всего лишь один системный вызов semop, можно указать несколько различных отдельных операций над семафорами. Ядро системы гарантирует выполнение либо всех этих операций, либо ни одной из них. Более того, ядро отслеживает работу semop, не позволяя начать обработку повторного вызова над тем же массивом семафоров до тех пор, пока первый вызов не завершит работу либо не будет заблокирован. Если вызову semop необходимо приостановить свое выполнение после завершения некоторых действий, то ядро системы после возобновления его работы повторит операцию сначала (отменит все изменения), гарантируя тем самым неделимость функции.

Значениями аргумента sem_flg могут быть два различных флага. Флаг IPC_NOWAIT указывает ядру на необходимость возврата ошибки функцией вместо приостановки ее выполнения. Если процесс, удерживающий семафор, завершит работу, не освободив его, появляется вероятность возникновения взаимоблокировки. Тогда другой процесс, пытающийся воспользоваться семафором, может оказаться заблокированным навсегда на стадии выполнения операции P(). Для защиты от этого вызову semop можно передавать флаг SEM_UNDO. В этом случае ядро запоминает произведенные им операции и автоматически прокручивает их назад по завершении работы процесса.

Семафоры необходимо удалять из системы принудительно. Для этого применяется команда IPC_RMID вызова semctl. В противном случае ядро системы будет поддерживать семафоры, даже если они не используются ни одним из процессов. Такой подход позволяет применять семафоры в течение длительного времени, не зависящего от жизненного цикла процессов. Но, с другой стороны, если приложение по завершении работы не освобождает семафоры, то они продолжают занимать ресурсы системы.

После выполнения процессом команды IPC_RMID ядро освобождает семафор в таблице ресурсов. Ядро также пробуждает все процессы, заблокированные во время проведения операций над тем же семафором. Результатом

работы semop для таких процессов является возврат статуса EIDRM. После удаления семафора процесс больше не имеет доступа к нему (с помощью ключа или идентификатора семафора).

Детали реализации семафоров

Ядро преобразует semid для получения элемента таблицы ресурсов семафоров, каждый из которых описывается следующей структурой данных:

```
struct semid_ds {
   struct ipc_rerm sem_perm; /* см. раздел 6.3.1 */
   struct sem* sem_base; /* указатель на массив семафоров в наборе */
   ushort sem_nsems; /* количество семафоров в наборе */
   time_t sem_otime; /* время последней операции */
   time_t sem_ctime, /* время последнего изменения */
   ...
};
```

Ядро поддерживает значение и информацию о синхронизации для каждого семафора в наборе в структуре, показанной ниже.

```
struct sem {
   ushort semval; /* текущее значение */
   pid_t sempid; /* идентификатор процесса, вызвавшего последнюю
      операцию */
   ushort semncnt; /* количество процессов, ожидающих увеличения значения
      семафора */
   ushort semzcnt; /* количество процессов, ожидающих установления значения
      семафора, равного нулю */
};
```

Ядро также поддерживает список отмены для каждого процесса, производящего операции над семафорами с флагом SEM_UNDO. Этот список содержит запись операций, каждую из которых можно отменить. Если процесс завершает выполнение, ядро проверяет наличие списка отмены. Если оно находит такой список, то «прокручивает» в обратном направлении все действия, совершенные ранее.

Применение семафоров

Технология семафоров позволяет разрабатывать сложные средства синхронизации, используемые взаимодействующими процессами. Первые системы UNIX не обладали поддержкой семафоров, что заставляло для синхронизации приложений искать и применять другие атомарные операции. Например, системный вызов link возвращает ошибку, если новое соединение уже существует. Если два процесса пытаются произвести одну и ту же операцию link в один момент времени, только для одного из них результат будет успешным. Однако применение операций файловой системы, таких как link, в целях синхронизации процессов является неудобным и громоздким.

Появление семафоров смогло удовлетворить большинство требований создателей программных приложений.

Основными проблемами, связанными с применением семафоров, являются условия состязательности и предупреждение взаимоисключений. Использование одиночных семафоров (вместо их массивов) может запросто привести к взаимной блокировке в том случае, если процессу необходимо запросить несколько семафоров. Например, на рис. 6.2 процесс А, удерживающий семафор С1, пытается получить семафор С2 в то время, как процесс Б уже владеет семафором С2 и стремится к тому же в отношении С1. В таком случае ни один из процессов не сможет продолжить функционирование. Хотя этот простейший случай легко обнаружить (и, следовательно, защититься от его возникновения), клинч может произойти и в более сложных вариациях, затрагивающих не один процесс и семафор.

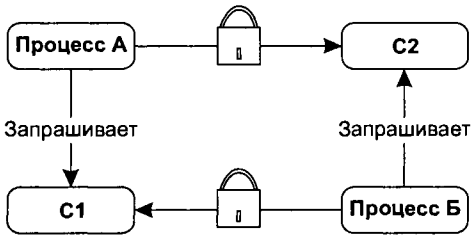


Рис. 6.2. Семафоры могут стать причиной возникновения взаимоблокировок

Коды обнаружения взаимоблокировок и защиты от них реализовывать внутри ядра системы непрактично. Более того, не существует общих и универсальных алгоритмов, защищающих от всевозможных ситуаций. Из этого можно сделать вывод, что ядро оставляет решение проблемы взаимоблокировок приложениям. Предлагая механизм наборов семафоров и неделимые операции над ними, ядро системы дает разработчику интеллектуальные механизмы обработки семафоров. Программисты могут выбрать несколько известных методов защиты от тупиковых ситуаций, некоторые из них будут показаны в разделе 7.10.1.

Одна из основных проблем реализации семафоров в System V связана с тем, что операции по их инициализации и размещению не являются неделимыми. Для размещения набора семафоров пользователь вызывает `semget` вслед за функцией `semctl`, которая инициализирует набор. Такой подход может привести к условию состязательности, которую необходимо пресекать на пользовательском уровне [13].

В завершение скажем, что необходимость принудительного удаления ресурса при помощи команды `IPC_RMID` является общей проблемой любых механизмов IPC. Хотя это свойство и позволяет создателю пережить созданный им ресурс, оно приводит к появлению «мусора» в системе в случае завершения работы процесса без высвобождения всех его ресурсов.

6.3.3. Очереди сообщений

Очередь сообщений — это заголовок, указывающий на связанный список сообщений. Каждое сообщение содержит 32-разрядную переменную *типа*, следующую за областью *данных*. Процесс создает или получает очередь сообщений при помощи системного вызова msgget, синтаксис которого показан ниже:

```
msgpid = msgget (key, flag),
```

Семантика вызова msgget совпадает с semget. Key — это целое число, задаваемое пользователем. Для создания новой очереди сообщений необходимо указать флаг IPC_CREAT. Задание флага IPC_EXCL ведет к ошибочному завершению работы вызова в том случае, если очередь с указываемым ключом уже существует. Переменная msgpid используется в дальнейших вызовах для доступа к очереди.

Для того чтобы поместить сообщение в очередь, необходимо произвести следующий вызов:

```
msgsnd (msgqid, msgp, count, flag);
```

где msgp указывает на буфер сообщения (содержащий поле типа type, следующий за областью данных), count — общее количество байтов в сообщении (включает поле type). Флаг IPC_NOWAIT используется для возврата ошибки, если сообщение невозможно отправить без блокировки (например, когда очередь переполнена, так как очередь обычно обладает настраиваемым ограничением на количество хранящихся в ней данных).

На рис. 6.3 показаны операции над очередями сообщений. Каждая очередь описывается в виде строки в таблице ресурсов очередей сообщений. Эта структура показана ниже:

```
struct msqid_ds {
   struct msg ipc_perm msg_perm; /* см. раздел 6.3.1 */
   struct msg* msg_first; /* первое сообщение в очереди */
   struct msg* msg_last; /* последнее сообщение в очереди */
   ushort msg_cbytes; /* текущая величина очереди в байтах */
   ushort msg_qbytes; /* максимально допустимый размер очереди в байтах */
   ushort msg_qnum; /* текущее количество сообщений в очереди */
   ...
},
```

Сообщения располагаются внутри очереди в порядке их поступления. Они удаляются из очереди по принципу «первым вошел, первым вышел» при чтении их процессом при помощи вызова

```
count = msgrcv(msgqid, msgp, maxcnt, msgtype, flag);
```

в котором msgp указывает на буфер, в который помещается входящее сообщение, maxcnt ограничивает максимально прочитываемое количество байтов. Если входящее сообщение длиннее, чем maxcnt, то оно будет обрезано. Поль-

зователь должен быть уверен в том, что буфер, указанный при помощи msgp, имеет достаточный объем для хранения maxcnt байтов данных. Возвращаемая функцией величина указывает на успешно прочитанное количество байтов.

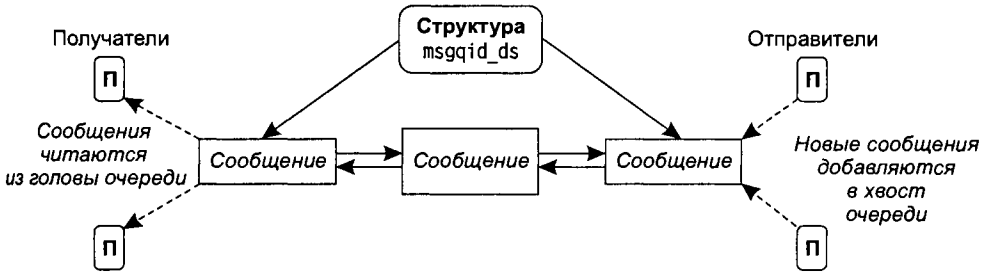


Рис. 6.3. Применение очередей сообщений

Если msgtype равно нулю, вызов msgrcv вернет первое по счету сообщение очереди. Если значение msgtype больше нуля, вызов возвратит первое сообщение типа msgtype. Если msgtype меньше нуля, функция вернет первое сообщение наименьшего типа, то есть типа, меньшего или равного абсолютной величине msgtype. Точно так же, как и в предыдущей функции, флаг IPC_NOWAIT заставляет msgrcv возвратиться немедленно, если необходимое сообщение отсутствует в очереди.

После прочтения сообщение удаляется из очереди и, следовательно, не может быть прочитано другими процессами. Если часть сообщения отрезана вследствие недостаточности объема буфера, то эта часть теряется навсегда. Никакой индикации о факте обрезки сообщения не производится.

Процесс должен удалять очередь сообщений принудительно при помощи вызова msgctl с указанием команды IPC_RMID. При этом ядро освобождает очередь и удаляет все сообщения, находившиеся в ней. Если какие-либо процессы были заблокированы в ожидании чтения или записи в очередь, ядро разбудит их, а вызванная им системная функция возвратит статус EIDRM (удалено).

Применение очередей сообщений

Каналы и очереди сообщений предлагают схожие услуги, однако последние являются более универсальным механизмом, в котором были преодолены некоторые ограничения, присущие каналам. Очереди сообщений передают данные в виде дискретных сообщений, в то время как каналы работают с неформатированным потоком байтов. Это дает возможность более интеллектуальной обработки передаваемой информации. Поле сообщения type (тип) можно использовать различными способами. Например, это поле может указывать на приоритет сообщений, давая возможность получателю проверять более важные сообщения раньше, чем остальные. Если очередь сообщений обрабатывается одновременно несколькими процессами, поле типа может быть использовано для определения адресата.

Очереди сообщений являются эффективным средством передачи небольших объемов данных, но становятся слишком неудобными для больших объемов информации. Когда процесс отправляет сообщение в очередь, ядро системы копирует его во внутренний буфер. Если другой процесс запросит это сообщение, ядро скопирует данные в адресное пространство получателя. Таким образом, передача сообщения требует проведения двух операций копирования, что приводит к низкой производительности. Позднее в этой главе будет рассказано о средствах межпроцессного взаимодействия Mach, позволяющих эффективно передавать большие объемы данных.

Еще одним ограничением очередей сообщений является невозможность указания получателя. Любой процесс, обладающий соответствующими полномочиями, имеет право запрашивать сообщение из очереди. Хотя, как это упоминалось ранее, взаимодействующие процессы могут договориться о протоколе указания адресатов, ядро системы никак не участвует в этом. Также механизм очередей сообщений не поддерживает *широковещательную передачу*, при применении которой процесс может отправлять одно и то же сообщение нескольким получателям.

Средства STREAMS поддерживаются большинством современных систем UNIX. Они обладают богатыми возможностями передачи сообщений. STREAMS являются более функциональными, чем очереди сообщений. Одной из возможностей, которой обладают очереди сообщений, но не поддерживающейся STREAMS, являются селективные запросы сообщений в зависимости от их типов. Однако разработчики большинства приложений считают STREAMS более удобным интерфейсом, поэтому очереди сообщений в современных системах UNIX оставлены больше из соображений обратной совместимости. Более подробно работа STREAMS будет рассмотрена в главе 17.

6.3.4. Разделяемая память

Область разделяемой памяти — это некоторый объем физической памяти, который используется совместно сразу же несколькими процессами. Процесс может присоединить эту область в качестве диапазона виртуальной памяти в адресном пространстве процесса. Диапазон может быть различным для каждого процесса (рис. 6.4). После присоединения процесс обладает доступом к этой области, не отличающимся от доступа к любому другому участку памяти, то есть без необходимости применения системных вызовов для записи или чтения данных из нее. Следовательно, *механизм разделяемой памяти предоставляет процессу максимально быстрый способ доступа к данным.* Если процесс записывает данные в ячейки разделяемой памяти, их содержимое незамедлительно становится видимым остальным процессам, разделяющим между собой эту область[1].

[1] На многопроцессорных системах для гарантии целостности кэша необходимы дополнительные операции. Некоторые из них будут описаны в разделе 15.13.

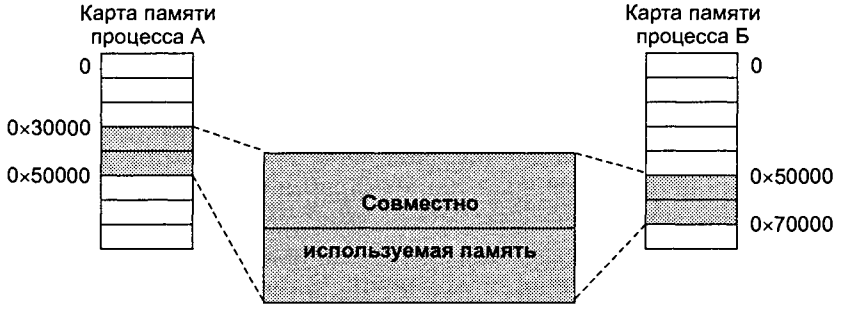


Рис. 6.4. Присвоение разделяемой области памяти

Для первоначального получения или создания области разделяемой памяти процесс использует следующий вызов:

```
shmid = shmget (key, size, flag);
```

где size — размер области; остальные параметры и флаги идентичны входным аргументам вызовов semget и msgget. Затем процесс производит присоединение области виртуальной адресации:

```
addr = shmat (shmid, shmaddr, shmflg);
```

Аргумент shmaddr указывает на адрес, к которому может быть присоединена область разделяемой памяти. В качестве аргумента shmflag можно указать флаг SHM_RND, который запросит у ядра присвоить shmaddr приблизительно, исходя из указанного диапазона. Если shmaddr равен нулю, ядро вправе выбрать для области любой адрес. Флаг SHM_RDONLY указывает на необходимость присвоения области только для чтения. Вызов shmat возвращает адрес, присвоенный области.

Процесс может исключить область разделяемой памяти из своего адресного пространства при помощи вызова

```
shmdt (shmaddr);
```

Для полного удаления области процессу необходимо использовать команду IPC_RMID системного вызова shmctl. Эта функция помечает область как удаленную, которая подвергнется удалению только после того, как все процессы произведут ее отключение от своего адресного пространства. Ядро поддерживает данные о числе процессов, подключенных к каждой области. Если область помечена как удаленная, новые процессы не могут подключаться к ней. Если область не была удалена принудительно, ядро системы будет продолжать поддерживать ее даже в случае отсутствия подключенных к ней процессов. Такой подход удобен для многих приложений, так как процесс имеет возможность перед завершением работы оставить какие-либо данные, которые могут быть получены позже. Для этого взаимодействующий процесс подключается к области памяти, используя тот же ключ key.

Реализация разделяемой памяти сильно зависит от архитектуры виртуальной памяти конкретной операционной системы. Некоторые варианты ОС используют для назначения области разделяемой памяти единую таблицу страниц, после чего предоставляют доступ к этой таблице всем процессам, подключенным к области. В других реализациях ОС применяются отдельные для каждого процесса карты трансляции адресов областей. При использовании этой модели, если процесс выполняет действие, изменяющее назначение памяти для разделяемой страницы, то это должно приводить к изменению всех назначений для этой страницы памяти. В системе SVR4 (о реализации обработки памяти которой будет рассказано в главе 14) для размещения страниц области разделяемой памяти используется структура anon_map. Таблица ресурсов разделяемой памяти содержит строки, представленные следующей структурой:

```
struct shmid_ds {
struct ipc_perm shm_perm; /* см  раздел 6.3.1 */
int shm_segsz; /* размер сегмента в байтах */
struct anon_map *shm_amp; /* указатель на информацию обработки памяти */
ushort shm_nattch; /* текущее количество подключений */
 ..
};
```

Механизм разделяемой памяти является быстрым и универсальным средством, позволяющим совместно использовать большие объемы данных без применения копирования или системных вызовов. Основным ограничением механизма является отсутствие средств синхронизации. Если два процесса пытаются изменить одну и ту же разделяемую область памяти, ядро системы не может обеспечить последовательность этих операций, что приведет к смешению записанных данных. Процессы, разделяющие между собой область памяти, должны самостоятельно поддерживать собственный протокол синхронизации. Обычно для этой цели используются простые конструкции, такие как семафоры. Применение таких конструкций приводит к необходимости вызова одной или нескольких системных функций, что уменьшает производительность работы с разделяемой памятью.

Многие современные системы UNIX (в том числе и SVR4) предлагают системный вызов mmap, который назначает файл (или его часть) как адресное пространство вызывающего процесса. Процессы могут применять вызов mmap для осуществления взаимодействия между собой путем назначения одного и того же файла в адресном пространстве каждого процесса (в режиме MAP_SHARED). В результате этих действий появляется область разделяемой памяти, которой является содержимое файла. Если процесс изменяет назначенный файл, то произведенные им изменения становятся сразу видны остальным процессам, подключенным к этому файлу. Обновление файла на диске производится ядром системы. Преимуществом вызова mmap является использование имен, принятых в файловой системе, вместо ключей. В отличие от

разделяемой памяти, чьи страницы резервируются в области свопинга (см. раздел 14.7.6), страницы, созданные mmap, резервируются при помощи файла, к которому они были присоединены. Более подробно работа mmap будет показана в разделе 14.2.

6.3.5. Применение механизмов IPC

Существуют определенные сходства между механизмами IPC и файловой системой. Идентификатор ресурса похож на дескриптор файла. Вызов get повторяет open, команда IPC_RMID схожа с вызовом unlink, а вызовы send и receive аналогичны read и write. Системный вызов shmdt предлагает возможности для разделяемой памяти, сходные с close. Однако для очередей сообщений и семафоров не существует эквивалента вызова close, так как представляется более предпочтительным удалять эти ресурсы сразу же. В результате процесс, использующий очередь сообщений или семафоры, может обнаружить, что необходимый ему ресурс больше не существует.

Ключи, ассоциируемые с ресурсами и формирующие пространство их имен, значительно отличаются от имен, применяемых в файловых системах[1]. Каждый из этих механизмов обладает собственным адресным пространством, ключ используется для его однозначной идентификации. Так как ключ представляет собой простое целое число, выбираемое пользователем произвольно, он может быть использован только на одной машине и не подходит для распределенных сред вычислений. Также задача использования уникальных ключей независимыми между собой процессами представляет определенную трудность, поскольку простота ключей может привести к конфликтам при использовании их другими приложениями. В системе UNIX поддерживается библиотечная процедура ftok (описанная в руководстве stdipc(C3)), которая применяется для создания ключей, основанных на именах файлов и целых числах. Синтаксис ftok показан ниже:

```
key = ftok (char *pathname, int ndx);
```

Процедура ftok создает значение ключа на основе ndx — номера индексного дескриптора файла. Намного проще решить задачу уникальности имени ключа при применении имен файлов (приложение, к примеру, может использовать в качестве такого имени путь к собственному выполняемому файлу) и тем самым уменьшить вероятность возникновения конфликтов. Параметр ndx увеличивает гибкость библиотечной процедуры и может быть использован для указания идентификатора проекта, известного всем взаимодействующим приложениям, или другой полезной информации.

Основной проблемой механизмов IPC является их незащищенность, так как идентификатор ресурса представляет собой ссылку на общую таблицу

[1] В некоторых ОС, таких как Windows NT или OS/2, для имен разделяемых областей используются пути файловой системы (но не в обязательном порядке, разделяемые области могут быть и неименованными). — *Прим. ред.*

ресурсов. Доступ к ресурсу может получить процесс, не обладающий соответствующими полномочиями. Для этого нужно просто угадать идентификатор ресурса. Таким образом, процесс имеет потенциальную возможность считывать или записывать сообщения в разделяемую память или искажать семафоры других процессов. Назначение определенных полномочий ресурсу может частично решить проблему защиты, однако часто процессам необходимо использовать ресурсы совместно с другими процессами, относящимися к разным пользователям системы. Следовательно, в таких случаях невозможно применять строгие разграничения прав доступа. Использование последовательности чисел в качестве идентификатора ресурса дает очень ограниченную защищенность из-за того, что их не так уж трудно угадать. Их применение ставит определенные проблемы перед приложениями, обладающими требованиями по безопасности.

Большинство средств, представленных в System V IPC, может быть реализовано при помощи других возможностей системы, например блокировки или каналов. Однако инструменты IPC являются более гибкими, эффективными и обладающими большей производительностью, чем компоненты файловой системы ОС.

6.4. Mach IPC

Оставшаяся часть главы будет посвящена рассмотрению средств IPC системы Mach, базирующихся на сообщениях. В Mach средства взаимодействия процессов являются центральным и наиболее важным компонентом ядра. Вместо реализации поддержки IPC системой разработчики Mach создали средства IPC, которые являются основой операционной системы. При разработке Mach IPC были поставлены следующие цели:

- передача сообщений должна быть фундаментальным механизмом коммуникаций;
- поддерживаемый объем данных одного сообщения должен варьироваться от нескольких байтов до размера всего адресного пространства системы (то есть до 4 Гбайт). Ядро должно осуществлять передачу больших объемов информации без копирования информации;
- ядро обязано поддерживать защищенные соединения и позволять отправку и получение сообщений только авторизированным нитям;
- средства коммуникаций и управления памятью должны быть тесно взаимосвязаны между собой. Подсистема IPC использует механизмы копирования при записи подсистемы памяти для повышения эффективности передачи больших объемов данных. С другой стороны, подсистема памяти прибегает к механизмам IPC для осуществления коммуникаций с пользовательскими программами управления памятью;

- средства Mach IPC должны поддерживать коммуникации между пользовательскими заданиями, а также между пользователем и ядром. В системе Mach нить производит системный вызов при помощи отправки сообщения ядру, а ядро, в свою очередь, возвращает результат вызова в ответном сообщении;
- механизм IPC должен подходить для клиент-серверных приложений. В системе Mach для поддержки многих служб реализованы серверные программы пользовательского уровня (например, поддержка файловой системы или управление памятью), которые ранее обычно обрабатывались ядром операционной системы. Такие серверные программы используют Mach IPC для обработки запросов к службам;
- интерфейс должен быть расширяемым до поддержки распределенных вычислений. Пользователь не должен знать, куда он отправляет сообщения: на локальный сервер или удаленный узел.

Средства IPC долгое время оставались неизменными в различных версиях системы Mach. Механизмы взаимодействия процессов Mach 2.5 будут представлены в разделах 6.4–6.9. Эта версия системы является наиболее популярной, и именно ее взяли за основу разработчики OSF/1 и Digital UNIX. В Mach 3.0 возможности IPC были расширены. О них пойдет речь в разделе 6.10.

В этой главе вы снова увидите термины «задание» или «нить» системы Mach. Более подробно они описывались в разделе 3.7.1. Вкратце, задание является набором ресурсов, в том числе пространства адресов, в котором выполняется одна или несколько нитей. Нить — это динамическая сущность, имеющая независимые счетчик команд и стек (логическая управляющая последовательность) программы. Традиционный процесс UNIX эквивалентен заданию, содержащему одну нить. Все нити задания разделяют между собой его ресурсы.

6.4.1. Основные концепции

Основными понятиями IPC в системе Mach являются сообщения и порты. *Сообщение* — это набор данных определенного типа. *Порт* — это защищенная очередь сообщений. Любое сообщение можно отослать только в порт, но не заданию или нити. В ОС Mach каждому порту присваиваются *права на получение* и *права на отправку*. Эти права поддерживаются задачами. Право на передачу разрешает задаче оправлять в порт сообщения. Право на получение позволяет получать сообщения, которые были посланы в порт. Правом на отправку сообщений в один и тот же порт могут обладать сразу несколько заданий, но только одно из них, называемое владельцем порта, имеет право на по-

лучение[1]. Таким образом, порты разрешают производить соединения типа «многие к одному», как это показано на рис. 6.5.

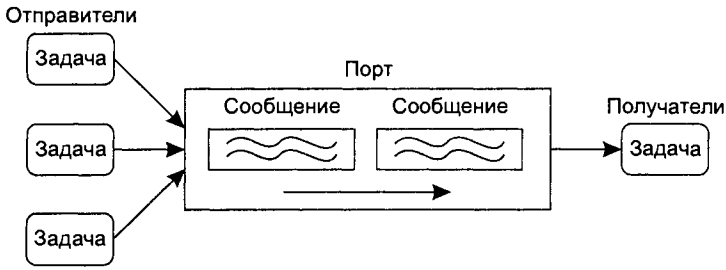


Рис. 6.5. Осуществление коммуникаций через порты системы Mach

Сообщение может быть простым или сложным. Простое сообщение включает в себя обычные данные, не интерпретируемые ядром. Сложное сообщение может содержать обычные данные, внешней памяти (данные, передаваемые при помощи метода «копирования при записи»), а также права на отправку и получение для различных портов. Ядро интерпретирует эту информацию и преобразует ее в форму, необходимую получателю.

Каждый порт имеет счетчик ссылок, отслеживающий количество его прав. Каждое такое право (также называемое совместимостью) представляет одно из имен порта. Имена представляют собой целые числа, диапазон имен является локальным для каждого задания. Следовательно, для одного и того же порта двумя заданиями могут применяться разные имена (рис. 6.6). Точно так же, одно и то же имя порта в разных заданиях может ссылаться на различные порты.

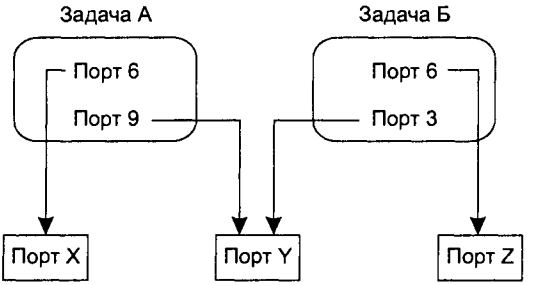


Рис. 6.6. Локальные имена портов

Порты также используются для представления объектов ядра. Следовательно, каждый объект, такой как нить, задание или процессор, может быть

[1] В ранних версиях системы Mach разделялись понятия владельца и обладателя права на получение. В версии 2.5 (и более поздних) права владельца заменены средством *восстановления порта*, которое будет описано в разделе 6.8.2.

представлен как порт. Права таких портов представляют собой ссылки на объекты и позволяют производить различные операции над объектами. Обслуживание прав таких портов закреплено за ядром.

Каждый порт имеет очередь сообщений конечного размера. Размер очереди является простым механизмом управления нитями. Отправители блокируются при заполнении очереди, а получатели — при пустой очереди.

Каждое задание или нить обладает набором портов, заданных по умолчанию. Например, каждое задание имеет право на отправку в порт task_self, представляющий само задание (правами на получение из этого порта обладает ядро) и право на получение из порта task_notify (отправлять сообщения в который может ядро). Задание владеет правами на получение из порта bootstrap, который предоставляет доступ к серверу имен. Каждая нить имеет право на отправку в порт thread_self и право на получение из порта reply, используемый для приема ответов от системных вызовов и удаленных вызовов процедур к другим нитям. Существует также порт exception, который ассоциируется с каждым заданием или нитью. Владельцем прав на порты нити является задание, в котором выполняется эта нить. Следовательно, такие порты являются доступными каждой нити этого задания.

Задания наследуют права на порты от своих предков. Каждое задание обладает списком *зарегистрированных портов*. Такой подход позволяет заданию иметь доступ к различным системным службам. Порты наследуются новыми заданиями на стадии их создания.

6.5. Сообщения

Ядро системы Mach основано на сообщениях, которые используются для доступа к большинству системных служб. Средства Mach IPC предоставляют возможность коммуникации между пользовательскими задачами, между пользователями и ядром, а также между различными подсистемами внутри ядра системы. Расширение средств IPC на сеть реализовано при помощи программы пользовательского уровня под названием netmsgserver, позволяющей приложениям обмениваться сообщениями по сети точно так же, как и внутри одной машины. Фундаментальными компонентами Mach IPC являются сообщения и порты. В этой главе вы увидите описание структур данных и функций, при помощи которых реализованы эти компоненты системы.

6.5.1. Структуры данных сообщения

Сообщение представляет собой набор данных определенного типа. Существует три основных типа данных, располагаемых в сообщениях:

- обычные данные, не интерпретируемые ядром и передающиеся получателю путем их физического копирования;

- внешняя память (out-of-line), используемая для передачи больших объемов данных при помощи технологии «копирования при записи» (см. раздел 6.7.2);
- передача на порты или прием прав.

Каждое сообщение имеет заголовок фиксированной величины, после которого сразу же располагается набор компонентов данных переменного размера (рис. 6.7). Заголовок сообщения содержит следующую информацию:

- **тип** (type). Простой (только обычные данные) или сложный (в сообщение включена информация внешней памяти или прав портов);
- **размер** (size). Указывается размер всего сообщения (тело + заголовок);
- **порт назначения** (destination port);
- **порт ответа** (reply port). Отправка прав на порт, на который можно послать ответ. Поле устанавливается только в случае необходимости пересылки ответа отправителю;
- **идентификатор сообщения** (message ID). Используется приложениями по своему усмотрению.

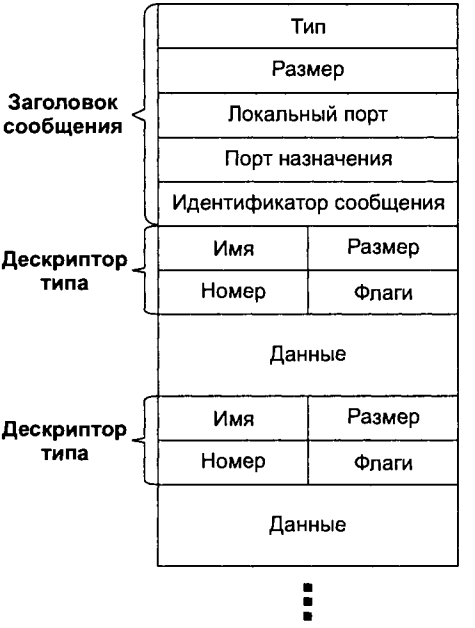


Рис. 6.7. Сообщение системы Mach

Перед отправкой сообщение создается в адресном пространстве задания-отправителя. На этом этапе для указания портов назначения и ответа применяются локальные имена задания. Перед пересылкой сообщения ядро системы преобразует локальные имена в значения, используемые получателем.

Каждый компонент содержит дескриптор типа, после которого следуют данные. Дескриптор представляет следующую информацию:

- **имя** (name). Указывает тип данных. Система Mach 2.5 распознает 16 различных значений этой переменной, в том числе внутреннюю память, права порта на отправку и получение или диапазон данных (byte, integer (16 и 32 разряда), string или real);
- **размер** (size). Размер наборов данных, присутствующих в компоненте;
- **количество** (number). Количество наборов данных компонента;
- **флаги** (flags). Указывает, находятся ли данные внутри сообщения или снаружи (out-of-line), а также необходимость освобождения памяти или прав на порты для задания-отправителя.

6.5.2. Интерфейс передачи сообщений

Приложения могут пользоваться службой передачи сообщений различными способами:

- отправлять сообщение, не ожидая ответа;
- ожидать незапрашиваемые сообщения и обрабатывать их по мере получения;
- отправлять сообщение и требовать ответа, но не ожидать его. В этом случае приложение получает ответы асинхронно и обрабатывает его через достаточно большой промежуток времени;
- отправлять сообщение и ожидать ответ.

Программный интерфейс средств передачи сообщений поддерживает три функции, которые при совместном применении обеспечивают вышеперечисленные способы коммуникаций [2].

```
msg_send (msg_header_t* hdr,
  msg_option_t option,
  msg_timeout_t timeout);

msg_rcv (msg_header_t* hdr,
  msg_option_t option,
  msg_timeout_t timeout);

msg_rpc (msg_header_t* hdr,
  msg_size_t rcv_size,
  msg_option_t option,
  msg_timeout_t send_timeout,
  msg_timeout_t receive_timeout);
```

Вызов `msg_send` используется для отправки сообщения, не требующего ответа. Этот вызов может быть заблокирован в случае отсутствия свободного

места в очереди сообщений принимающего порта. Функция msg_rcv также блокируется до тех пор, пока сообщение не будет получено. Каждый из этих вызовов поддерживает флаг SEND_TIMEOUT или RCV_TIMEOUT. При их задании функция может быть заблокирована на период, не превышающий значения timeout, задаваемого в миллисекундах. После исчерпания этого периода времени происходит возврат функции со статусом timed out (превышение лимита времени). Флаг RCV_NO_SENDERS заставляет функцию msg_rcv закончить работу, если больше ни одно задание не имеет прав на отправку сообщений в этот порт.

Вызов msg_rpc применяется для отправки исходящего сообщения в случае необходимости получения ответа на него. Эта функция представляет собой оптимизированный вариант использования msg_send и последующего msg_rcv. Ответ использует тот же буфер, где до этого содержалось исходящее сообщение. Функция msg_rpc поддерживает все опции вызовов msg_send и msg_rcv.

Заголовок сообщения включает в себя данные о его размере. При вызове msg_rcv заголовок будет показывать максимальный размер исходящего сообщения, поддерживаемый заданием, вызвавшим функцию. При возврате заголовок будет содержать данные о действительном размере полученного сообщения. При применении msg_rpc необходимо задавать максимальный размер отдельно в переменной rcv_size, так как заголовок сообщения в этом случае уже содержит размер исходящего сообщения.

6.6. Порты

Порты представляют собой защищенные очереди сообщений. Права на отправку или получение сообщений для портов могут задаваться заданиями. Доступ к портам разрешен только заданиям, обладающим соответствующими правами. Многие задания могут обладать правами на отправку сообщений в тот или иной порт, но только одно задание имеет право получать из порта. Обладатель прав на получение автоматически получает и право на отправку в этот порт.

В ОС Mach порты применяются для представления объектов системы, таких как задания, нити и процессоры. Правами на отправку и получение для таких портов обладает ядро. Порты имеют счетчики ссылок. Каждое право на отправку является ссылкой на объект, представляемый портом. Такие ссылки дают возможность обладателю производить действия над объектом. Например, порт задания task_self представляет само задание. Задание может отправлять сообщения на этот порт для запроса служб ядра. Если иное задание (чаще всего отладчик) также обладает правами на отправку в тот же порт, то оно вправе выполнять над этим заданием различные операции, такие как приостановка выполнения, при помощи отправки в порт сообщений.

В следующих разделах описываются диапазон имен и структуры данных, используемые для представления портов.

6.6.1. Диапазон имен портов

Каждое право порта представлено при помощи его имени. Имена портов — это всего лишь целые числа, диапазон возможных значений которых локален для каждого задания. Таким образом, один и тот же порт в разных заданиях может иметь различные имена. С этой точки зрения имена портов схожи с дескрипторами файлов UNIX.

Для каждого порта задания должно быть определено по крайней мере одно имя. Права портов могут передаваться при помощи сообщений, следовательно, задание может запросить право на один и тот же порт несколько раз. Ядро системы следит за тем, чтобы каждый раз использовалось одно и то же имя. В результате задание может сравнивать два имени порта. Если они не совпадают, то они не могут указывать на один и тот же порт.

Каждый порт также представляется при помощи глобальной структуры данных ядра. Ядро производит преобразование локального имени порта в глобальное, то есть на адрес его глобальной структуры данных этого порта, а также обратное преобразование. Если говорить о дескрипторах файлов, то в системах UNIX каждый процесс поддерживает таблицу дескрипторов в своей области u, в которой хранятся указатели на объекты открытых файлов. Однако в системе Mach принят другой метод преобразования, описание которого вы можете увидеть в разделе 6.6.3.

6.6.2. Структура данных порта

Ядро поддерживает структуру данных kern_port_t для каждого порта, содержащую следующую информацию:

- счетчик ссылок на все имена (права) порта;
- указатель на задание, являющееся обладателем прав на получение;
- локальное имя порта для задания-получателя;
- указатель на резервный порт. Если оригинальный порт будет освобожден, все сообщения будут передаваться на порт, указанный в качестве резервного;
- двунаправленный связанный список сообщений;
- очередь блокированных отправителей;
- очередь блокированных получателей. Хотя задание может обладать правами на получение, отдельные его нити могут оставаться в режиме ожидания сообщений;
- связанный список всех преобразований объекта;
- указатель на набор портов; указатели на следующий и предыдущий порты набора (если данный порт является частью набора, см. раздел 6.8.3);
- текущее число сообщений в очереди;
- максимальное поддерживаемое количество сообщений («журнал заказов»).

6.6.3. Преобразования портов

В Mach поддерживается одно вхождение преобразования для каждого права порта. Все вхождения должны включать в себя набор записей <task, port, local_name, type>, где task — задание, обладающее правом, port — указатель на структуру данных ядра, local_name — локальное имя порта, а переменная type соответствует получению или отправке. Система Mach использует эти вхождения для различных целей:

- вызов msg_send должен конвертировать <task, local_name> в port;
- вызов msg_rcv должен конвертировать <task, port> в local_name;
- если задание освобождает порт, ядро отыскивает все права порта;
- при удалении задания ядро отыскивает преобразования для этого задания и освобождает соответствующие ссылки;
- при удалении порта ядро отыскивает преобразования этого порта и предупреждает об этом все задания, обладающие правами на этот порт.

Для реализации этих механизмов необходима технология, которая бы эффективно поддерживала все перечисленные операции. На рис. 6.8 показаны структуры данных преобразования порта системы Mach 2.5. В ОС Mach применяются две глобальные таблицы хэширования для быстрого поиска вхождений: таблица TP_table хэширует вхождения, основанные на <task, port>, таблица TL_table хэширует их по кортежу <task, local_name>. Структуры данных kernel_port_t и task содержат заголовки связанных списков преобразований для порта и задания соответственно.

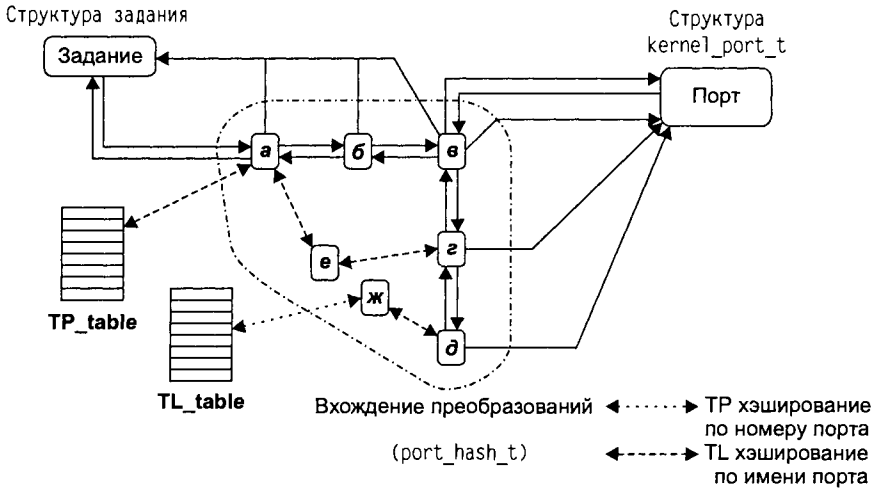


Рис. 6.8. Преобразование портов в ОС Mach

На рис. 6.8 вхождения *а*, *б* и *в* описывают преобразования различных портов одного задания, а вхождения *в*, *г* и *д* — преобразования одного и того

же порта для различных заданий. Вхождения *б*, *г* и *е* транслируются в одинаковый индекс в таблице TP_table[1], в то время как вхождения *д* и *ж* хэшируются одинаковым индексом в таблице TL_table. Каждое вхождение преобразования описывается структурой port_hash_t, содержащей следующую информацию:

- task (задание). Задание-владелец права;
- local_name (локальное имя). Имя права задания;
- type (тип). Отправка или получение;
- obj (объект). Указатель на объект порта в ядре системы.

Кроме этого, каждое вхождение преобразования находится в каждом из нижеперечисленных двунаправленных связанных списков:

- TP_chain. Цепочка хэширования, основанная на кортеже <task, port>;
- TL_chain. Цепочка хэширования, основанная на кортеже <task, local_name>;
- task_chain. Список всех преобразований, владельцем которых является задание;
- obj_chain. Список всех преобразований порта.

6.7. Передача сообщений

При передаче сообщения производятся следующие действия:

1. Сначала отправитель создает сообщение в своем адресном пространстве.
2. Для его отправки отправитель вызывает системную функцию msg_send. Порт назначения указывается в заголовке сообщения.
3. Ядро копирует сообщение во внутреннюю структуру данных (kern_msg_t) при помощи процедуры msg_copyin(). На этой стадии происходит преобразование прав порта в указатели на объекты порта ядра и копирование внешней памяти в карту хранения.
4. В дальнейшем действия могут отличаться в зависимости от условий:
   - если нить ожидает сообщение (находится в списке блокированных в ожидании приема из порта), она будет разбужена и сообщение будет передано ей напрямую;
   - в ином случае, если очередь сообщений окажется заполненной полностью, отправитель будет заблокирован до тех пор, пока из очереди не будет удалено сообщение;

[1] Согласно рисунку не «б», а «а». — *Прим. ред.*

- в ином случае сообщение будет поставлено в очередь порта, где оно будет находиться до тех пор, пока нить принимающего задания не загрузит msg_rcv.

5. Выход ядра из выполнения msg_send происходит после того, как сообщение будет поставлено в очередь или передано получателю напрямую.
6. Когда получатель вызывает системную функцию msg_rcv, ядро загружает процедуру msg_dequeue() для удаления сообщения из очереди. Если очередь окажется пустой, получатель будет заблокирован до тех пор, пока в ней не появится сообщение.
7. Затем ядро производит копирование сообщения в адресное пространство получателя при помощи функции msg_copyout(), которая производит дальнейшие преобразования внешней памяти и прав портов.
8. Во многих случаях отправитель сообщения требует ответа на него. Для того чтобы обратная передача была возможна, получатель должен обладать правами отправки в порт, владельцем которого является отправитель. Отправитель передает это право в поле *порта ответа* заголовка сообщения. При этом отправитель может воспользоваться вызовом mach_rpc для оптимизации обмена. Вызов mach_rpc эквивалентен вызову msg_send, за которым следует msg_rcv.

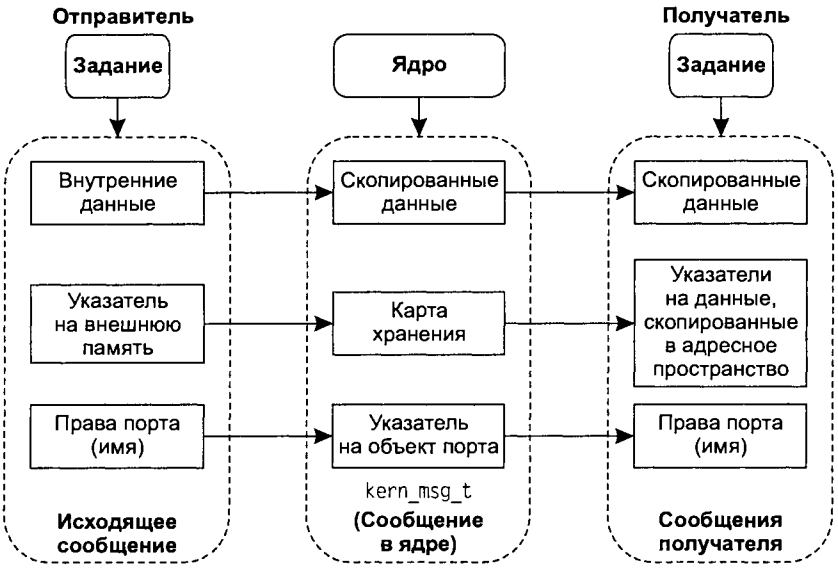


Рис. 6.9. Две стадии передачи сообщений

На рис. 6.9 показаны изменения различных компонентов сообщения, происходящие в процессе его передачи. Давайте рассмотрим некоторые аспекты передачи сообщения более подробно.

6.7.1. Передача прав порта

Существует несколько причин осуществления передачи прав портов через сообщения. Наиболее частым случаем является передача прав на порт ответа (рис. 6.10). Нить задания Н1 отправляет сообщение в порт П2, владельцем которого является задание Н2. В этом сообщении Н1 передает право на отправку для порта П1, для которого у нити Н1 имеется право на получение. В результате нить Н2 может отправить на порт П1 ответное сообщение, которое будет ожидать нить-отправитель. Описанная ситуация возникает настолько часто, что разработчики внесли в заголовок сообщения отдельное поле для хранения прав на порт ответа.

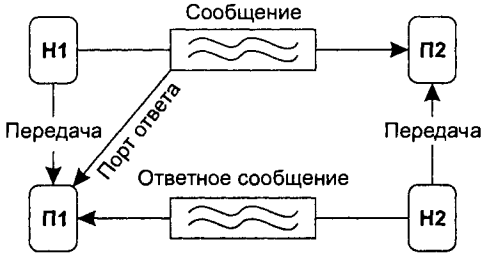


Рис. 6.10. Сообщение может содержать права отправки на порт ответа

Еще одной часто встречающейся ситуацией является взаимодействие между серверной программой, клиентом и сервером имен (рис. 6.11). Сервер имен хранит права отправки для некоторых серверных программ системы. Обычно серверные программы регистрируются в сервере имен самостоятельно сразу после начала работы (1). Все задания наследуют права отправки на сервер имен во время процедуры их создания (значение порта хранится в поле порта инициализации структуры задания).

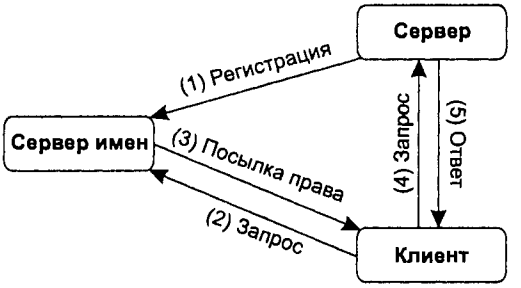


Рис. 6.11. Использование сервера имен для инициализации взаимодействия между клиентом и сервером

Если клиенту необходим доступ к серверной программе, он должен сначала получить право отправки в порт, владельцем которого является сервер. Для этого клиент делает запрос к серверу имен (1), который в ответ возвра-

тит право на отправку сообщений в его порт (3). Этим правом клиент воспользуется для передачи запроса серверу (4). Запрос содержит порт получения, который может быть использован сервером для отправки своего ответа клиенту (5). Все последующие взаимодействия между клиентами и серверами не требуют проведения повторных запросов к серверу имен.

Отправитель передает права порта, используя для него локальное имя. Дескриптор типа для этого компонента сообщения информирует ядро о том, что данное поле является правами порта. Локальное имя ничего не говорит получателю сообщения, следовательно, ядро системы должно перед отправкой преобразовать его. Для этого оно производит поиск вхождений преобразований в <task, local_name>, идентифицируя объект ядра (или глобальное имя) для этого порта.

При запросе сообщения ядру необходимо преобразовать глобальное имя в локальное имя, принятое в задании-получателе. Для этого ядро сначала проверяет, не имеет ли получатель прав на этот порт (хэшируя при этом <task, port>). Если это так, ядро преобразует его в указанное имя. В противоположном случае ядро выберет новое имя порта получателя и создаст вхождение преобразования, указывающее на этот порт. Имена портов обычно представляют собой целые небольшие числа, поэтому ядро будет при выборе имени использовать наименьшее целое свободное число.

Так как ядро системы создает для этого порта дополнительную ссылку, ему необходимо также и инкрементировать счетчик объектов порта. Ядро производит эту операцию при копировании сообщения в системное адресное пространство, так как новая ссылка создается именно в этот момент. Отправитель также может указать в дескрипторе типа флаг освобождения. Тогда ядро системы освободит право на порт в задании-отправителе и не будет увеличивать счетчик ссылок на этот порт.

6.7.2. Внешняя память

Сообщение может содержать небольшой объем данных. В этом случае оно может быть передано путем физического копирования информации, сначала в буфер ядра, а затем (при запросе сообщения) — в адресное пространство задания-получателя. Однако такой подход является непрактичным при передаче больших объемов данных. Система Mach позволяет передавать в одном сообщении содержание всего адресного пространства (то есть до 4 Гбайт информации в 32-разрядных системах), поэтому необходимо средство, позволяющее вести передачу таких значительных объемов информации более эффективно.

Обычно при передаче данных большого объема большинство из них не изменяется ни отправителем, ни их получателем. В таких случаях нет нужды создавать отдельные копии данных. Страница памяти может дублироваться

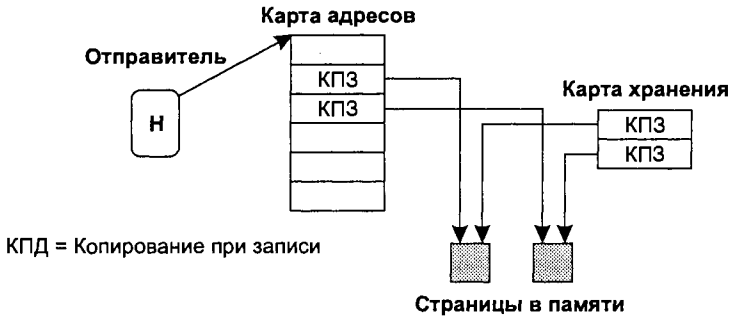
только в том случае, если одно или оба задания попытаются внести в нее изменения. До этого момента оба задания будут разделять между собой одну и ту же физическую копию страницы. В системе Mach это реализовано при помощи метода «копирования при записи» подсистемы виртуальной памяти. В главе 15 управление памятью в ОС Mach будет описано более подробно. В этом разделе мы расскажем только об аспектах управления, относящихся к средствам IPC.

На рис. 6.12 показана передача внешней памяти (out-of-line). Отправитель указывает на использование этого типа памяти при помощи флага в *дескрипторе типа*. Процедура `msg_copyin()`, вызываемая из `msg_send`, изменяет карты памяти отправителя, делая передаваемые страницы доступными только для чтения и копируемыми при записи. Затем для этих страниц создается временная карта хранения в ядре, и они также получают атрибуты «только для чтения» и «копирования при записи» (рис. 6.12, *а*). Если получатель вызывает `msg_rcv`, то функция `msg_copyout()` получает диапазон адресов в его пространстве и копирует туда вхождения из карты хранения страниц. Она также маркирует эти новые вхождения как доступные только в режиме чтения и копируемые при записи. Затем происходит освобождение временной карты хранения (рис. 6.12, *б*).

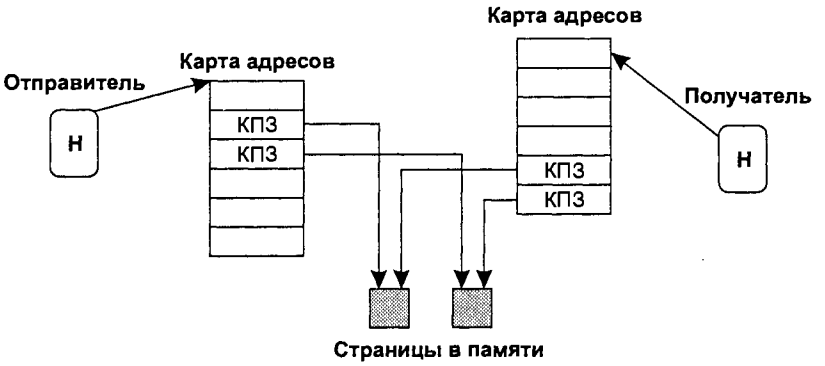
С этого момента отправитель и получатель используют страницы совместно в режиме копирования при записи. Если какое-либо из этих заданий попытается изменить страницу с таким атрибутом, результатом будет возникновение ошибки. Обработчик этой ошибки распознает установки и решает проблему при помощи создания копии страницы и изменения карты адресации задания на эту копию. Обработчик также изменяет атрибуты, разрешая отправителю и получателю создавать собственные копии страницы (рис. 6.12, *в*).

Следует запомнить, что передача внешней памяти минует две стадии. На первой стадии сообщение помещается в очередь, а страницы находятся в стадии пересылки. Позже, при запросе сообщения, страницы становятся разделяемыми между отправителем и получателем. Карты хранения используются на стадии передачи, что гарантирует создание ядром системы новой копии страницы для отправителя, если он попытается внести в нее изменения до того, как страница будет запрошена получателем. В этом случае отправитель будет иметь доступ к ее оригинальной копии.

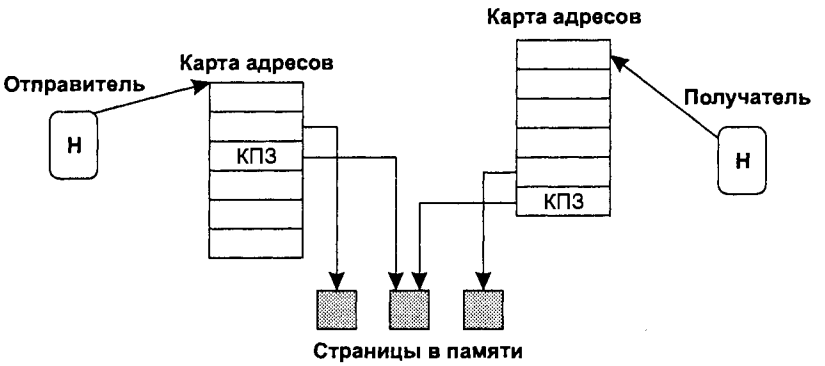
Описанное средство работает с максимальной производительностью, если ни отправитель, ни получатель не вносят изменения в совместно используемые страницы. Так функционируют многие приложения. *Внутренняя память копируется дважды: сначала от отправителя в ядро и затем из ядра получателю. Внешняя память копируется первый раз только тогда, когда одно из заданий попытается внести в нее изменения.*



а



б



в

Рис. 6.12. Передача внешней памяти: *а* — сообщение копируется в карту хранения; *б* — сообщение копируется в задание-получатель; *в* — получатель производит изменения в странице

Отправитель может указывать в дескрипторе типа флаг освобождения. В этом случае ядро не использует методику копирования при чтении. Оно просто копирует вхождения карты адресации в карту хранения во время вы-

полнения процедуры msg_copyin() и удаляет их из карты адресации отправителя. При запросе сообщения функция msg_copyout() производит копирование вхождений в карту адресации получателя и удаляет временную карту хранения. В результате происходит перемещение страницы из адресного пространства отправителя в пространство получателя без какого-либо копирования данных.

6.7.3. Управление нитью

Передача сообщений происходит по двум возможным путям — медленному и быстрому. Медленный путь применяется в том случае, если получатель не ожидает сообщение в момент его отправления. В таком случае отправитель помещает сообщение в очередь и затем заканчивает свою часть работы. После того как получатель вызовет msg_rcv, ядро удалит сообщение из очереди и скопирует его в адресное пространство получателя.

Каждый порт обладает настраиваемым ограничением, называемым *журналом заказов* (backlog), в котором указывается максимальное количество сообщений, хранящихся в очереди. Если их число достигнет предельного значения, порт будет считаться заполненным, а все новые отправители будут блокированы до тех пор, пока из очереди не будет получено сколько-нибудь сообщений. Каждый раз, когда происходит запрос одного сообщения из порта, приостановившего работу отправителей, выполнение одного из них будет продолжено. Следовательно, если из порта будут выбраны все сообщения, то будут разбужены все блокированные отправители.

По быстрому пути передача происходит тогда, когда получатель уже ожидает поступления сообщения. В этом случае функция msg_send не будет помещать сообщение в очередь порта. Вместо этого она разбудит получателя и передаст ему сообщение напрямую. В системе Mach реализовано средство под названием *автоматического* (hand-off) *планирования* [8], при использовании которого одна нить может освобождать процессор непосредственно для другой определенной нити. Код быстрой отправки сообщений использует это средство для переключения на нить-получатель, которая завершает вызов msg_rcv, используя функцию msg_copyout() для копирования сообщения в свое адресное пространство. Такой подход предупреждает перегрузки, которые могут возникнуть при постановке сообщения в очередь и удалении из нее. Это также ускоряет контекстное переключение, так как новая выполняемая нить выбирается непосредственно.

6.7.4. Уведомления

Уведомления — это асинхронные сообщения, посылаемые ядром для информирования задания об определенных событиях в системе. Эти сообщения от-

правляются в *порт уведомления* заданий. В Mach IPC применяются три типа уведомлений:

NOTIFY_PORT_DESTROYED	Это сообщение посылается владельцу резервного порта, если его основной порт был удален (в случае существования резервного порта). Удаление портов и резервные порты будут обсуждаться в следующем разделе
NOTIFY_PORT_DELETED	Рассылается всем заданиям, обладающим правом отправки на порт при его удалении
NOTIFY_MSG_ACCEPTED	Уведомление может быть запрошено отправителем, если он посылает сообщение в заполненную полностью очередь (используя опцию SEND_NOTIFY). Уведомление создается ядром после того, как сообщение удаляется из очереди

Последний тип уведомлений требует дополнительных уточнений. При использовании опции SEND_NOTIFY сообщение будет передано даже в том случае, если очередь будет заполнена полностью. Ядро возвращает статус SEND_WILL_NOTIFY, который указывает отправителю на необходимость приостановить передачу дальнейших сообщений до того, как им будет получено уведомление NOTIFY_MSG_ACCEPTED. Такой подход позволяет отправителю передавать сообщения без блокировки.

6.8. Операции порта

В этом разделе будут описаны некоторые операции, производимые над портами.

6.8.1. Удаление порта

Порт удаляется при освобождении прав на получение из него. Обычно это происходит при завершении работы задания, являющегося владельцем порта. Если порт необходимо аннулировать, то непосредственно после этой операции все задания, обладающие правами на отправку в этот порт, получают уведомление NOTIFY_PORT_DELETED. При удалении порта удаляются и все сообщения, находящиеся в его очереди, будятся все заблокированные отправители и получатели, им отправляются уведомления SEND_INVALID_PORT и RCV_INVALID_PORT соответственно.

Операция удаления порта является сложной процедурой, так как сообщения, находящиеся в его очереди, могут содержать права на другие порты. Если это права на получение, будут упразднены и порты, на которые они ссылаются. Фактически злоумышленник может отправить права на получение для определенного порта через сообщение на каждый порт. Такое случается нечасто, но если это происходит, результатом может стать возникновение взаимных блокировок и неограниченной рекурсии. К сожалению, в ОС Mach 2.5 эти проблемы так и не были решены.

6.8.2. Резервные порты

Системный вызов port_set_backup устанавливает *резервный порт* для указанного порта. Если порт, имеющий резерв, аннулируется, ядро системы не освобождает его, а передает все права на получение резервному порту.

Этот процесс показан на рис. 6.13. Порт П1 назначен в качестве резервного для порта П2. При удалении П1 (возможно, что это произошло из-за завершения работы задания, владеющего им) ядро системы посылает сообщение NOTIFY_PORT_DESTROYED владельцу П1. При этом порт П1 не освобождается и становится владельцем прав на отправку в порт П2. Все сообщения, адресуемые П1, автоматически перенаправляются на П2, откуда они могут быть затребованы его владельцем.

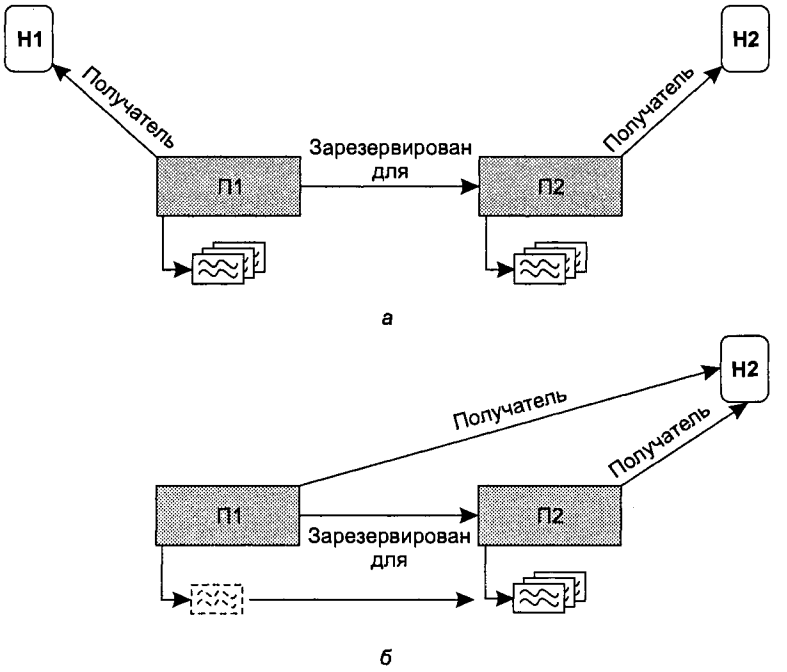


Рис. 6.13. Реализация резервных портов: *а* — назначение резервного порта, *б* — после удаления порта П1

6.8.3. Наборы портов

Набор портов состоит из группы портов, чьи индивидуальные права на получение заменены единственным групповым правом получения (рис. 6.14). Таким образом, задание может получать сообщения из набора портов. Более того, оно может получать их выборочно от определенных портов набора. В противоположность описанной возможности, сообщения могут отправляться толь-

ко на индивидуальные порты, а не на их наборы. Задания могут обладать правами отправки на определенные порты набора. При получении сообщения в нем будет содержаться информация о том порте, с которого оно было послано.

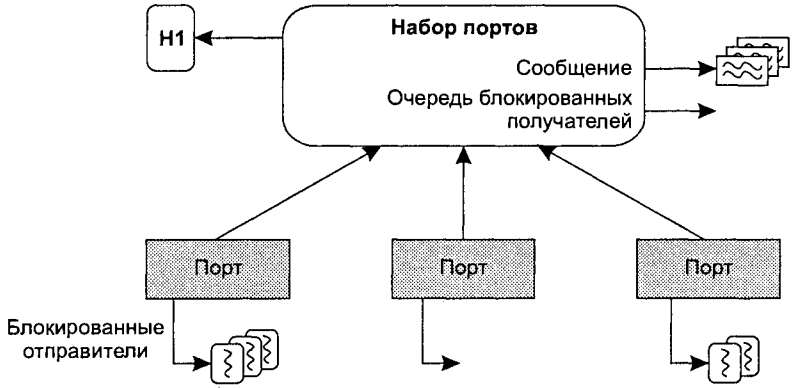


Рис. 6.14. Набор портов

Наборы портов удобны при обработке сервером сразу нескольких объектов. Серверы могут ассоциировать с каждым объектом отдельный порт, а затем объединить эти порты в набор. После этого сервер будет получать сообщения, отправленные на любой из портов набора. Каждое сообщение содержит в себе запрос, адресованный объекту, ассоциированному с портом. Так как в заголовке сообщения имеется информация о порте-получателе, сервер знает, с каким из объектов необходимо произвести манипуляции.

Возможности наборов портов сходны с вызовом системы UNIX select, позволяющим процессу проверить ввод на нескольких дескрипторах. Однако существует и важное отличие, заключающееся в том, что *время, потраченное на запрос сообщения из набора портов или на отправку сообщения в порт из набора, не зависит от общего количества портов в этом наборе.*

Объект ядра порта содержит указатель на набор, к которому относится этот порт, а также указатели на двунаправленный связанный список всех портов набора. Если порт не входит в набор, значения этих указателей равны NULL. Объект набора портов содержит единую очередь сообщений. Очереди входящих в набор портов не используются. Однако каждый порт продолжает поддерживать собственный счетчик и ограничение на количество сообщений, а также очередь блокированных отправителей.

Когда нить отправляет сообщение в порт, являющийся членом набора, ядро проверяет, не превышен ли лимит порта (если это так, отправитель будет добавлен в очередь блокированных отправителей порта) и увеличивает количество сообщений, находящихся в очереди. Затем ядро записывает имя порта в сообщение и переносит его в общую очередь набора портов. При вызове получателем функции msg_rcv ядро запрашивает первое сообщение из очереди набора независимо от конкретного порта, которому оно было адресовано.

6.8.4. Передача прав

Mach позволяет заданию произвести замену прав, относящихся к иному заданию, правами другого порта или получить право от другого задания. Такая возможность поручительства придала дополнительную гибкость управлению заданиями отладчиками и эмуляторами.

На рис. 6.15 показан возможный сценарий взаимодействия между отладчиком и отслеживаемым им заданием (жертвой). Сначала отладчик переносит право отправки жертвы на свой порт task_self при помощи вызова task_extract_send и забирает его себе. Затем он вызывает task_insert_send для передачи права отправки порту П1, владельцем которого является отладчик взамен права task_self жертвы. Точно так же отладчик использует вызовы task_extract_receive и task_insert_receive для передачи права на получение порта жертвы на свой порт task_self и подстановки права получения на другой порт П2, на который у отладчика имеются права отправки.

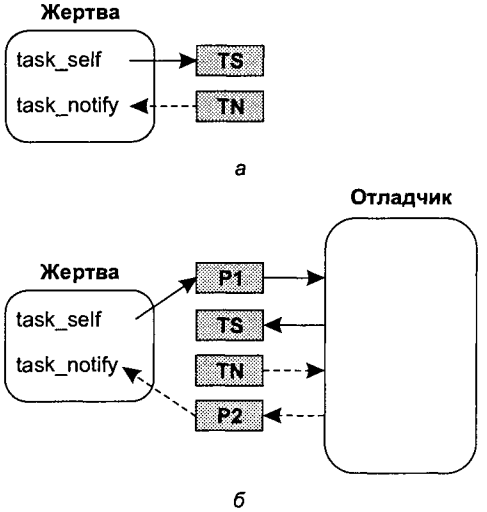


Рис. 6.15. Переназначение портов отладчиком: *а* — задача-жертва перед передачей прав; *б* — после передачи прав порта

После завершения этих действий ядро перехватывает все сообщения, отправляемые управляемым им заданием на порт task_notify (то есть все системные вызовы Mach). Отладчик обрабатывает вызов и проверяет отправку ответа на порт ответа, указанного в исходном сообщении. Отладчик может имитировать вызов и отправить ответ самостоятельно. Вместе с тем возможен и более стандартный ход: обработчик пересылает сообщение ядру при помощи изначально установленного порта task_notify задания (на который отладчик обладает правами отправки). При пересылке сообщения ядру ответ может быть послан напрямую заданию или на порт, указанный обработчиком. В последнем случае произойдет перехват ответа ядра.

С другой стороны, отладчик в состоянии перехватывать и все уведомления, передаваемые управляемому заданию, после чего он принимает решение о самостоятельной обработке уведомления или передаче его заданию. Из вышесказанного следует, что обработчик (или иное другое задание) может управлять заданием, если он передаст свои права на получение из порта task_self управляемого им задания.

6.9. Расширяемость

Средства IPC системы Mach разрабатывались с учетом возможности прозрачного расширения на распределенные среды. Для расширения Mach IPC на сеть используется программа пользовательского уровня netmsgserver, дающая возможность взаимодействия с заданиями на удаленных машинах точно так же, как будто эти задания выполняются на локальном компьютере. Приложения ничего не знают об удаленном соединении, так как они продолжают использовать тот же интерфейс и набор вызовов, применяемых для локальных коммуникаций. Приложение обычно не обладает информацией о том, является ли задание, с которым оно взаимодействует, локальным или удаленным.

В системе Mach имеются две очень важные особенности, позволившие прозрачно расширить возможности IPC на сеть. Во-первых, права портов используют независимое от местонахождения пространство имен. Отправитель посылает сообщение на локальное имя порта (право отправки), при этом он не знает, какой из объектов представлен этим портом — локальный или удаленный. Карты преобразований локальных имен портов в объект ядра поддерживаются на уровне ядра системы.

Во-вторых, отправители являются анонимными. Сообщения их не идентифицируют. Отправитель вправе переслать право на отправку в порт ответа прямо в сообщении. Ядро преобразует это таким образом, что получателю будет видно только локальное имя, действительное только для данного задания. Это означает, что получатель не может определить, кто является отправителем сообщения. Более того, получатель даже не обязан быть владельцем права на порт ответа, ему достаточно обладать правом на отправку в этот порт. Указывая порт ответа, владельцем которого является другое задание, отправитель перенаправляет ответ на это задание. Перечисленные возможности также могут применяться отладчиками, эмуляторами и другими программами.

Функции netmsgserver достаточно просты. Типичный сценарий работы программы показан на рис. 6.16. Программа netmsgserver загружается на каждой машине в сети. Если клиенту на узле А необходимо установить соединение с сервером на сервере Б, программа netmsgserver на узле А установит порт-посредник, на который отправитель будет отсылать сообщения. Затем она запросит сообщения, посланные на этот порт, и перешлет их программе netmsgserver узла Б, которая в свою очередь передаст их на порт сервера.

Если клиент ожидает ответа, он указывает в исходном сообщении порт ответа. Программа netmsgserver на узле А получит права отправки в ответный порт. После этого netmsgserver на узле Б создаст порт-посредник для ответного порта и отправит право на порт-посредник сервера. Сервер передает ответное сообщение на свой порт-посредник, которое далее будет передано через два netmsgserver (сервера и узла) на порт ответа клиента.

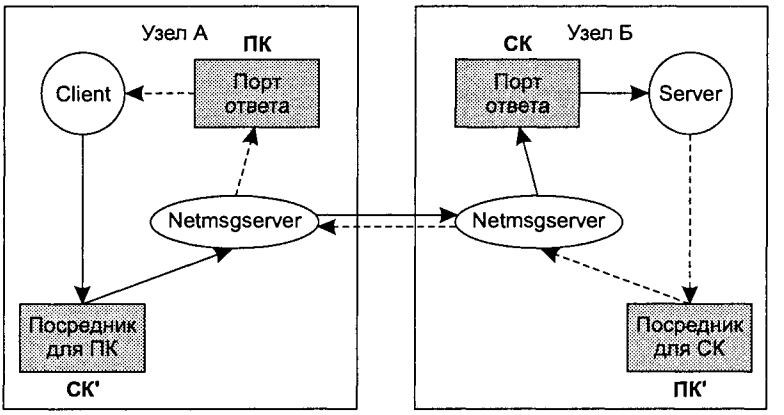


Рис. 6.16. Удаленные коммуникации с использованием программ netmsgserver

Серверы регистрируют себя в локальной программе netmsgserver и затем передают ей право отправки на порт, прослушиваемый сервером. Программа поддерживает распределенную базу данных таких зарегистрированных сетевых портов и предоставляет для них набор служб (защиту, уведомления и т. д.), сходный с набором, выделяемых ядром для локальных портов машины. Таким образом, программа поддерживает глобальную службу поиска для всех имен портов. Задания используют эту службу для получения прав отправки на порты, зарегистрированные удаленными программами netmsgserver. В отсутствии сети программа работает как локальная служба имен.

Программы netmsgserver взаимодействует между собой при помощи сетевых протоколов низкого уровня, не применяя при этом сообщений IPC.

6.10. Новые возможности Mach 3.0

В системе Mach 3.0 были расширены некоторые возможности средств IPC [7]. Были преодолены проблемы, возникавшие в ОС версии 2.5. Изменения коснулись как интерфейсной части, так и внутренних структур данных и используемых алгоритмов. В этом разделе показаны основные усовершенствования подсистемы IPC.

Главной проблемой Mach 2.5 является передача прав. Средства IPC тесно интегрируются с технологией «клиент-сервер». Обычно клиент посылает сооб-

щение, содержащее право отправки в порт ответа, владельцем которого он является. После генерации ответа серверу нет необходимости далее поддерживать посланное ему право. С другой стороны, сервер не может после обработки запроса освободить порт, так как существует вероятность использования этого порта другой нитью серверного приложения, обрабатывающей одновременно другой запрос от того же задания. Ядро преобразует одно и то же право отправки для следующих нитей в одно и то же имя, следовательно, сервер обладает единым правом отправки на указанный порт. Если нить, обработавшая первый запрос, освободит право, то вторая нить уже не сможет отправить на этот порт ответное сообщение.

Означенная проблема решена серверами просто: они никогда не освобождают присланные им права отправки. Однако это приводит к необходимости решения уже новых проблем. Во-первых, появляется определенный риск безопасности работы: клиент может не желать, чтобы сервер постоянно имел право на отправку сообщений в его порт. Также существует проблема усложнения обработки «раздутых» таблиц преобразований, что влияет на общую производительность системы. И наконец, при удалении порта клиентом серверу приходится обрабатывать не нужные ему уведомления об этом событии. Ядро обязано разослать такие уведомления всем серверам, которые имеют права на отправку в удаляемый порт, серверы, в свою очередь, должны получить и обработать их, даже если судьба порта их не интересует.

В системе Mach 3.0 введены новые средства, позволяющие решить эту проблему: права на однократную отправку, запросы уведомлений и пользовательские ссылки на права отправки.

6.10.1. Права на однократную отправку

Право на однократную отправку в порт — это, как и следует из его названия, право на отправку, которое может быть использовано только один раз. Такое право создается заданием-владельцем права на получение из порта и в дальнейшем может передаваться между заданиями. Оно дает гарантию на получение ответного сообщения именно из указанного в нем порта. Обычно такое право применяется для использования порта в качестве получателя ответного сообщения и отзывается сразу после передачи на него ответа. Если право аннулируется вследствие других причин (например, при завершении работы его владельца), то вместо ответа ядром будет отослано однократное уведомление.

В системе Mach права на однократную отправку используются для указания портов ответа. Такое право удаляется сразу после получения ответного сообщения, следовательно, теперь сервер не продолжает удерживать не нужные ему права. Такой подход защищает от необходимости обработки излишних уведомлений, создаваемых при удалении порта клиентом.

6.10.2. Уведомления в Mach 3.0

В системе Mach 2.5 уведомления отправлялись заданиями асинхронно в ответ на различные системные события. Задания не могли контролировать процесс получения таких уведомлений. В большинстве случаев информация о происшедших событиях не нужна заданиям, поэтому уведомления просто уничтожаются. Однако существование таких излишних уведомлений влияет на общую производительность системы.

Более того, использование единого порта для уведомления всех нитей задания приводит к определенным ограничениям. Одна из нитей может перехватить и изъять уведомление, ожидаемое другой нитью задания. Применение пользовательских библиотек увеличило вероятность такого варианта развития событий, так как основные программы и библиотеки используют уведомления независимо друг от друга и обрабатывают их различными способами.

В системе Mach 3.0 уведомления отправляются только тем заданиям, которые запросили их при помощи вызова mach_port_request_notification. При этом запросе используется право на однократную отправку в порт уведомлений. Таким образом, в ОС Mach 3.0 появилась возможность использования нескольких портов уведомлений внутри одного задания. Каждый программный компонент или нить может теперь указать собственный порт уведомлений, защитив себя тем самым от вышеуказанных проблем.

6.10.3. Подсчет прав на отправку

Если задание дает несколько прав отправки для одного порта, ядро использует для всех них единое имя в пространстве имен портов, комбинируя тем самым все права в единое право на отправку. Если одна из нитей освободит право, ни одна другая нить больше не сможет воспользоваться им, даже если эта нить затребовала данное право ранее самостоятельно.

Применение прав на однократную отправку помогло решить проблему, но только частично. Этого рода права используются по большей мере для задания портов ответа серверам. Права на отправку можно получить несколькими способами. Если клиенту необходимо начать взаимодействие с сервером, он запросит право на отправку к нему сообщений (через сервер имен). Если несколько нитей клиента независимо друг от друга инициализируют общение с сервером, каждая из них получит право на получение, которое затем будет объединено одним общим именем. Если одна из нитей освободит это имя, то это действие повлияет и на все другие нити: никто из них не сможет в дальнейшем пользоваться портом.

В системе Mach 3.0 эта проблема была решена при помощи ассоциации пользовательской ссылки с каждым правом отправки. Теперь ядро увеличивает значение счетчика ссылок на единицу при каждом запросе заданием одного и того же права и точно так же уменьшает его при каждом освобождении права. После освобождения последней ссылки ядро удаляет право.

6.11. Дискуссия о средствах Mach IPC

В системе Mach возможности IPC являются не только средствами осуществления коммуникаций между процессами, но и фундаментальным компонентом структуры ядра. Например, подсистема виртуальной памяти использует их для реализации технологии копирования при записи [15], а ядро применяет IPC для управления заданиями и нитями. Основные объекты системы Mach, такие как задания, нити и порты, используют для взаимодействия друг с другом средства передачи сообщений.

Архитектура ОС Mach имеет интересные свойства. Средства IPC построены таким образом, что при помощи программы `netmsgserver` можно расширить возможности взаимодействия между процессами на распределенные системы, позволяя заданиям управлять объектами на удаленных узлах. Это свойство системы Mach позволило реализовать такие средства, как удаленная отладка, распределенная разделяемая память и другие клиент-серверные программы.

Однако если посмотреть на архитектуру системы с другой точки зрения, мы увидим, что интенсивная передача сообщений ведет к ухудшению производительности. Разработчики направили свои усилия на построение операционных систем с микроядром, в которых большинство возможностей реализовывалось на уровне пользовательских серверных заданий, взаимодействующих друг с другом при помощи IPC. Над созданием подобных систем трудились многие производители, но проблемы производительности практически свели на нет их усилия: такие системы не получили широкого распространения.

Сторонники архитектуры Mach считали, что производительность IPC не является важным фактором, который следует рассматривать при разработке ОС с микроядром [4]. Они объясняли это следующими причинами:

- достижения в улучшении производительности IPC и так велики по сравнению с другими компонентами операционной системы;
- с увеличением доверия к аппаратным кэшам стоимость системных служб будет сильно зависеть от их емкостей. Так как код подсистемы IPC является хорошо локализированным, он может быть легко настроен для оптимального использования кэш-памяти;
- некоторый объем передачи данных можно осуществить другими путями, например, используя механизмы разделяемой памяти;
- перевод некоторых возможностей ядра на уровень серверов пользовательского уровня позволил уменьшить количество переключений режима и выходов за границы защиты, являющихся весьма затратными операциями.

Разработчики потратили много усилий на увеличение производительности средств IPC [3], [4]. Но несмотря на это подсистема IPC получила лишь ограниченное распространение среди коммерческих ОС. Даже в системе Digital UNIX, разработанной на основе Mach, в большинстве компонентов ядра средства IPC Mach не применяются.

6.12. Заключение

В этой главе описывалось сразу несколько вариантов механизмов взаимодействия процессов (IPC). Сигналы, каналы и трассировка процессов — это универсальные средства, поддерживаемые всеми ранними версиями системы UNIX. В наборе System V IPC были представлены новые возможности, такие как разделяемая память, семафоры и очереди сообщений, которые после появились во всех современных вариантах UNIX. В ОС Mach средства IPC используются для взаимодействия всех объектов ядра друг с другом. Границы взаимодействия объектов Mach могут быть легко расширены при помощи программы netmsgserver, дающей возможность построения распределенных клиент-серверных приложений.

Некоторые другие средства IPC будут описаны в книге позже: в главе 8 вы увидите рассказ о защите файлов, в главе 14 прочтете о файлах, отображаемых в память, а глава 17 расскажет о каналах библиотеки STREAMS.

6.13. Упражнения

1. Назовите ограничения ptrace как инструмента, используемого для создания отладчиков.
2. Аргумент pid функции ptrace должен указывать на идентификатор процесса, являющегося потомком вызвавшего ее. Зачем необходимо выполнение этого требования? Почему процессы не могут использовать ptrace для взаимодействия с любыми другими процессами системы?
3. Сравните коммуникационные возможности, предоставляемые каналами и очередями сообщений. Какими преимуществами и недостатками обладает каждое из упомянутых средств? В каких случаях предпочтительнее использовать каналы, а в каких — очереди сообщений?
4. Во многих системах UNIX возможно подключение одной и той же разделяемой области памяти сразу на несколько адресов в адресном пространстве. Это недоработка или, наоборот, преимущество систем? В каких случаях описанная возможность может быть использована? К возникновению каких проблем это может привести?
5. О чем должен помнить программист при выборе адреса для подключения к нему разделяемой области памяти? От каких ошибок способна при этом защитить сама операционная система?
6. Каким образом можно задействовать флаг IPC_NOWAIT для предупреждения возникновения взаимных блокировок при использовании семафоров?
7. Создайте программы, позволяющие взаимодействующим процессам эксклюзивно использовать ресурс:
   - на основе канала;
   - FIFO;

- системного вызова mkdir;
- системных вызовов flock и lockf.

Сравните созданные программы и их производительность.

8. С какими возможными побочными эффектами может столкнуться программист при создании всех вариантов, предложенных в предыдущем вопросе?
9. Можно ли реализовать блокировку ресурса, используя:
   - только сигналы;
   - сигналы и разделяемую память?

   Какой производительностью будут обладать эти средства?
10. Создайте программы, передающие большой объем данных между процессами, используя
    - каналы;
    - FIFO;
    - очереди сообщений;
    - разделяемую память совместно с семафорами (для синхронизации).

    Сравните созданные программы и их производительность.
11. Какие проблемы безопасности существуют в средствах IPC System V? Каким образом может вредоносная программа вмешаться во взаимодействие процессов или перехватить информацию?
12. Семафоры создаются при помощи semget, но для их инициализации необходимо применить вызов semctl. Следовательно, инициализация и создание семафора не могут быть проведены как единая неделимая операция. Опишите ситуацию, при которой такой подход может привести к состязательности. Предложите способ решения создавшейся проблемы.
13. Можно ли реализовать очереди сообщений System V на основе Mach IPC? Какие проблемы могла бы решить такая реализация?
14. По каким причинам может быть удобно применение прав на однократную отправку?
15. Каким образом наборы портов помогают разработчикам в создании клиент-серверных приложений для ОС Mach?

6.14. Дополнительная литература

1. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
2. Baron, R. V., Black, D., Bolosky, W., Chew, J., Draves, R. P., Golub, D. B., Rashid R. F., Tevanian, A., Jr., and Young, M. W., «Mach Kernel Interface

Manual», Department of Computer Science, Carnegie-Mellon University, Jan. 1990.

3. Barrera, J. S., «A Fast Mach Network IPC Implementation», Proceedings of the USENIX Mach Symposium, Nov. 1991, pp. 1–12.
4. Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M., «Lightweight Remote Procedure Call», ACM Transactions on Computer Systems, Vol. 8, No 1, Feb. 1990, pp. 37–55.
5. Bershad, B. N., «The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems», USENIX Workshop on Micro-Kernels and Other Kernel Architectures, Apr. 1992, pp. 205–212.
6. Dijkstra, E. W., «Solution of a Problem in Concurrent Programming Control», Communications of the ACM, Vol. 8, Sep. 1965, pp. 569–578.
7. Draves, R. P., «A Revised IPC Interface», Proceedings of the First Mach USENIX Workshop, Oct. 1990, pp. 101–121.
8. Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W., «Using Continuations Implement Thread Management and Communication in Operating Systems», Technical Report CMU-CS-91-115R, School of Computer Science, Carnegie Mellon University, Oct. 1991.
9. Faulkner, R. and Gomes, R., «The Process File System and Process Model in UNIX System V», Proceedings of the 1991 Winter USENIX Conference, Jan. 1991, pp. 243–252.
10. Presotto, D. L., and Ritchie, D. M., «Interprocess Communications in the Ninth Edition UNIX System», UNIX Research System Papers, Tenth Edition, Vol. II, Saunders College Publishing, 1990, pp. 523–530.
11. Rashid, R. F., «Threads of a New System», UNIX Review, Aug. 1986, pp. 37–49.
12. Salus, P. H., «A Quarter Century of UNIX», Addison-Wesley, Reading, MA, 1994.
13. Stevens, R. W., «UNIX Network Programming», Prentice-Hall, Englewood Cliffs, NJ, 1990.
14. Thompson, K., «UNIX Implementation», The Bell System Technical Journal, Vol. 57, No. 6, Part 2, Jul.–Aug. 1978, pp. 1931–1946.
15. Young, M., Tevanian, A., Rashid, R. F., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R., «The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System», Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Nov. 1987, pp. 63–76.

Глава 7
Синхронизация. Многопроцессорные системы

7.1. Введение

Желание увеличить производительность компьютерных систем привело к появлению новых аппаратных архитектур. Основным направлением развития вычислительной техники стало создание многопроцессорных систем. Такие системы могут содержать от двух и более центральных процессоров, при этом остальные ресурсы, такие как память, используются ими совместно. Многопроцессорные технологии обладают некоторыми бесспорными преимуществами. Они позволяют гибко расширять объем ресурсов, используемых программой, которая может начать выполнение на одном процессоре и в дальнейшем, при увеличении объема производимых вычислений, использовать другие процессоры системы. Большинство вычислительных задач зависят, прежде всего, от производительности CPU. Для таких задач мощность процессора (или процессоров) является самым критичным параметром, в отличие от работы с шиной ввода-вывода или памятью, используемых менее интенсивно. Применение многопроцессорных систем позволяет увеличивать вычислительные мощности без дублирования остальных ресурсов. Следовательно, такое решение для вычислительных задач является самым эффективным по себестоимости.

Многопроцессорные системы обладают повышенной надежностью: если на одном из процессоров произойдет сбой, система сможет продолжать работу дальше. Но, с другой стороны, многопроцессорные архитектуры таят в себе и определенные проблемы, связанные с большим количеством потенциально опасных точек сбоев. Чтобы гарантировать определенное количество *среднего времени наработки на отказ* (mean time before failure, MTBF), многопроцессорные системы должны разрабатываться на основе отказоустойчивого оборудования и программного обеспечения. В частности, многопроцессорные системы должны уметь восстанавливаться после сбоя одного из процессоров.

Существуют различные операционные системы, созданные с учетом многопроцессорных архитектур. Одна из первых многопроцессорных реализаций UNIX работала на машинах AT&T 3B20A и IBM 370 [1]. На сегодняшний день большинство вариантов систем UNIX поддерживают многопроцессорную обработку изначально (DEC UNIX, Solaris 2.x), либо для этой цели имеются специализированные версии ОС (SVR4/MP, SCO/MPX).

Теоретически производительность системы должна увеличиваться пропорционально умножению числа процессоров. Однако в реальности все происходит немного иначе. На это существует несколько причин. Так как остальные компоненты системы не дублируются, именно они и становятся узким местом, тормозящим ее работу. Доступ к разделяемым структурам данных требует определенной синхронизации. Для многопроцессорных систем необходимо обеспечивать синхронизацию при помощи дополнительных средств, увеличивающих загрузку процессоров и уменьшающих тем самым общую производительность. Операционная система обязана по возможности уметь минимизировать такие перегрузки и оптимизировать использование ресурсов CPU.

Ядро традиционных вариантов UNIX разрабатывалось для однопроцессорных машин. Для того, чтобы такие системы работали на многопроцессорных архитектурах, требуется внести в дизайн их ядра значительные изменения. Необходимо заново переписать такие компоненты, как реализация синхронизации, параллельного выполнения задач и правил планирования. Синхронизация используется для управления доступом к разделяемым данным и ресурсам системы. Традиционный подход, основанный на приостановке и возобновлении работы процессов, совмещенный с блокировкой прерываний, является непригодным для многопроцессорных сред и должен быть заменен более производительными методиками.

Параллельность позволяет эффективно использовать объекты синхронизации для управления доступом к разделяемым ресурсам. Эта технология предлагает решения, позволяющие регулировать степень детализации и место блокировок, защищать от возникновения взаимоблокировок и т. д. Некоторые из перечисленных проблем будут описаны в разделе 7.10. Для оптимизации использования всех процессоров системы необходимо разработать новые правила планирования. Некоторые аспекты реализации планирования в многопроцессорных системах будут обсуждаться в разделе 7.4.

Эта глава начинается с описания механизмов синхронизации традиционных систем UNIX и анализа их ограничений. Затем вы познакомитесь со многопроцессорными архитектурами. В конце главы приведено описание технологий синхронизации, применяемых в современных вариантах UNIX. Эти методы работают одинаково хорошо как на однопроцессорных, так и на многопроцессорных системах.

В традиционных вариантах UNIX базовым объектом диспетчеризации является процесс, который обладает единственным управляемой нитью. В главе 3 описывались современные системы UNIX, которые позволяют процессу иметь

несколько нитей выполнения, поддерживаемых на уровне ядра. Многонитевые системы существуют как для однопроцессорных, так и для многопроцессорных архитектур. В таких системах нити соревнуются за обладание разделяемыми ресурсами и блокируют их. В дальнейшем (до окончания главы) мы будем считать нить базовой единицей планирования. Для однонитевых систем нить является синонимом процесса.

7.2. Синхронизация в ядре традиционных реализаций UNIX

Ядро UNIX является реентерабельным. Это означает, что в одно и тоже время в ядре системы могут работать сразу несколько процессов, иногда выполняя при этом одинаковые задачи. На однопроцессорных системах в один момент времени может выполняться только один процесс. Система постоянно переключается от выполнения одного процесса к другому, создавая иллюзию параллельного функционирования. Такая возможность получила название *многозадачной работы*. Так как все процессы используют единое ядро, ему необходимо как-то синхронизировать доступ к своим структурам данных для предупреждения возможности их повреждения. В разделе 2.5 подробно рассказывалось о технологиях синхронизации, применяемых в традиционных системах UNIX. В этой главе мы подведем краткие итоги.

Первым защитным механизмом традиционного ядра UNIX является его невытесняемость. Любая нить будет продолжать работу в режиме ядра до тех пор, пока сама не будет готова освободить его либо приостановить выполнение в ожидании ресурса, даже в том случае, если эта нить исчерпала выделенный ей квант времени. Такой подход дает возможность кодам ядра управлять различными структурами данных без обеспечения какой-либо защиты содержащейся в них информации, поскольку заранее известно, что никакая другая нить не сможет получить доступ к ним до тех пор, пока текущая нить не закончит манипуляции данными и не переведет ядро в непротиворечивое состояние.

7.2.1. Блокировка прерываний

Непрерываемость ядра традиционных версий UNIX является свойством, обеспечивающим мощный инструментарий синхронизации, но, к сожалению, она обладает и некоторыми ограничениями. Текущую выполняемую нить невозможно вытеснить, но ее работу можно прервать. Прерывания являются внутренними событиями системы и должны обрабатываться как можно быстрее. Обработчик прерывания в состоянии манипулировать структурами данных, принадлежащих текущей нити, что может стать причиной их поврежде-

ния. Следовательно, ядро системы должно как-то синхронизировать доступ к данным, используемым его подпрограммами и обработчиками прерываний.

В системе UNIX вышеописанная проблема решена при помощи блокировки (или маскирования) прерываний. Каждому прерыванию присваивается *уровень приоритета* (interrupt priority level, ipl). Система поддерживает текущий уровень ipl и проверяет его после возникновения прерывания. Если приоритет окажется выше текущего значения, то такое прерывание будет обработано незамедлительно (то есть будет приостановлена обработка прерывания с более низким уровнем ipl). В противоположном случае ядро заблокирует прерывание до тех пор, пока текущий уровень ipl не снизится до необходимого уровня. Сразу перед загрузкой обработчика ядро системы устанавливает ipl в значение текущего прерывания, после завершения его обработки ядро восстанавливает предыдущее сохраненное значение. Ядро обладает возможностью установить значение ipl в принудительном порядке с целью временной блокировки прерываний в течение выполнения критических участков кода.

Например, одной из процедур ядра необходимо удалить дисковый буфер из очереди буферов. Эта очередь доступна и обработчику прерываний диска. Участок кода, производящий действия с очередью, является *критическим*. До начала его выполнения процедура поднимает уровень ipl до значения, достаточного для блокировки дисковых прерываний. После завершения действий над очередью процедура восстановит предыдущее значение ipl, разрешая тем самым обработку дисковых прерываний. Поддержка уровней ipl позволяет эффективно решить проблему синхронизации ресурсов, используемых совместно ядром и обработчиками прерываний.

7.2.2. Приостановка и пробуждение

Во многих случаях нить должна использовать какой-либо ресурс системы эксклюзивно даже в том случае, если она приостанавливает выполнение. Например, нити необходимо прочесть блок информации с диска и записать ее в дисковый буфер. Для этого нить запрашивает буфер и начинает производить действия с диском. Ей необходимо дождаться завершения ввода-вывода, что означает возможность освобождения процессора на некоторое время для другой нити. Если новая текущая нить попытается получить доступ к тому же буферу и использовать его для своих целей, результатом может стать изменение или повреждение данных, находящихся в нем. Это означает, что система должна поддерживать некие методики защиты блокированных ресурсов.

В UNIX защита ресурсов реализована при помощи флагов locked (ресурс занят) и wanted (ресурс необходим). Если нити необходимо получить доступ к разделяемому ресурсу (например, буферу), то в первую очередь ей нужно проверить состояние флага locked. Если флаг не установлен, нить установит

его и начнет манипуляции с ресурсом. Если другая нить попытается получить доступ к тому же ресурсу, она обнаружит флаг locked и приостановит работу («заснет») до тех пор, пока ресурс не станет доступным. Перед блокировкой нить установит для ресурса флаг wanted. Приостановка работы нити означает помещение ее в очередь спящих нитей и изменение информации о ее статусе, оповещающей о приостановке работы в ожидании определенного ресурса. После этого нить освобождает процессор и передает его следующей работоспособной нити системы.

Через некоторое время первая нить закончит манипуляции с ресурсом, сбросит флаг locked и проверит флаг wanted. Если флаг окажется установленным, то это означает, что по крайней мере еще одна нить находится в режиме ожидания того же ресурса. В этом случае текущая нить проверит очередь спящих нитей и разбудит все нити, ожидающие освобожденный ею ресурс. Пробуждение нити приводит к удалению ее из очереди спящих нитей, изменению ее статуса в состояние «работоспособная» и помещению ее в очередь планировщика. В какой-то момент времени нить станет текущей. Первым ее действием при получении процессорного времени будет являться проверка флага locked. Если флаг не установлен, нить сможет продолжить манипуляции с необходимым ей ресурсом.

7.2.3. Ограничения традиционного ядра UNIX

Традиционная модель синхронизации корректно работает на однопроцессорных машинах, но она, к сожалению, не избавлена от проблем, связанных с производительностью системы. В многопроцессорных системах использование традиционной модели невозможно, что и будет более подробно показано в разделе 7.4.

Блокировка ресурсов и очереди приостановленных нитей

В некоторых ситуациях организация очередей приостановленных нитей может являться причиной заметного ухудшения производительности системы. В системах UNIX нить блокируется при ожидании освобождения ресурса или возникновении определенного события. Каждый ресурс или событие ассоциированы с каналом ожидания, представляющим собой 32-разрядную переменную, обычно указывающую на адрес ресурса. Система поддерживает набор очередей ожидания, соединение канала с одной из очередей реализовано при помощи хэширования (рис. 7.1). Нить приостанавливает работу, помещая себя в одну из очередей ожидания, при этом ссылка на соответствующий канал ожидания сохраняется в структуре proc.

Применение такой методики блокировки приводит к двум последствиям. Во-первых, на каждый канал может ссылаться более чем одно событие. Например, одна из нитей может заблокировать буфер, инициализировать проведение над ним определенных действий и приостановить работу до тех пор,

пока эти действия не будут закончены. Если другая нить попытается получить доступ к тому же буферу, то обнаружит его заблокированным. Следовательно, ей придется приостановить выполнение в ожидании освобождения ресурса. Оба события ссылаются на один и тот же канал, указывающий на необходимый обеим нитям буфер. После завершения процедур ввода-вывода обработчик прерываний разбудит обе нити, несмотря на то, что последняя из них ожидает еще не произошедшее событие.

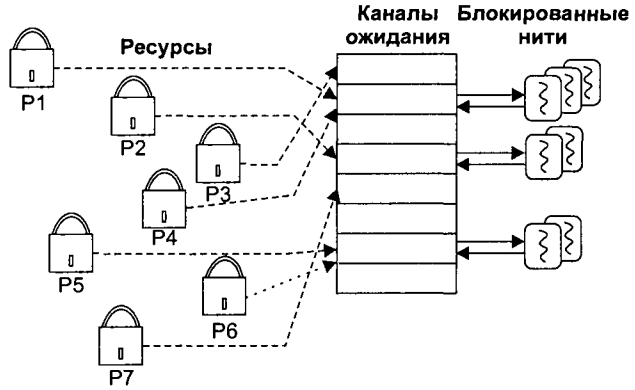


Рис. 7.1. Отображение ресурсов в глобальных очередях приостановленных нитей

Во-вторых, количество структур хэширования обычно меньше, чем различных каналов ожидания (то есть ресурсов и событий), следовательно, несколько каналов могут ссылаться на одно и то же место. Таким образом, оказаться хэшированными могут нити, ожидающие несколько различных каналов. Процедуре wakeup() необходимо проверять каждый из них и делать работоспособными только те нити, которые ожидают определенный ресурс. Общее время работы wakeup() зависит не от количества процессов, находящихся в режиме ожидания данного канала, а от общего количества хэшированных процессов. Это приводит к непредусмотренным задержкам, появление которых нежелательно для систем, поддерживающих приложения реального времени, требующих соблюдения определенных верхних границ задержек планировщика (см. раздел 5.5.4).

Одним из решений проблемы является ассоциация отдельной очереди ожидания каждому ресурсу или событию (рис. 7.2), что позволит оптимизировать задержки при пробуждении процессов, но приведет к необходимости выделения большего объема памяти для хранения большего количества очередей. Обычный заголовок очереди наряду с другой информацией имеет два указателя (на предыдущую и следующую очереди). Общее количество объектов синхронизации может оказаться слишком большим, так что поддержка отдельной очереди ожидания для каждого из них является неприемлемой.

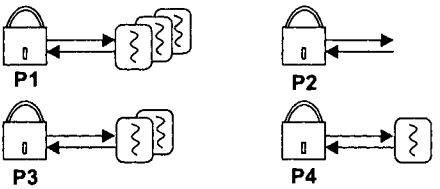


Рис. 7.2. Отдельные очереди для каждого ресурса системы

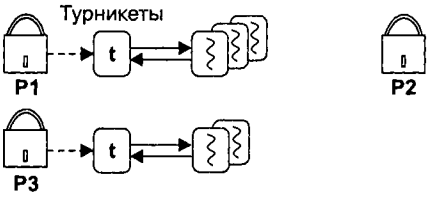


Рис. 7.3. Блокировка нитей с применением турникетов

В системе Solaris 2.x представлено более эффективное решение [7]: каждый объект синхронизации поддерживает двухразрядное поле, указывающее на структуру турникета, которая содержит очередь ожидания и другую необходимую информацию (рис. 7.3). Ядро выделяет турникеты только для тех ресурсов, которые ожидают заблокированные нити. Для ускорения процедуры выделения ядро поддерживает набор турникетов, размер которого больше, чем общее количество активных нитей в системе. Такой подход удобен для приложений реального времени, так как дает минимальные задержки. Более подробно технология синхронизации системы Solaris 2.x описана в разделе 5.6.7.

Разделяемый и эксклюзивный доступ

Механизм приостановки и возобновления выполнения является удобным в том случае, если в один момент времени ресурс используется только одной нитью. Однако он не позволяет использовать более сложные протоколы, такие как синхронизация читающих-пишущих нитей. Иногда необходимо предоставить ресурс сразу нескольким нитям на чтение, но потребовать для его изменения эксклюзивных полномочий. Некоторые ресурсы, такие как файлы или каталоги, будут использоваться более эффективно при применении этой технологии.

7.3. Многопроцессорные системы

Многопроцессорные системы обладают тремя важнейшими свойствами. Во-первых, это модель памяти, определяющая способ ее разделения между всеми процессорами. Второй отличительной особенностью является аппаратная поддержка синхронизации. Под третьим свойством подразумевается программная архитектура, определяющая взаимодействие процессоров, подсистем ядра и пользовательских процессов.

7.3.1. Модель памяти

С точки зрения аппаратуры многопроцессорные системы можно разделить на три категории в зависимости от доступа к памяти и взаимосвязи компонентов (рис. 7.4):

- **с унифицированным доступом к памяти** (Uniform Memory Access, UMA);
- **с неунифицированным доступом к памяти** (Non-Uniform Memory Access, NUMA);
- **с невозможностью доступа к удаленной памяти** (No Remote Memory Access, NORMA).

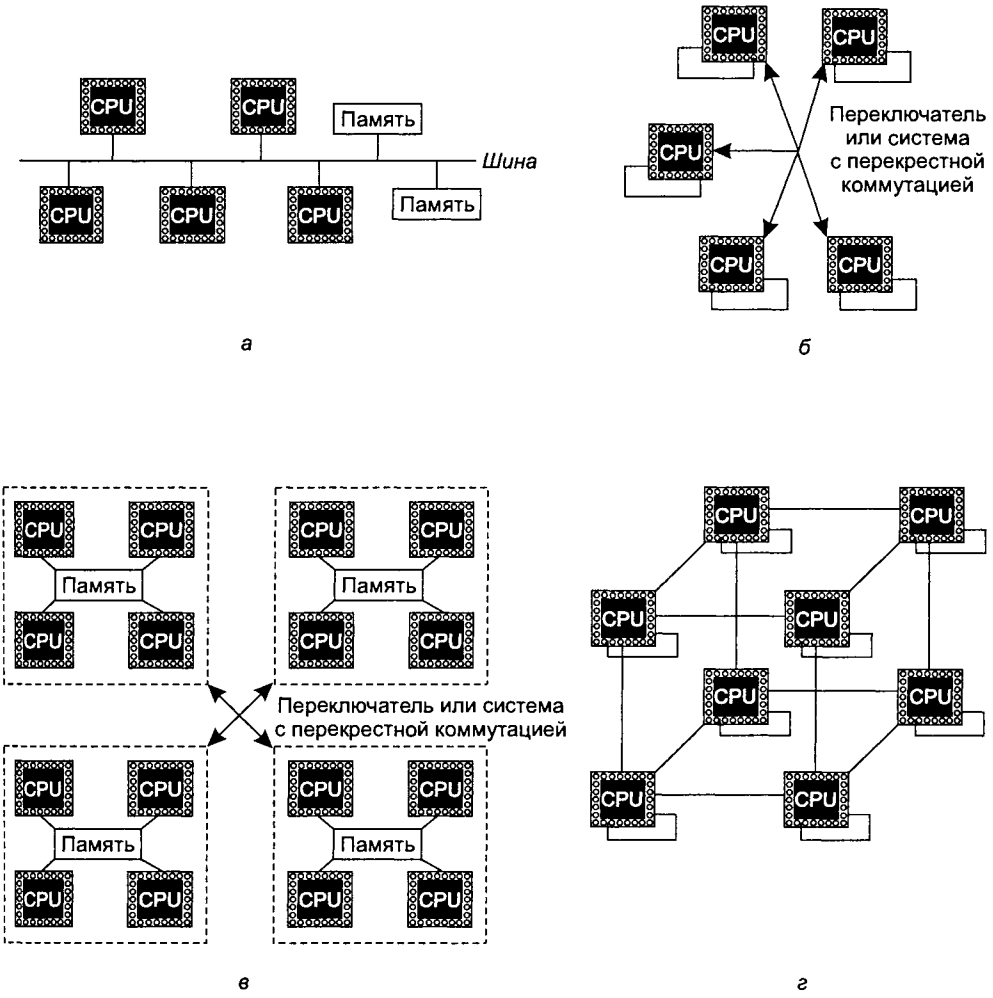


Рис. 7.4. Категории систем: *а* — UMA, *б* — NUMA, *в* — гибридная архитектура NUMA, *г* — NORMA

Наиболее распространенные системы основаны на методе UMA или разделяемой памяти (рис. 7.4, *а*). В таких системах все процессоры обладают равнозначным доступом к основной памяти[1] и устройствам ввода-вывода, которые обычно соединяются между собой единой шиной. С точки зрения разработки операционных систем эта модель является наиболее простой. Ее основным недостатком является масштабируемость. Архитектура UMA поддерживает небольшое количество процессоров. При увеличении числа CPU возникают конфликты в общей для всех шине. Одной из самых крупных систем, основанных на UMA, является SGI Challenge, поддерживающая до 36 процессоров на одной шине.

В системах NUMA (рис. 7.4, *б*) каждый процессор имеет некоторый объем локальной памяти и обладает возможностью доступа к памяти других CPU. Скорость доступа к удаленной памяти ниже, чем локальный обмен. Также существуют гибридные системы (рис. 7.4, *в*), в которых группы процессоров разделяют между собой некоторый объем локальной памяти и при этом обладают возможностью доступа к памяти других групп CPU. Модель NUMA весьма сложна для практической реализации, так как программисту требуется немало усилий для того, чтобы скрыть подробности аппаратной архитектуры от пользовательских приложений.

В системах NORMA каждый процессор обладает прямым доступом только к локальной для него области памяти. Доступ к удаленной памяти осуществляется посредством передачи сообщений. Аппаратура должна поддерживать высокую скорость взаимодействия, обладать большой пропускной способностью, достаточной для осуществления удаленного доступа к памяти. При построении систем на основе архитектуры NORMA необходимо реализовать в ОС управление кэш-памятью, поддержку модели на уровне планировщика, а также создать компиляторы, умеющие оптимизировать коды программ конкретно для данной архитектуры.

В этой главе мы будем рассказывать только о системах, основанных на модели UMA.

7.3.2. Поддержка синхронизации

Синхронизация в многопроцессорных системах сильно зависит от ее аппаратной поддержки. Представим, как проходит простейшая операция блокировки ресурса при помощи флага locked, обрабатываемого в области разделяемой памяти. Процедура может состоять из указанной ниже последовательности операций.

1. Чтение флага.
2. Если значение флага равно нулю (следовательно, ресурс свободен), то блокируем ресурс путем установки флага в единицу.

[1] При этом кэширование данных, инструкций и преобразования адресов происходит в каждом процессоре отдельно.

3. Возвращаем TRUE в случае удачной блокировки ресурса или FALSE в противоположном случае.

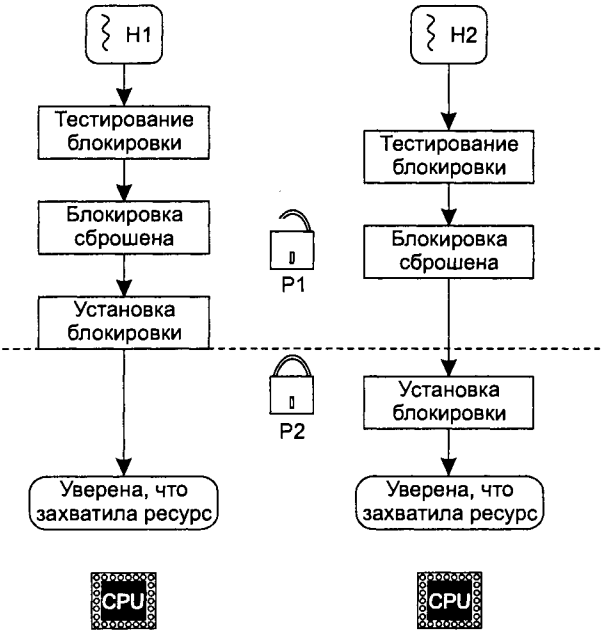


Рис. 7.5. Возникновение состязательности при невыполнении условия неделимости операции тестирования и установки

В многопроцессорной системе две нити, выполняющиеся на двух различных процессорах, могут производить эти операции одновременно. При этом обе нити будут считать, что обладают эксклюзивным правом на ресурс (как это показано на рис. 7.5). Для предупреждения возникновения подобных ситуаций аппаратура компьютера должна предлагать более мощную конструкцию, объединяющую все три операции. Существуют две реализации этой задачи, основанные на применении инструкции тестирования и установки, либо хранения по условию.

Неделимая команда тестирования и установки

Неделимая операция *проверки и установки* (*test-and-set*) обычно используется для действий над одним битом. Команда проверяет его значение, устанавливает в единицу и возвращает предыдущее значение. Таким образом, после завершения ее работы бит равняется 1 (ресурс занят), а возвращаемая величина показывает предыдущее значение этого бита. Если две нити, выполняющиеся на двух различных процессорах, попытаются одновременно произвести действия над одним и тем же битом, то свойство неделимости операции даст возможность сделать это только последовательно, друг за другом. Эта

действие также неделимо с точки зрения прерываний. Если оно возникнет во время выполнения команды, то будет обработано только после ее завершения.

Операция тестирования и установки идеально подходит для простых случаев блокировки ресурсов. Если команда возвращает единицу, вызвавшая ее нить занимает ресурс. Если возвращает ноль, то требуемый ресурс уже занят другой нитью. Освобождение ресурса производится путем сброса бита в 0. Примерами практической реализации операции тестирования и установки являются BBSSI (Branch on Bit Set and Set Interlocked) на машине VAX-11 [5] и LDSTUB (LoaD and STore Unsigned Byte) на SPARC.

Инструкции взаимосвязи с загрузкой и хранения по условию

В некоторых процессорах применяется пара специальных инструкций загрузки и хранения, используемых для неделимых операций чтения, изменения и записи. Команда *взаимосвязи с загрузкой* (*load-linked*, иногда называемая *load-locked*) загружает значение из памяти в регистр и устанавливает флаг, заставляющий аппаратуру проверять это значение. Если какой-либо из процессов запишет данные в эту область памяти, аппаратное обеспечение сбросит флаг. Инструкция *хранения по условию* (store-conditional) сохраняет новую величину переменной в область памяти, представленную установленным флагом. Кроме этого, команда установит значение еще одного регистра, указывающего на то, что была проведена операция сохранения.

7.3.3. Программная архитектура

С точки зрения программного обеспечения существуют три типа многопроцессорных систем: основанные на связке ведущий-ведомый, несимметричные и симметричные. Модель типа *ведущий-ведомый* (master-slave) является несимметричной [8], так как один из процессоров в системе играет роль *ведущего* (master), в то время как остальные являются *подчиненными* (slave). В таких системах возможна ситуация, при которой только ведущий процессор обладает правом обработки ввода-вывода и получения прерываний от различных устройств. В некоторых случаях коды ядра выполняются только на ведущем процессоре, при этом остальные CPU могут быть заняты исключительно пользовательскими программами. При соблюдении таких условий упрощается разработка систем, но преимущества многопроцессорных архитектур используются далеко не в полной мере. Результаты тестов показали [1], что в системах UNIX тратится более 40% общего процессорного времени на выполнение в режиме ядра. Следовательно, наиболее приемлемым вариантом является распределение подпрограмм ядра между несколькими процессорами.

В функционально несимметричных многопроцессорных системах различные подсистемы выполняются на отдельных процессорах. Например, на одном из процессоров могут работать сетевые процедуры, в том время как другой

занят обработкой ввода-вывода. Такой вариант более подходит для специализированных систем, чем для систем общего применения, к которым относится UNIX. Одной из удачных реализаций модели является файловый сервер Auspex NS5000 [10].

Симметричная многопроцессорная обработка (symmetric multiprocessing, SMP) является наиболее популярной моделью выполнения. В системах, основанных на этой архитектуре, все процессоры равнозначны и разделяют между собой единственную копию кодов и данных ядра, а также обладают доступом к различным системным ресурсам, таким как память или периферийные устройства. В режиме ядра может функционировать каждый процессор. Пользовательский процесс может быть направлен диспетчером на выполнение на любой CPU системы. В этой главе мы будем описывать только системы, основанные на архитектуре SMP (если не указано иное).

Оставшаяся часть главы посвящена современным механизмам синхронизации, применяемым в однопроцессорных и многопроцессорных системах.

7.4. Особенности синхронизации в многопроцессорных системах

Одним из основных условий, на которых построена традиционная модель синхронизации, является непрерываемое использование ядра нитью до тех пор, пока та не выйдет из привилегированного режима самостоятельно или не будет заблокирована в ожидании ресурса (кроме случаев возникновения прерываний). Такое свойство однопроцессорных ОС совершенно не подходит для многопроцессорных систем, в которых коды ядра могут выполняться одновременно на нескольких CPU. Данные, не требовавшие защиты от повреждения при использовании единственного процессора, теперь необходимо защищать. Рассмотрим проблему на примере организации доступа к таблице ресурсов IPC (см. раздел 6.3.1). Таблица ресурсов недоступна со стороны обработчиков прерываний и не поддерживает операции, которые могут повлечь за собой блокировку процесса. Следовательно, в однопроцессорных системах ядро может производить действия с этой структурой данных без какой-либо ее защиты. При применении многопроцессорных архитектур две нити, выполняющиеся на двух различных CPU, могут одновременно запросить доступ к таблице. В последнем случае перед использованием таблицы нити необходимо заблокировать ее тем или иным образом, запретив к ней доступ другим нитям на время проведения манипуляций.

Необходимо также пересмотреть основные элементы, использующиеся при осуществлении блокировки ресурсов. В традиционных системах ядро просто проверяет флаг locked и устанавливает его для блокировки объекта. В многопроцессорных архитектурах может случиться так, что один и тот же флаг

locked начнет проверяться двумя нитями одновременно. В этом случае обе нити обнаружат флаг сброшенным и сделают вывод о том, что ресурс свободен. Затем нити установят флаг и начнут использовать объект, что может привести к совершенно неожиданным результатам. Следовательно, многопроцессорные системы должны поддерживать неделимые операции проверки и установки флагов, гарантирующие занятие ресурса одновременно только одной нитью.

Еще одним примером необходимости изменений является приостановка обработки прерываний. В многопроцессорных системах нить обычно имеет право блокировать прерывания, относящиеся только к тому процессору, на котором она выполняется. Чаще всего не существует возможности приостановки прерываний на всех CPU системы, так как некоторые процессоры могут уже получить прерывания, приводящие к возникновению конфликта. Существует возможность повреждения структур данных нити, выполняющейся на одном процессоре, в том случае, если обработчик стартует на другом CPU. Еще одной сопутствующей проблемой является невозможность использования обработчиком модели синхронизации, основанной на приостановке и возобновлении выполнения, так как большинство реализаций ОС не позволяет обработчикам производить блокировку нитей. Система должна поддерживать некий механизм, позволяющий приостанавливать обработку прерываний, возникающих на других процессорах. Одним из возможных решений проблемы является использование глобальных уровней ipl, поддерживаемых на программном уровне.

7.4.1. Проблема выхода из режима ожидания

В многопроцессорных системах механизм ожидания-возобновления выполнения работает некорректно. Пример возникновения состязательности при применении этой технологии показан на рис. 7.6. Нить H1 заблокировала ресурс P1. Нить H2, выполняющаяся на другом процессоре, пытается получить тот же ресурс, но обнаруживает, что он уже занят. Тогда нить H2 вызывает функцию sleep(), чтобы перейти в режим ожидания ресурса P1. Между операциями проверки занятости ресурса и вызовом sleep() происходит освобождение P1, при этом H1 инициирует пробуждение всех нитей, находящихся в режиме ожидания этого ресурса. Так как нить H2 еще не успела переместиться в очередь ожидания, она не может получить сигнал о необходимости возобновления работы. В результате возникает ситуация, когда ресурс уже свободен, но при этом нить H2 продолжает ожидать его разблокирования. Если в дальнейшем ни одна нить не запросит ресурс P1, то H2 будет оставаться в режиме ожидания постоянно. Описанная ситуация получила название проблемы потери сигнала выхода из режима ожидания. Для ее решения необходимо объединить процедуру проверки ресурса на занятость и последующий вызов sleep() в единую, неделимую операцию.

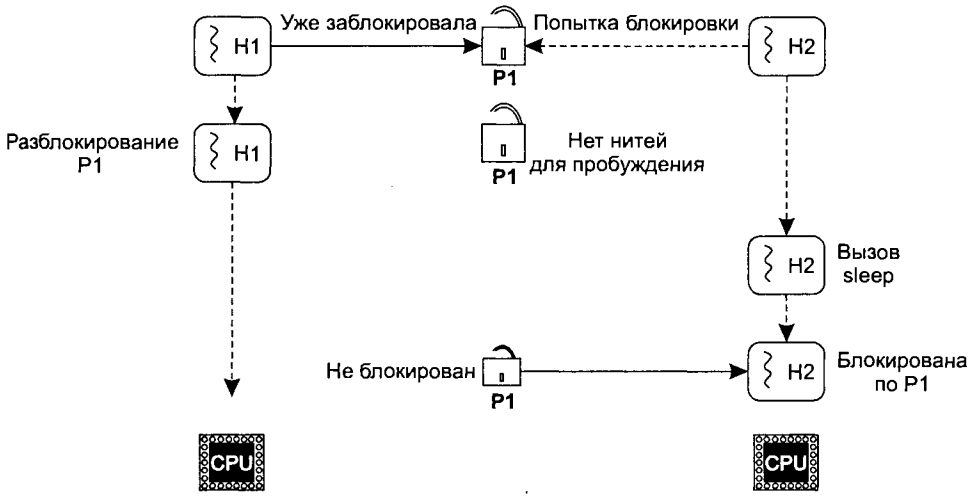


Рис. 7.6. Проблема потери сигнала выхода из режима ожидания

Приведенный пример показывает необходимость привлечения совершенно иных механизмов работы с ресурсами для многопроцессорных систем. Если при их разработке учесть и другие проблемы традиционных реализаций UNIX, то результатом станет создание наилучших решений, что повлияет и на производительность систем.

7.4.2. Проблема быстрого роста

При освобождении ресурса нитью происходит пробуждение всех остальных нитей, ожидавших объект. При этом только одна из них может занять ресурс, в то время как остальные нити обнаружат его заблокированным и снова перейдут в режим ожидания. Такой подход приводит к дополнительной загрузке системы, тратящей определенное время на возобновление работы нитей и контекстные переключения.

Описанная проблема не фатальна для однопроцессорных систем, поскольку в один момент времени текущим является только одна нить, которая может освободить ресурс. Однако в многопроцессорных архитектурах вероятно возникновение ситуации, когда несколько нитей, ожидавших освобождения ресурса, могут быть направлены планировщиком на выполнение одновременно на различных процессорах, что приведет к новой попытке получения этого ресурса всеми нитями. Такая ситуация получила название проблемы быстрого роста.

В случае если только одна нить оказалась заблокированной в ожидании ресурса, все равно между операциями восстановления работоспособного состояния нити и началом ее выполнения происходит определенная задержка. За этот промежуток времени ресурс может успеть занять совершенно другая

нить, что приведет к повторной блокировке первой нити. Если такая ситуация будет возникать часто, то она способна повлечь устаревание нити.

В текущих разделах вы увидели краткое описание проблем традиционной модели синхронизации, влияющих на корректную работу системы и ее производительность. Оставшаяся часть главы посвящена описанию механизмов синхронизации, подходящих как для однопроцессорных, так и для многопроцессорных систем.

7.5. Семафоры

Синхронизация в ранних реализациях системы UNIX для многопроцессорных машин строилась в основном на использовании *семафоров Дейкстры* (Dijkstra's semaphores) [6]. Их иногда также называют *семафорами со счетчиком* (counted semaphores). Семафоры — это переменные целого типа, поддерживающие две основные операции, P() и V(). Операция P() декрементирует (уменьшает на единицу) значение семафора и блокирует процесс, если результат меньше нуля. Операция V() инкрементирует переменную семафора. Если результат меньше или равен нулю, то она разбудит процесс, заблокированный по нему (если таковой существует). Листинг 7.1 показывает пример использования этих операций, а также функцию initsem(), применяемую для инициализации семафора, и функцию CP(), являющуюся версией P() и производящую блокировку.

Листинг 7.1. Операции с семафорами

```
void initsem(semaphore *sem, int val);
{
   *sem = val;
}

void P(semaphore sem) /* запрос семафора */
{
   *sem -= 1;
   while (*sem<0)
      sleep;
}
void V(semaphore *sem) /* освобождение семафора */
{
   *sem += 1;
   if (*sem <= 0)
      будим процесс, заблокированный по sem;
}
boolean_t CP(semaphore *sem) /* попытка получить семафор без блокировки */
{
```

```
   if (*sem > 0) {
      *sem -= 1;
      return TRUE;
} else
      return FALSE;
{
```

Неделимость операций над семафорами обеспечивается ядром, даже в случае использования многопроцессорных систем. Таким образом, если две нити одновременно попытаются произвести какие-либо операции над одним и тем же семафором, то действия второй начнутся только после того, как произойдет завершение или блокировка процедур первой нити. Операции P() и V() аналогичны sleep и wakeup, но имеют отличающуюся от них семантику. Команда CP() позволяет запрашивать семафоры без блокировки и может быть использована обработчиками прерываний или другими функциями, для которых блокировка нежелательна. Операцию CP() также можно использовать для предупреждении взаимоблокировки в тех случаях, при которых использование стандартной операции P() может привести к их возникновению.

7.5.1. Семафоры как средство взаимного исключения

Пример, приведенный в листинге 7.2, показывает, как можно использовать семафоры для взаимного исключения по ресурсу. Семафор можно ассоциировать с совместно используемыми ресурсами, например взаимосвязанным списком, и присвоить ему значение 1 при инициализации. Для блокировки ресурса нить выполняет операцию P(), для его освобождения — V(). Первое применение P() установит значение семафора в ноль, следовательно, все последующие вызовы этой операции приведут к блокировке. При вызове V() значение семафора будет инкрементировано, следовательно, одна из заблокированных ранее нитей будет разбужена.

Листинг 7.2. Применение семафоров для эксклюзивного использования ресурса

```
/* при инициализации */
semaphore sem;
initsem (&sem, 1),

/* при каждом применении семафора */
P(&sem);
...
использование ресурса
...
V(&sem);
```

7.5.2. Семафоры и ожидание наступления событий

Листинг 7.3 демонстрирует пример использования семафоров для организации ожидания события. Для этого семафор должен быть установлен при инициализации в значение «ноль». Тогда нить, производящая операцию P(), будет заблокирована. При наступлении события каждая приостановленная нить должна выполнить V(). Это можно реализовать при помощи вызова V(), производимого единожды после возникновения ожидаемого события, и применения той же операции V() каждой нитью после пробуждения.

Листинг 7.3. Применение семафоров для организации ожидания наступления событий

```
/* при инициализации */
semaphore event;
initsem (&event, 0); /* инициализацию можно произвести при загрузке */

/* выполняется нитью, которой необходимо ждать наступления события */
P(&event); /* блокировка в случае, если событие еще не произошло */
/* после наступления события */
V(&event); /* дает возможность другой нити возобновить выполнение */
/* продолжение кода нити */

/* выполняется при наступлении события */
V(&event); /* будит одну нить */
```

7.5.3. Семафоры и управление исчисляемыми ресурсами

Семафоры можно использовать для размещения различных исчисляемых ресурсов, таких как заголовки блоков сообщений в библиотеке STREAMS. Как показано на примере в листинге 7.4, для этого семафор при инициализации получает значение, равное допустимому количеству экземпляров ресурса. При его запросе нить вызывает операцию P(), при освобождении — V(). Таким образом, значение семафора показывает текущее количество доступных экземпляров ресурса. Если значение отрицательно, то абсолютное значение семафора равно количеству ожидающих запросов ресурса (или блокированных нитей). Представленный алгоритм является решением проблемы взаимосвязи производителей-потребителей ресурсов.

Листинг 7.4. Применение семафоров для подсчета доступного количества экземпляров ресурса

```
/* при инициализации */
semaphore counter;
initsem(&counter, resourseCount);
```

```
/* выполняется на стадии использования ресурса */
P(&counter). /* блокировка нити, пока ресурс не станет доступен */
использование ресурса; /* ресурс доступен в текущий момент */
V(&counter); /* освобождение ресурса */
```

7.5.4. Недостатки семафоров

Семафоры представляют собой достаточно гибкие, расширяемые компоненты, при помощи которых можно решать различные проблемы синхронизации, но они обладают несколькими недостатками, не позволяющими применять их в некоторых ситуациях. Во-первых, семафоры являются компонентами высокого уровня, основанными на элементах более низкого уровня, обладающих свойством неделимости и механизмами блокировки. Для того чтобы сохранить неделимость операций P() и V() на многопроцессорных системах, необходимо гарантировать их выполнение на низком уровне, обладая при этом эксклюзивным доступом к объекту семафора. Блокировка и возобновление работы требуют проведения контекстных переключений и манипуляций с очередями сна и планирования, что делает выполнение этих операций очень медленным. Такая скорость может быть приемлема для ресурсов, удерживаемых на большие промежутки времени, однако совершенно не подходит для объектов, требуемых на малое время.

Семафоры не предоставляют информации о том, какая конкретная нить блокируется вследствие проведения операции P(). Чаще всего это не важно, но в некоторых случаях отсутствие информации о блокируемой нити является критичным. Например, буфер кэша в системе UNIX использует функцию getblk() для просмотра конкретного дискового блока, размещенного в кэше. Если необходимый блок обнаружен, вызов getblk() попытается заблокировать его путем проведения операции P(). Если выполнение P() приведет к переходу нити в режим ожидания (так как блок окажется уже занятым), то при возобновлении ее работы может возникнуть ситуация, когда требуемый буфер содержит уже совершенно другой блок. За время ожидания необходимый блок может оказаться уже в совершенно ином месте буфера. Таким образом, после окончания работы P() вполне вероятно, что нить заняла совершенно другой буфер. Описанная проблема может быть решена и при помощи семафоров, но такое решение является весьма неэффективным и громоздким, следовательно, более разумным является применение в этом случае других элементов и конструкций системы [15].

7.5.5. Конвои

Если сравнить семафоры с традиционными механизмами ОС, основанными на приостановке-возобновлении выполнения, то станет видно, что первые обладают определенным преимуществом, связанным с возобновлением выполнения

дополнительных процессов. Если процесс был разбужен в результате операции P(), то он гарантированно получит необходимый ресурс. Семантика вызова гарантирует передачу пробуждаемой нити прав владения семафором до начала его выполнения. Если в этот промежуток времени иная нить попытается запросить тот же семафор, то она потерпит неудачу. Однако такое свойство семафоров приводит к возникновению проблемы, названной *конвоированием* семафоров [12]. Конвой возникает в случае частого обращения к семафору. Это может уменьшить производительность любого механизма блокировки, но особенность семантики семафоров значительно усложняет возникающую проблему.

На рис. 7.7 показан пример формирования конвоирования. P1 — это критический участок кода, защищенный семафором. На стадии *а* семафор удерживается нитью H2, а нить H3 находится в режиме его ожидания. Нить H1 выполняется на другом процессоре системы, в то время как H4 находится в очереди планирования. Теперь представьте, что H2 завершает выполнение критической секции и освобождает семафор. Тогда она разбудит нить H3 и перенесет ее в очередь планировщика. Теперь семафор удерживает нить H3 (как это показано на рис. 7.7, *б*).

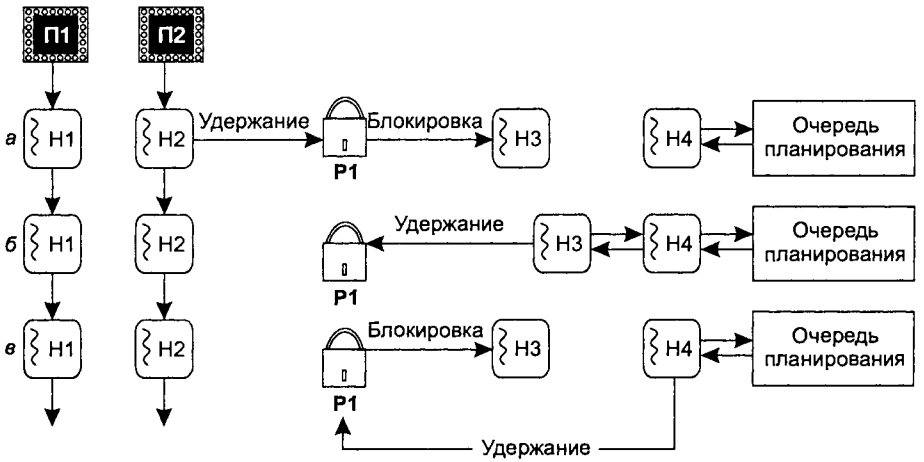


Рис. 7.7. Пример формирования конвоев

Затем выполнение критического участка кода необходимо начать нити H1. Так как семафор в текущий момент удерживается нитью H3, H1 будет заблокирована и освободит процессор П1. Система направит нить H4 на выполнение на этом процессоре. Таким образом, на этапе *в* семафор удерживает нить H3, H1 ожидает его освобождения, а нити H2 и H4 могут выполняться до тех пор, пока сами не освободят занимаемые ими процессоры.

Проблема возникает именно на этой стадии. Семафор был присвоен нити H3, но она в данный момент не выполняется и, следовательно, не находится в критической области. В результате нить H1 должна быть заблокирована в ожидании семафора, несмотря на то, что ни одна из нитей не выполняет

критичный участок кода. Семантика команд работы с семафорами предоставляет ресурсы по принципу: «кто первым пришел, тот и будет обслужен». Такой подход приводит к большому количеству ненужных переключений контекста. Представьте ситуацию, в которой вместо семафора используется *объект mutex* (взаимное исключение). Тогда на стадии *б* нить H2 должна освободить объект и разбудить H3. Но последняя не является на этой стадии владельцем объекта. Следовательно, на следующем этапе объект будет пытаться получить нить H1, что исключит переключение контекста.

ПРИМЕЧАНИЕ

Mutex, или взаимное исключение (сокращение от mutual exclusion lock), является общим термином, обозначающим любой примитив, приводящий к семантике эксклюзивного доступа.[1]

Из сказанного можно сделать вывод, что наиболее предпочтительным является применение вместо единого монолитного объекта высшего уровня простых в обращении элементов низшего уровня. Именно этой идеей руководствуются разработчики ядра современных многопроцессорных систем. Следующие разделы будут посвящены описанию механизмов низшего уровня, создающих вместе гибкую систему синхронизации.

7.6. Простая блокировка

Простейшим элементом блокировки является объект, называемый *циклической блокировкой* (spin lock), а иногда — *простой блокировкой* (simple lock) или *простым взаимным исключением* (simple mutex). Если ресурс защищен

[1] Согласно POSIX, как семафоры, так и взаимные исключения являются объектами синхронизации, но первые представляют собой *минимальный примитив*, на котором основываются комплексные механизмы синхронизации, а вторые — не что иное как *объекты синхронизации*, используемые для обеспечения *последовательного доступа множества нитей к разделяемым данным*. Последнее определение не отличается в своей основе от такового для циклической (spin) блокировки. Следуя логике примечания и примеру в 7.5.1, семафоры также можно считать взаимными исключениями. Отсюда нетрудно сделать вывод, что семафоры и простые блокировки можно причислить к объектам mutex. И наоборот, применяемые в иных ОС (VxWorks) целочисленные семафоры (со счетчиком) нельзя назвать семафорами, если ориентироваться на POSIX (следование которому — дело добровольное). И уж тем более такими нельзя назвать multiple wait (muxwait) семафоры OS/2, аналогичные наборам семафоров в UNIX, используемые для управления именованными каналами. В этой же ОС имеется понятие семафора взаимного исключения (все тот же mutex). Или посмотрите далее в начале раздела 7.6... Таким образом, реализация объекта, то есть семантика обращений к нему, и является главным его идентификатором. Например, если в System V для управления несколькими семафорами требуется один вызов, а не несколько, то это — один объект. В этом отношении POSIX недостаточно учитывает ситуацию на местах, что логично, поскольку «авторы стандартов не пишут компьютерных программ». Но с другой стороны, программисты часто, пытаясь «договориться между собой», используют жаргонные термины, например «мьютекс», поэтому дальше мы будем придерживаться более грамотного «взаимного исключения». — *Прим. ред.*

при помощи этого механизма, то нить, пытающаяся получить доступ к такому ресурсу, перейдет в режим занятого ожидания (выполнения определенного цикла) до тех пор, пока объект не будет освобожден. Взаимоблокировка обычно представляет собой переменную, имеющую значение 0, если ресурс доступен, и значение 1, если он занят. Отслеживание переменной происходит внутри цикла, основанного на одной из неделимых операций, поддерживаемых конкретной аппаратной архитектурой, например при помощи команды тестирования и установки. Ниже представлен пример использования взаимоблокировки. Он основан на том, что операция test_and_set() возвращает предыдущее значение объекта.

Листинг 7.5. Применение циклической блокировки

```
void spin_lock (spinlock_t *s) {
   while (test_and_set(s)!= 0) /* ресурс занят */
      ; /* выполнение цикла до тех пор, пока ресурс не освободится */
}
void spin_unlock (spinlock_t s) {s = 0; }
```

Несмотря на простоту алгоритма, в нем имеются определенные недостатки. На многих машинах операция test_and_set() блокирует шину памяти. Таким образом, длительный цикл может привести к занятию шины одной нитью – и, следовательно, к существенному уменьшению производительности системы. Решить проблему можно при использовании двух циклов: внешний цикл предназначен для проверки переменной, если проверка не проходит успешно, внутренний цикл будет ждать обнуления значения. Простейшая проверка условия во внутреннем цикле не требует занятия шины памяти. Реализация взаимоблокировки при помощи двух циклов показана в листинге 7.6.

Листинг 7.6. Более корректный способ применения простой блокировки

```
void spin_lock (spinlock_t *s)
{
   while (test_and_set(s)!= 0) /* ресурс занят */
      while (*s != 0)
      ; /* ожидание освобождения ресурса */
}
void spin_unlock (spinlock_t s) {s = 0; }
```

7.6.1. Применение простой блокировки

Наиболее важным свойством простой блокировки является продолжение выполнения нити (или использования процессорного времени) во время ожидания освобождения ресурса. Следовательно, такую блокировку можно применять только для коротких промежутков ожидания. В частности, эти операции не следует производить совместно с блокировкой нитей. Рекомендуется также приостановить обработку прерываний до вызова объекта простой блокировки, что гарантирует уменьшение общей продолжительности ожидания.

Главным условием применения простой блокировки является использование ресурса другой нитью (выполняющейся на другом процессоре), в то время как первая нить находится в режиме ожидания ресурса. Следовательно, использование таких объектов возможно только в многопроцессорных архитектурах. На однопроцессорной машине нить, находящаяся в цикле ожидания освобождения ресурса, не выйдет из этого цикла никогда. Однако многопроцессорные алгоритмы должны работать корректно при любом количестве процессоров в системе, в том числе и в случае одного CPU. Для этого необходимо ограничить применение правила занятия процессора нитью при ожидании ресурса. В частности, на однопроцессорных машинах ни одна нить не должна иметь право входить в режим ждущего цикла при применении взаимоисключения.

Основным достоинством объектов простой блокировки является невысокое использование ресурсов системы. Блок не содержит никакой информации внутри себя, а операции занятия и освобождения объекта требуют выполнения всего лишь одной команды для каждой из них. Простая блокировка идеально подходит для блокировки структур данных, к которым производятся частые обращения, например для удаления элемента из двунаправленного связанного списка или проведения операции загрузки, изменения и сохранения переменной. Из сказанного можно сделать вывод, что применение объектов простой блокировки наиболее целесообразно для защиты тех структур данных, которые не нуждались в таковой в однопроцессорных системах. Однако простые объекты блокировки применяются и для более сложной защиты, что будет продемонстрировано в следующих разделах главы. Например, они могут быть использованы совместно с семафорами для подтверждения неделимости операции (листинг 7.7.).

Листинг 7.7. Применение простых блокировок для организации доступа к двунаправленному связанному списку

```
spinlock_t list;
spin_lock (&list);
item->forw->back = item->back;
item->back->forw = item->forw;
spin_unlock (&list);
```

7.7. Условные переменные

Условные переменные представляют собой более сложный механизм блокировки, основанный на логике *предикатов* (выражения, имеющие значения TRUE и FALSE). Такие переменные позволяют нити приостановить работу в ожидании совместно используемых ресурсов и возобновить выполнение только одной или сразу всех ожидающих нитей при изменении значения выражения. Использование условных переменных является более приемлемым, чем применение простых блокировок.

Представьте ситуацию, в которой одна или несколько нитей серверного приложения находятся в ожидании запросов клиентов. Поступающие запросы передаются ожидающим их нитям. Если ни одна из нитей не может в данный момент обработать поступивший запрос, то последний будет поставлен в очередь. Когда очередная нить серверного приложения заканчивает работу с одним запросом и становится готовой обрабатывать новый, то сначала она проверяет очередь. Если в ней будет обнаружено сообщение, нить удалит его из очереди и начнет обработку запроса. Если очередь окажется пустой, нить приостанавливает выполнение в ожидании поступления нового запроса. Обработка сообщений может быть реализована при помощи условных переменных, ассоциируемых с очередью запросов. В этом случае совместно используемыми данными является очередь сама по себе, а проверяемым условием — выражение «очередь не пуста».

Механизм условных переменных схож с каналами сна, при использовании которых нити серверного приложения блокируются по условию и возобновляют работу после получения сообщений. На многопроцессорных системах необходимо защищать эти механизмы от состязательности, которая может привести к возникновению таких проблем, как потеря сигнала выхода из режима ожидания. В качестве примера можно представить ситуацию, при которой сообщение приходит уже после проведения проверки очереди, но до того момента, как нить перешла в режим ожидания. В таком случае нить окажется приостановленной даже при наличии необработанных сообщений. Следовательно, для защиты от подобных проблем необходимо использовать неделимую операцию проверки условия и блокировки нити.

Такая возможность достигается при помощи дополнительных элементов блокировки. Объекты блокировки (обычно это простые блокировки или *spin locks*) защищают совместно используемые данные и предохраняют их от возникновения ситуации «невыхода» из режима ожидания. Для этого нить серверного приложения создает объект блокировки над очередью сообщений и проверяет очередь на отсутствие запросов. Если очередь окажется пустой, нить вызовет функцию wait() с условием, по которому происходит удержание блокировки. Объект блокировки является входным аргументом функции wait(). Функция производит приостановление выполнения нити и освобождение объекта блокировки. После получения сообщения и возобновления выполнения нити вызов wait() снова запрашивает объект блокировки, после чего ее работа завершается. Пример использования условных переменных показан в листинге 7.8.

Листинг 7.8. Применение условных переменных

```
struct condition {
   proc *next; /* двунаправленный связанный список */
   proc *prev; /* приостановленных нитей */
   spinlock_t lostLock; /* используется для защиты списка */
};
```

```
void wait (condition *c, spinlock_t *s)
{
   spin_lock(&c->listLock);
   add self to the linked list;
   spin_unlock(&c->listLock);
   spin_unlock(s); /* освобождение объекта блокировки до приостановки
      выполнения нити */
   swtch(); /* переключение контекста */
   spin_lock(s); /* повторный запрос объекта блокировки */
   return;
{
void do_signal(condition *c)
/* пробуждение одной нити, находящейся в ожидании по этому условию */
{
   spin_lock(&c->listLock);
   удаление одной нити из связанного списка, если тот не пуст,
   spin_unlock(&c->listLock);
   ...
   если нить удалена из списка, делаем ее работоспособной;
   ...
   return;
{
void do_broadcast(condition *c)
/* пробуждение всех нитей, находящихся в режиме ожидания по заданному
условию */
{
   spin_lock(&c->listLock);
   while (связанный список не пуст) {
      ...
      удаление нити из связанного списка и
      восстановление работоспособности нити
      ...
   }
spin_unlock(&c->listLock);
}
```

7.7.1. Некоторые детали реализации условных переменных

При рассмотрении механизма условных переменных необходимо сделать несколько важных замечаний. Логическое условие само по себе не является частью переменной, таким образом, оно должно проверяться до вызова wait(). Более того, необходимо помнить о том, что при реализации механизма применяются сразу два объекта блокировки. Одним из этих объектов является listLock, который используется для защиты списка нитей, находящихся в ожидании изменения условия. Второй объект защищает проверяемые данные. Он не

является частью условной переменной, но передается в качестве аргумента функции wait(). Вызов swtch() и участок кода, меняющий режим ожидающей нити на работоспособный, может использовать для защиты очередей планировщика еще один объект блокировки.

Таким образом, возникает ситуация, при которой нить, пытающаяся получить один объект блокировки, уже удерживает еще один объект такого же типа. Это не представляет опасности, так как объекты блокировки имеют всего лишь одно ограничение: они запрещают приостановку выполнения нити, удерживающей один из таких объектов. Защита от зависания гарантирована при строго определенном порядке следования объектов: блокировка по условию должна запрашиваться до запроса listLock.

Очередь ожидающих нитей не обязательно должна быть частью структуры условия. Вместо этого можно применять глобальный набор каналов сна, точно такой же, как в традиционных системах UNIX. В этом случае объект listLock в условии заменяется объектом блокировки, защищающим соответствующий канал сна. Оба приведенных метода обладают определенными достоинствами, о которых мы уже рассказывали ранее.

Одним из достоинств условных переменных является существование двух различных методов обработки события. После его возникновения можно возобновить работу либо одной нити (при помощи do_signal()), либо всех ожидающих нитей сразу (используя команду do_broadcast()). Каждый из этих вариантов может оказаться наиболее подходящим в той или иной ситуации. В случае применения механизма условных переменных серверными приложениями эффективнее всего возобновлять работу только одной нити, так как каждый запрос обычно обрабатывается единственной нитью. Однако существуют и иные ситуации, например, когда программа выполняется сразу же несколькими нитями, использующими совместно одну копию исходных кодов программы. Если код программы, не хранящийся постоянно в памяти, попытаются запросить сразу несколько нитей, результатом станет получение ошибки каждой из них. Первая нить инициирует доступ к странице, расположенной на диске. Остальные нити получают уведомление о начале операции чтения данных и приостановят выполнение в ожидании завершения ввода-вывода. Когда страница кода будет считана в память, наилучшим решением является вызов do_broadcast() и возобновление работы всех приостановленных нитей, после чего все они могут иметь доступ к этой странице без возникновения ошибок.

7.7.2. События

Чаще всего логическое условие переменной является простым. Обычно нить ожидает завершения выполнения определенной задачи. Момент завершения легко обозначить при помощи установки глобального флага. Ситуацию можно описать более приемлемым способом при помощи элемента высшего уров-

ня, называемого *событием* (event). Событие включает в себя флаг done, объект блокировки, защищающий этот флаг, и условную переменную на единый объект. Событием легко манипулировать при помощи двух простых операций — awaitDone() и setDone(). Команда awaitDone() приведет к блокировке, установленной до тех пор, пока не произойдет определенное событие, а операция setDone() помечает событие event как уже свершившееся и возобновляет выполнение всех нитей, заблокированных в ожидании этого события. Помимо этих команд интерфейс взаимодействия с объектами-событиями может поддерживать неблокирующие функции testDone() и reset(), которые снова помечают событие event как еще не свершившееся. В некоторых случаях флаг done можно заменить переменной, что позволит передавать более подробную информацию при наступлении события.

7.7.3. Блокирующие объекты

Часто возникают ситуации, при которых нити нужно удерживать ресурс на продолжительный период времени, в течение которого объект должен иметь возможность блокировки по другим событиям. При этом нить, которой необходим ресурс, не начинает выполнять цикл ожидания его освобождения, а приостанавливается. Для реализации такого подхода следует использовать элемент, называемый *блокирующим объектом синхронизации* (blocking lock). Такой объект поддерживает две основные операции, lock() и unlock(), а также дополнительную команду tryLock(). Элемент синхронизирует два объекта — флаг locked, относящийся к ресурсу, и очередь сна. Неделимость операций гарантирует дополнительное использование объектов простой блокировки. Такие объекты могут быть реализованы при помощи обычных условных переменных, где проверяемым выражением является сброс флага locked. С точки зрения производительности объекты блокирующей синхронизации лучше всего реализовывать как базовые элементы. В частности, если каждый ресурс обладает собственным каналом сна, то для защиты флага и канала можно применять единственный объект блокировки.

7.8. Синхронизация чтения-записи

Изменение ресурса требует установки эксклюзивного права на доступ к нему. Это означает, что его модификацию может производить в один момент времени только одна нить. Однако в большинстве случаев целесообразно разрешить сразу нескольким нитям читать совместно используемые данные в течение промежутка времени, пока никто не попытается произвести запись информации. Для реализации такого подхода требуется сложный механизм синхронизации, поддерживающий одновременно два типа доступа к ресурсу, эксклюзивный и совместный. Механизм может быть создан на основе простой

блокировки и условных переменных [2]. Однако перед тем как перейти к рассмотрению деталей реализации этого механизма, обсудим вкратце его семантику. Синхронизация чтения-записи может разрешать либо изменение ресурса (запись) одной нитью, либо его чтение несколькими нитями. Основными операциями над объектом являются lockShared(), lockExclusive(), unlockShared(), unlockExclusive(). Кроме этого, объекты можно попытаться сделать эксклюзивно используемыми или разделяемыми при помощи команд tryLockShared(), tryLockExclusive(), возвращающих значение FALSE вместо блокировки нити. Дополнительно необходима поддержка преобразования объекта из «эксклюзивного» в «совместный» и обратно при помощи команд upgrade() и downgrade(). Операция lockShared() должна приводить к блокировке, если существует объект эксклюзивной синхронизации, в то время как lockExclusive() блокирует в случае любого занятия ресурса (как в эксклюзивном, так и в режиме совместного доступа).

7.8.1. Задачи, стоящие перед разработчиками

Что должна сделать нить после освобождения ресурса? В традиционных системах UNIX произойдет пробуждение всех нитей, ожидающих этот объект. Такой подход является абсолютно неэффективным. Если нить, производящая изменение информации, снова потребует объект блокировки, остальные читающие и модифицирующие ресурс нити должны будут перейти в режим ожидания повторно. Если объект затребует нить-получатель данных, то все модифицирующие ресурс нити обязаны будут приостановить выполнение. Таким образом, становится очевидной необходимость использовать некий протокол, защищающий от излишних переключений нитей из режима ожидания.

Когда ресурс освобождает нить-получатель, то она не производит никаких дополнительных действий, если существуют другие нити, считывающие данные из этого ресурса. После того как последняя читающая нить освобождает ресурс, она должна разбудить единственную нить, имеющую возможность записи информации.

Если ресурс освобождает пишущая нить, ей необходимо производить выбор между восстановлением работоспособности другой модифицирующей нити либо других читателей (представим, что объект ожидают и считывающие, и записывающие нити). Если отдать предпочтение нитям, изменяющим информацию, то читающие нити могут быть подвержены устареванию. Наиболее приемлемым решением проблемы действий при освобождении объекта эксклюзивной синхронизации является возобновление работоспособности всех ожидающих читающих нитей. Если таковых не существует, то разбужена будет нить, осуществляющая запись данных.

Такая методика действий может привести к устареванию нитей, осуществляющих запись. Если объект необходимо читать большому количеству нитей, ресурс будет постоянно блокирован на чтение; следовательно, в этом случае

пишущая нить не сможет получить доступ к нему. Для защиты от возникновения подобной ситуации запрос lockShared() должен осуществлять блокировку при существовании ожидающих записи нитей даже в том случае, если ресурс защищен только для чтения. Такое решение при определенных условиях может позволить чередовать доступ между несколькими пишущими нитями и большими множествами нитей, осуществляющих только чтение ресурса.

Функция upgrade() должна быть защищена от взаимоблокировки. Такая ситуация может возникнуть, если в реализации механизма не предусмотрены определенные правила при изменении статуса запросов от ожидающих пишущих нитей. Если две нити пытаются изменить статус объекта синхронизации, то каждая из них окажется заблокированной, так как другая нить удерживает совместно используемый объект. Одним из методов защиты от возникновения указанной ситуации является освобождение функцией upgrade() разделяемого объекта перед блокировкой нити в том случае, если объект для эксклюзивного доступа окажется невозможно получить сразу же. Это приведет к дополнительным проблемам с точки зрения пользователя, так как во время выполнения upgrade() другая нить может произвести изменение объекта. Еще одним вариантом выхода из описанной ситуации является возврат функции upgrade() с ошибкой и освобождение разделяемого объекта синхронизации в том случае, если другая нить уже находится в ожидании изменения его статуса.

7.8.2. Реализация синхронизации чтения-записи

Пример реализации механизмов синхронизации чтения-записи показан в листинге 7.9.

Листинг 7.9. Реализация синхронизации чтения-записи

```
struct rwlock {
   int nActive; /* количество активных читателей, либо -1, если существует
      активная нить, модифицирующая ресурс */
   int nPendingReads;
   int nPendingWrites,
   spinlock_t sl;
   condition canRead,
   condition canWrite;
};
void lockShared(struct rwlock *r)
{
   spin_lock(&r->sl);
   r->nPendingReads++,
   if (r->nPendingsWrites > 0)
      wait (&r->canRead, &r->sl); /* защита от устаревания
         пишущих нитей */
```

продолжение

Листинг 7.9 (*продолжение*)

```
   while (r->nActive < 0) /* если кто-то удерживает эксклюзивно
         объект синхронизации */
      wait (&r->canRead, &r->sl);
   r->nActive++;
   r->nPendingReads--;
   spin_unlock(&r->sl);
}
void unlockShared(struct rwlock *r)
{
   spin_lock(&r->sl);
   r->nActive--;
   if (r->nActive == 0) { /* нет других читающих нитей */
      spin_unlock(&r->sl);
      do_signal(&r->canWrite);
   } else
      spin_unlock(&r->sl);
}
void lockExclusive(struct rwlock *r)
{
   spin_lock(&r->sl);
   r->nPendingWrites++;
   while (r->nActive)
      wait (&r->canWrite, &r->sl);
   r->nPendingWrites--;
   r->nActive = -1;
   spin_unlock(&r->sl);
}
void unlockExclusive(struct rwlock *r)
{
   boolean_t wakeReaders;
   spin_lock(&r->sl);
   r->nActive = 0;
   wakeReaders = (r->nPendingReads != 0);
   spin_unlock(&r->sl);
   if (wakeReaders)
      do_broadcast (&r->canRead); /* пробуждение всех нитей, ожидающих
         возможности чтения ресурса */
   else
      do_signal (&r->canWrite); /* пробуждение одной нити,
         осуществляющей запись данных */
}
void downgrade(struct rwlock *r)
{
   boolean_t wakeReaders;
   spin_lock(&r->sl);
   r->nActive = 1;
   wakeReaders= (r->nPendingReads != 0);
```

```
   spin_unlock(&r->sl);
   if (wakeReaders)
      do_broadcast (&r->canRead); /* пробуждение всех нитей, ожидающих
         возможности чтения ресурса */
}
void upgrade(struct rwlock *r)
{
   spin_lock(&r->sl);
   if (r->nActive == 1) { /* нет нитей, ожидающих ресурс на чтение */
      r->nActive = -1;
   } else {
      r->nPendingWrites++;
      r->nActive--; /* освобождение совместного объекта синхронизации */
      while (r->nActive)
         wait (&r->canWrite, &r->sl);
      r->nPendingWrites--;
      r->nActive = -1;
   }
   spin_unlock(&r->sl);
}
```

7.9. Счетчики ссылок

Объект блокировки в состоянии защитить данные, находящиеся внутри ресурса. Но существует немало ситуаций, при которых необходима защита ресурса самого по себе. Многие объекты ядра запрашиваются и освобождаются динамически. Если одна из нитей освободит такой объект, другие нити не будут знать об этом и могут попытаться получить доступ к ресурсу при помощи прямого указателя на него. Возможно, что ядро уже произвело передачу участка памяти, занятого объектом, другому ресурсу. Тогда попытка доступа к нему может привести к нарушению работы системы.

Если нить обладает указателем на объект, то такой указатель считается правильным на тот период времени, пока он не будет освобожден нитью. Ядро системы может гарантировать правильность указателя при помощи *счетчика ссылок* на каждый объект. Для этого ядро устанавливает значение счетчика равным единице при первом запросе объекта (создавая при этом указатель). Счетчик инкрементируется при создании каждого нового указателя на объект.

В таком случае при получении указателя на объект нить получает на него ссылку. В дальнейшем нить может освободить ссылку, которая ей уже не нужна, при этом ядро системы произведет декремент счетчика ссылок объекта. Если значение счетчика станет равным нулю, это будет означать, что ни одна нить не обладает ссылкой на объект; тогда ядро осуществит освобождение объекта.

Механизм счетчиков ссылок применяется при управлении файловой системой. Система поддерживает счетчик для объектов vnode, в которых хранит информацию об используемых файлах (см. раздел 8.7). Если пользователь открывает файл, ядро возвращает дескриптор этого файла, содержащий ссылку на vnode. Пользователь передает этот дескриптор системным вызовам read или write, предоставляя возможность быстрого доступа ядра к файлу без проведения преобразований имен файлов. Если пользователь закрывает файл, ссылка освобождается. Если один и тот же файл открывают сразу несколько пользователей, то они получают ссылки на один и тот же объект vnode. После того как последний пользователь закроет файл, ядро освобождает объект vnode.

Пример показывает, что механизм счетчиков ссылок больше подходит для однопроцессорных систем, так как он слишком прост для применения в многопроцессорных архитектурах, в которых может возникнуть ситуация, при которой одна из нитей пытается освободить объект, в то время как другая нить, выполняющаяся на другом CPU, продолжает пользоваться им. Для применения механизма в таких системах необходимо реализовать несколько другую методику подсчета ссылок.

7.10. Другие проблемы, возникающие при синхронизации

При разработке и практическом применении сложных механизмов синхронизации для многопроцессорных систем необходимо учитывать несколько важных факторов. Наиболее важные из них будут описаны в этом разделе.

7.10.1. Предупреждение возникновения взаимоблокировки

Во многих случаях нити необходимо удерживать сразу же несколько ресурсов. Например, в реализации условных переменных, описанных в разделе 7.7, применяется два объекта простой блокировки, один из которых защищает данные и логическое условие, в то время как второй используется для защиты списка нитей, находящихся в режиме ожидания изменения значения условия. В этом случае попытка получения сразу же нескольких объектов синхронизации приведет к взаимоблокировке, как это показано на рис. 7.8. Нить H1 удерживает ресурс P1 и пытается получить ресурс P2. В тот же момент времени нить H2 может являться владельцем P2 и добиваться доступа к ресурсу P1. В такой ситуации ни одна из нитей не достигнет своей цели. Обе нити зависнут.

Для предупреждения возникновения взаимоблокировки обычно применяется одна из двух методик, называемых *иерархической блокировкой* (hierarchical locking) и *вероятностной блокировкой* (stochastic locking). Иерархический метод основан на назначении определенного порядка зависимых друг от друга

объектов синхронизации и требует, чтобы все нити получали такие объекты в указанном порядке. При этом в случае применения, к примеру, условных переменных нить должна занять логическое условие до блокировки связанного списка. Точное соблюдение порядка следования приводит к невозможности возникновения взаимоблокировки.

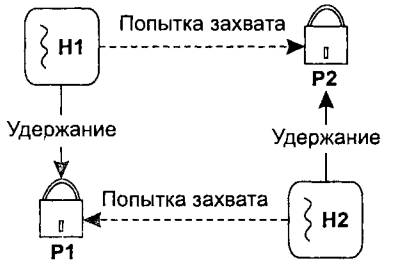


Рис. 7.8. Возникновение взаимной блокировки при применении простых объектов синхронизации

Иногда возникают ситуации, при которых порядок блокировки требуется изменить. Представьте буферный кэш, который обслуживается при помощи двунаправленного связанного списка дисковых буферов. Список отсортирован в порядке *«наиболее давно использовавшихся»* (least recently used, *LRU*) элементов. Все буферы, не используемые в текущий момент времени, занесены в этот список. Защита заголовка очереди, а также указателей на предыдущий и последующий элементы поддерживается при помощи единственного объекта блокировки. Каждый буфер в отдельности использует еще один объект для защиты другой информации, содержащейся внутри буфера. Такой объект необходимо удерживать во время использования буфера.

Если нити необходимо загрузить один из дисковых блоков, сначала она запрашивает буфер (используя для этого хэширование или другие указатели), после чего блокирует его. Затем нить блокирует список LRU для удаления из него полученного блока. Следовательно, обычный порядок блокировки можно описать фразой: *«сначала буфер, затем — список»*.

В некоторых случаях нити нужно получить любой свободный буфер, для чего она обращается к заголовку списка LRU. При этом нить сначала блокирует список и только затем блокирует буфер, оказавшийся в списке первым, после чего удаляет его из списка. Этот случай демонстрирует изменение порядка блокировка на противоположный: нить сначала блокирует список и только затем — буфер.

Можно заметить, что в этом случае возможно возникновение взаимоблокировки. Представьте, что одна нить захватывает буфер из заголовка списка и затем пытается заблокировать сам список. В тот же момент времени другая нить, которая уже заблокировала список, пытается занять буфер из его заголовка. Это приведет к тому, что каждая нить приостановит выполнение в ожидании освобождения блокировки, удерживаемой другой нитью.

Для предупреждения возникновения подобных ситуаций ядро системы использует методику вероятностной блокировки. Если нить пытается получить объект синхронизации в нарушение иерархии, она использует функцию try_lock() вместо lock(). Операция try_lock() применяется для получения объекта, но результатом ее работы в случае обнаружения занятости объекта окажется не блокировка нити, а выход с ошибкой. В приведенном выше примере нить, которой необходимо получить любой свободный буфер, сначала заблокирует список, а затем будет пытаться получить первый свободный объект при помощи операции try_lock(), перемещаясь по списку. Листинг 7.10 показывает пример применения функции try_lock() над простыми объектами синхронизации.

Листинг 7.10. Пример использования операции try_lock()

```
int try_lock(spinlock_t *s)
{
   if (test_and_set(s) != 0) /* объект уже занят */
      return FAILURE;
   else
      return SUCCESS;
}
```

7.10.2. Рекурсивная блокировка

Блокировка называется рекурсивной, если попытка захвата объекта, которым уже обладает нить, окажется успешной и не повлечет приостановку ее выполнения. В каких случаях можно использовать эту возможность? В каких ситуациях нить пытается запросить объект, которым она уже обладает? Обычно такое происходит, если нить, удерживающая ресурс, вызывает процедуру низкого уровня, производящую какие-либо операции с этим ресурсом[1]. Процедуры могут быть вызваны и другими командами высшего уровня, не занявшими перед этим ресурс. Становится ясно: процедура низкого уровня может попросту не знать о том, что ресурс является заблокированным. Если процедура попытается произвести блокировку, результатом станет зависание процесса.

Описанные выше ситуации можно предупредить, информируя процедуру о блокировке ресурса при помощи дополнительного входного аргумента. Но, к сожалению, такой подход требует изменения интерфейсов взаимодействия. Он может привести к определенным неудобствам, так как процедуры способны производить вызовы дополнительных функций. В результате обновленный интерфейс потеряет свойство модульности. Альтернативным решением проблемы является применение рекурсивной блокировки. При этом возникают определенные перегрузки, так как объекту блокировки необходимо

[1] Или при рекурсивном вызове самой себя. — *Прим. ред.*

хранить некий идентификатор владельца и проверять его каждый раз при блокировке или запрещении запроса. Однако более важным является тот факт, что рекурсивная блокировка позволяет иметь дело только с общими требованиями, независимо от того, какой конкретный объект блокировки используется нитью. Применение рекурсивной блокировки позволяет строить понятные модульные интерфейсы.

Примером использования методики является осуществление записи каталога файловой системы BSD (*ufs*). Для записи каталогов и отдельных файлов применяется одна и та же процедура ufs_write(). Доступ к записываемому файлу обычно осуществляется через элемент файловой таблицы, предоставляющей указатель на объект vnode этого файла. Таким образом, объект vnode передается напрямую процедуре ufs_write(), которая и должна в дальнейшем осуществить его блокировку. Однако дескриптор vnode при осуществлении записи каталога запрашивается посредством процедуры просмотра имен путей, возвращающей этот объект в уже заблокированном состоянии. Если затем произвести вызов ufs_write() для записи в каталог, результатом станет взаимоблокировка. Для предупреждения возникновения такого состояния необходимо использовать рекурсивную блокировку.

7.10.3. Что лучше: приостановка выполнения или ожидание в цикле?

Сложная синхронизация объектов может быть реализована как блокирующий объект или как сложный ожидающий объект без проведения изменений в свойствах или интерфейсах обоих типов объектов. Представьте, что ресурс защищен при помощи сложного объекта синхронизации (например, посредством семафора или объекта синхронизации чтения-записи). В реализации большинства механизмов, описываемых в этой главе, если нить пытается получить доступ к объекту и находит его уже занятым, она приостанавливает выполнение в ожидании освобождения этого объекта. Нить может и продолжать выполнение, осуществляя цикл ожидания ресурса.

Выбор между приостановкой выполнения и ожиданием в цикле зависит чаще всего от соображений производительности работы системы. В режиме ждущего цикла занимается процессор, что делает выбор этого варианта нежелательным. Но в системе иногда возникают ситуации, при которых ждущий цикл оказывается удобнее всего. Если нить уже занимает простой объект (mutex) синхронизации, он не имеет права блокировки. Если нить попытается получить еще один простой объект, то она перейдет в режим циклического ожидания. В случае необходимости получения сложного объекта синхронизации занимаемый нитью объект-mutex будет освобожден (точно по такому же алгоритму происходят действия с условными переменными).

Операции приостановки и возобновления выполнения нити являются громоздкими сами по себе, так как требуют проведения контекстных пере-

ключений и манипуляций с очередями сна и планирования. Одинаково неразумным представляется ожидание ресурса, который удерживается в течение продолжительного периода времени, точно так же как и приостановка выполнения нити в ожидании объекта, который вскоре может быть освобожден.

Более того, некоторые ресурсы могут удерживаться как на малый, так и на большой промежуток времени, в зависимости от определенных условий. Например, ядро может хранить в памяти некий список свободных дисковых блоков. Если список становится пустым, его необходимо заполнить новыми данными о свободных блоках. В большинстве случаев список блокируется на небольшие интервалы времени, в течение которых происходит добавление или удаление его элементов. Если при этом необходимо производить операции ввода-вывода, список окажется заблокированным на большой период времени. Следовательно, ни ждущий цикл, ни блокировка не окажется в данном случае идеальным выходом. Одним из вариантов решения проблемы является использование двух объектов синхронизации. При этом блокирующий объект применяется только в случае необходимости пополнения списка. Однако наиболее предпочтительным вариантом в этом случае является использование другого, более гибкого элемента синхронизации.

Один из наиболее приемлемых вариантов решения проблемы заключается в сохранении в объекте синхронизации указания, по которому и определяется, как нужно поступать нити: приостанавливать работу или выполнять цикл ожидания. Указание устанавливается обладателем ресурса и проверяется при неудачном завершении попытки получения объекта синхронизации. Указание может иметь как рекомендательный, так и директивный характер.

Альтернативное решение под названием *адаптивной блокировки* (adaptive locks) представлено в системе Solaris 2.x [7]. Если нити T1 необходимо получить адаптивный объект, удерживаемый нитью T2, то сначала производится проверка выполнения T2 на одном из процессоров системы. Если T2 выполняется в текущий момент времени, нить T1 перейдет в режим циклического ожидания; если T2 окажется заблокированной, T1 также приостановит свою работу.

7.10.4. Объекты блокировки

Блокировка применяется для защиты различных типов объектов: данных, логических выражений, инвариантных выражений или операций. К примеру, объекты блокировки чтения-записи используются для защиты данных. Условные переменные обычно ассоциируются с логическими выражениями. Инвариантное выражение имеет определенные сходства с логическим выражением, но отличается в семантике. Если объект защищен инвариантом, то это означает, что его значение будет равно TRUE всегда, кроме случаев удержания объекта кем-либо. В качестве примера можно привести связанный список, который использует единственный объект блокировки при добавлении или удалении записей. Инвариантным выражением, защищающим список, может быть целостность его состояния.

Объекты блокировки можно применять для управления доступом к определенной операции или функции. При этом устанавливается ограничение на код программы, который может выполняться одновременно только на одном процессоре системы. На этом свойстве построена модель синхронизации *управляющих программ* (monitors) [9]. Во многих реализациях UNIX один из процессоров используется для выполнения участков ядра, не поддерживающих параллельность. Такой метод применяется для последовательного выполнения кода, не рассчитанного для работы на многопроцессорных машинах. Это часто приводит к возникновению эффекта «бутылочного горлышка» — следовательно, по возможности от него следует отказаться.

7.10.5. Степень разбиения и длительность

Производительность системы сильно зависит от степени разбиения объектов синхронизации. С одной стороны, в некоторых асимметричных многопроцессорных ОС коды ядра выполняются только на одном процессоре, называемом главным. В этом случае для всего ядра достаточно одного объекта блокировки. С другой стороны, система может применять большое количество таких объектов, предоставляя их каждой переменной данных. Понятно, что ни первое, ни второе решение не является идеальным. Объекты блокировки занимают большие объемы памяти. Производительность системы может снизиться из-за необходимости постоянного получения и освобождения блокировки. Повышается вероятность возникновения взаимоблокировки нитей, так как при большом количестве синхронизируемых ресурсов трудно поддерживать определенный порядок их блокировки.

Идеальное решение, как обычно, находится в районе золотой середины. По этому вопросу до сих пор не достигнут консенсус среди разработчиков ОС. Приверженцы методики крупногранулированной блокировки предлагают использовать сначала небольшое количество объектов, защищающих основные подсистемы, и добавлять их при возникновении эффекта «бутылочного горлышка». Однако в ОС Mach и некоторых других применяется структура с подробной детализацией, в которой блокируются отдельные объекты данных.

Длительность блокировки также требует тщательного анализа. Оптимальным вариантом является удержание ресурса на короткие промежутки времени, что минимизирует степень соперничества при попытке обладания им. Однако в некоторых случаях такой подход может привести к дополнительным операциям занятия и освобождения ресурса. Представьте, что нити требуется произвести две операции над одним и тем же объектом, каждая из которых требует его блокировки. После окончания первой операции нити необходимо совершить некоторое действие, не связанное с объектами. Нить может освободить ресурс после завершения первой операции и снова занять его при проведении второй. В такой ситуации лучшим решением может оказаться блокировка ресурса на весь промежуток времени, особенно если промежу-

точное действие производится быстро. Решения о длительности блокировки объекта должны приниматься в индивидуальном порядке.

7.11. Реализация объектов синхронизации в различных ОС

При описании различных методик синхронизации в разделах 7.5–7.8 было показано, что операционная система может использовать объекты совместно друг с другом, предлагая тем самым комплексный интерфейс синхронизации ресурсов. В этом разделе будут описаны средства синхронизации в основных многопроцессорных вариантах системы UNIX.

7.11.1. SVR4.2/MP

SVR4.2/MP — это версия системы SVR4.2, поддерживающая многопроцессорные архитектуры. ОС обеспечивает четыре типа синхронизации: простую блокировку, блокировку сна, блокировку чтения-записи и переменные синхронизации [17]. Каждый объект должен запрашиваться или освобождаться явно при помощи операций *xxx*_ALLOC и *xxx*_DEALLOC, где *xxx*_ — это префикс, зависящий от типа используемого объекта. Операции получения объекта имеют входные аргументы, которые могут быть использованы для отладки.

Простые объекты блокировки

Простой блокировкой называется нерекурсивный объект mutex, позволяющий производить краткосрочную блокировку ресурсов. Он не может удерживаться при операциях блокировки нитей. Объект задается переменной типа lock_t. Для его получения и освобождения используются следующие операции:

```
pl_t LOCK (lock_t *lockp, pl_t new_ipl);
UNLOCK (lock_t *lockp, pl_t old_ipl);
```

Вызов LOCK перед запросом объекта изменяет уровень приоритета до new_ipl и возвращает предыдущий уровень ipl. Эта переменная передается операции UNLOCK для восстановления предыдущего уровня приоритета.

Блокировка чтения-записи

Блокировка чтения-записи — это нерекурсивный метод синхронизации, основанный на семантике одного отправителя и нескольких получателей данных. Он не может удерживаться при операциях блокировки нитей. Объект задается переменной типа rwlock_t и поддерживает следующие операции:

```
pl_t RW_RDLOCK (rwlock_t *lockp, pl_t new_ipl);
pl_t RW_WRLOCK (rwlock_t *lockp, pl_t new_ipl);
void RW_UNLOCK (rwlock_t *lockp, pl_t old_ipl);
```

Действия с приоритетами прерываний идентичны приведенным для простой блокировки. Операции блокировки изменяют уровень ipl до указанного и возвращают предыдущий уровень. Операция RW_UNLOCK восстанавливает предыдущее значение уровня ipl. Объект блокировки чтения-записи также поддерживает операции RW_TRYRDLOCK и RW_TRYWRLOCK, не приостанавливающие работу нити.

Блокировка сна

Блокировкой сна является применение нерекурсивных взаимных исключений, поддерживающих длительное удержание ресурсов. Такой объект может удерживаться при операциях блокировки нитей. Он реализован в виде переменной типа sleep_t и поддерживает следующие операции:

```
void SLEEP_LOCK (sleep_t *lockp, int pri);
bool_t SLEEP_LOCK_SIG (sleep_t *lockp, int pri);
void SLEEP_UNLOCK (sleep_t *lockp);
```

Параметр pri используется для указания приоритета планирования, который будет присвоен процессу после его пробуждения. В случае, когда процесс блокируется в результате вызова SLEEP_LOCK, он не может прерываться сигналом. Если процесс блокируется в результате вызова SLEEP_LOCK_SIG, поступивший сигнал разбудит процесс. Функция возвратит TRUE, когда объект будет получен, или FALSE, если сон процесса окажется прерванным. Объекты блокировки сна поддерживают и другие операции, такие как SLEEP_LOCK_AVAIL (проверка доступных объектов), SLEEP_LOCKOWNED (проверяет, не является ли вызывающий процесс обладателем объекта), SLEEP_TRYLOCK (возвращает ошибку вместо блокировки процесса в случае невозможности получения объекта).

Переменные синхронизации

Переменные синхронизации идентичны условным переменным, описанным в разделе 7.7. Они реализованы в виде переменных типа sv_t. Условие, которое должно быть защищено простым объектом блокировки, проверяется отдельно пользователями переменной синхронизации. Поддерживаются следующие операции:

```
void SV_WAIT (sv_t *svp, int pri, lock_t *lockp);
bool_t SV_WAIT_SIG (sv_t *svp, int pri, lock_t *lockp);
void SV_SIGNAL (sv_t *svp, int flags);
void SV_BROADCAST (sv_t *svp, int flags);
```

Так же как и в предыдущем случае, аргумент pri указывает на приоритет диспетчеризации, который будет присвоен процессу после его пробуждения, а операция SV_WAIT_SIG позволяет прерывание сна по сигналу. Аргумент lockp используется для передачи указателя на простой объект блокировки, защищающий логическое условие. Перед вызовом SV_WAIT или SV_WAIT_SIG

процесс должен занять объект lockp. Эти операции являются неделимыми и приводят к блокировке вызывающего процесса ядром и освобождении объекта, заданного при помощи lockp. После возврата SV_WAIT или SV_WAIT_SIG объект lockp оказывается незанятым. Заданное условие не всегда является истинным выражением в момент возобновления работы вызывающей нити, поэтому при использовании SV_WAIT или SV_WAIT_SIG необходимо производить его циклическую проверку.

7.11.2. Digital UNIX

Механизмы синхронизации системы Digital UNIX основаны на элементах Mach. Поддерживается два типа синхронизации: простая и комплексная [4]. Простая блокировка выполняется при помощи объекта mutex, реализованного на основе неделимой инструкции тестирования и установки конкретного типа аппаратной архитектуры. Объект перед использованием необходимо описать и проинициализировать, при этом он окажется в незанятом состоянии. Блокировка не будет удерживаться при операциях приостановки-возобновления выполнения нитей или при контекстных переключениях.

Сложная блокировка представляет собой элемент высокого уровня, поддерживающий большое количество возможностей, таких как разделяемый и эксклюзивный доступ, блокировка нитей или рекурсивные объекты. Такой элемент можно использовать для организации доступа чтения-записи ресурса. Он поддерживает две опции, sleep (режим сна) и recursive (рекурсивность). Опция сна может быть включена или отключена как на стадии инициализации объекта, так и позже при его использовании. Если она установлена, ядро будет блокировать все запросы в случае неудачного завершения попытки получения объекта. Более того, в случае необходимости блокировки нити во время удержания объекта опция сна должна быть обязательно установлена. Рекурсивность задействуется только в случае получения объекта в эксклюзивное пользование и может быть снята только той нитью, которая ранее установила ее.

Поддерживаются неблокирующие варианты различных процедур, возвращающие ошибку при невозможности получения объекта блокировки. Предлагаются функции изменения статуса объекта из *разделяемого* в *эксклюзивный* (upgrade) и обратно (downgrade). Операция повышения уровня освободит разделяемый объект и возвратит ошибку в том случае, если другая нить находится в режиме ожидания такого же запроса. Неблокирующий вариант операции upgrade в этом случае возвращает ошибку, но не освобождает совместный объект блокировки.

Сон и пробуждение

Так как большая часть кода системы была взята из 4BSD, в Digital UNIX оставлена поддержка вызовов sleep() и wakeup(). Таким образом, блокируе-

мые нити попадают в глобальные очереди сна чаще, чем в очереди отдельных объектов синхронизации. Для корректной работы механизма на многопроцессорных архитектурах алгоритмы блокировки необходимо подвергнуть изменениям. Основная проблема, которой следует уделить внимание, это потеря сигнала необходимости пробуждения нити. С этой целью исходные коды функции sleep() были переписаны заново с использованием двух операций низшего уровня, assert_wait() и thread_block(). Схема работы продемонстрирована на рис. 7.9.

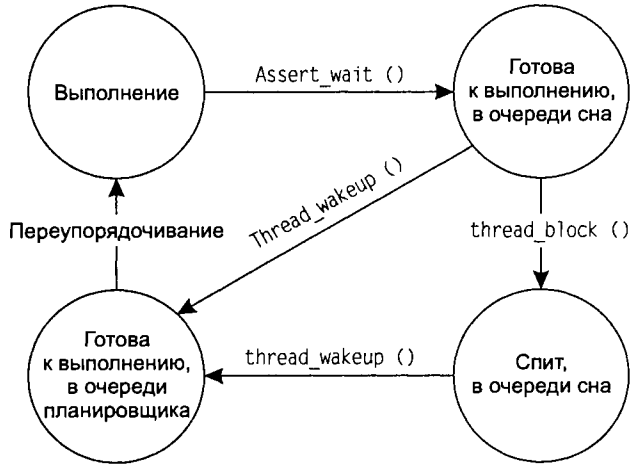


Рис. 7.9. Реализация перехода в режим сна в системе Digital UNIX

Представьте, что нити необходимо ожидать наступления некоторого события, описанного логическим выражением и защищенного при помощи простого объекта синхронизации. Нить запрашивает объект, после чего проверяет условие. Если в дальнейшем необходимо приостановить выполнение нити, происходит вызов assert_wait(), который перемещает ее в соответствующую очередь сна. Затем происходит освобождение объекта и вызов функции thread_block() для инициализации переключения контекста. Если событие произойдет в промежутке времени между освобождением объекта и контекстным переключением, ядро удалит нить из очереди сна и переместит его в очередь планирования. Следовательно, при применении вышеописанных функций нить не пропустит момент необходимости возобновления выполнения.

7.11.3. Другие реализации систем UNIX

Первая версия SVR4, поддерживающая многопроцессорность, была создана в NCR [3]. В ней была представлена концепция консультативных *объектов блокировки процессора* (advisory processor locks, APL), при использовании

которой рекурсивные объекты содержат указание действий над нитями. Рекомендация (или hint) подсказывает, что должна делать нить: приостановить выполнение или перейти в ждущий цикл ожидания ресурса. Также можно задать обязательный или рекомендательный характер исполнения указания. Нить, обладающая объектом блокировки, может изменить указание с необходимости приостановки выполнения на ожидание в цикле и обратно. Объекты APL обычно используются для доступа нити к долгосрочно блокируемым ресурсам. Особенностью объектов APL является их автоматическое освобождение и запрос при проведении переключения контекста. Это означает возможность применения традиционного интерфейса сна-пробуждения без проведения каких-либо изменений. Более того, объекты блокировки одного и того же класса защищены от проблемы потери сигнала пробуждения, так как при переходе нити в режим сна происходит освобождение всех ранее удерживаемых объектов. Система также поддерживает нерекурсивные взаимные исключения и объекты APL защиты чтения-записи.

Версия SVR4, представленная NCR, в дальнейшем была усовершенствована консорциумом Intel Multiprocessor Consortium, созданным группой компаний для разработки официальной многопроцессорной реализации системы [14]. Одним из самых важных изменений, произведенных в этой ОС, стал вызов функции, запрашивающей объект APL. Добавился новый входной аргумент – уровень приоритета прерываний. Это позволило изменять уровни ipl при манипуляциях с объектами блокировки. Если объект не может быть получен сразу же, цикл занятого ожидания будет иметь изначальный (более низкий) приоритет. Функция возвращает оригинальный уровень ipl, значение которого в дальнейшем может быть передано функции освобождения объекта.

Элементами наиболее низкого уровня являются наборы неделимых арифметических и логических операций. Арифметические операции позволяют инкрементировать и декрементировать значения счетчиков ссылок. Логические операции применяются для побитовых манипуляций с полями флагов. Операции обоих типов возвращают оригинальное значение переменной. На следующем, более высоком уровне находятся простые объекты блокировки (spin locks), не освобождаемые автоматически при переключениях контекста. Они применяются для простейших операций, таких как добавление или удаление элементов из очереди. На самом высоком уровне находится *блокировка ресурсов*. Это объекты долгосрочного использования, основанные на семантике одной читающей и множества пишущих нитей, которые могут удерживаться при проведении операций блокировки нитей. Система также поддерживает синхронные и асинхронные *межпроцессорные прерывания*, которые могут быть использованы для проведения таких операций, как *распределение тиков таймера* и *согласования кэша трансляции адресов* (см. раздел 15.9).

В системе Solaris 2.x для увеличения производительности применяются адаптивные объекты блокировки (см. раздел 7.10.3) и турникеты (см. раздел 7.2.3).

ОС поддерживает семафоры, объекты защиты чтения-записи и условные переменные как объекты синхронизации высокого уровня. Для обработки прерываний служат нити ядра, поэтому обработчики прерываний используют те же элементы синхронизации, что и остальная часть ядра, и могут блокировать их по необходимости. Эта возможность подробнее описывалась в разделе 3.6.5.

Реализация любой многопроцессорной системы использует одну из форм простых объектов блокировки для краткосрочной синхронизации на низком уровне. Механизм сна-пробуждения чаще всего поддерживается этими системами (иногда с небольшими изменениями), что защищает от необходимости замены большого количества исходного кода ОС. Основные различия наблюдаются в выборе элементов синхронизации высшего уровня. Первые реализации для IBM/370 и AT&T 3B20A [1] основывались в основном на семафорах. ОС Ultrix [16] использует эксклюзивную блокировку объектов. Ядро системы UTS, созданной Amdahl, основано на условиях [15]. ОС DG/UX применяет для реализации *последовательных объектов блокировки неделимые счетчики событий*, предоставляющие несколько нестандартный способ пробуждения одного процесса.

7.12. Заключение

Проблемы синхронизации в многопроцессорных архитектурах являются более сложными и сильно отличаются от проблем, возникающих при использовании одного процессора. Привлекается большое количество новых решений, таких как механизмы сна-пробуждения, условия, события, объекты защиты чтения-записи и семафоры. Все эти элементы во многом схожи. К примеру, по возможности следует применять семафоры совместно перед условиями и наоборот. Многие из представленных решений не ограничены многопроцессорными системами и могут быть применены при синхронизации на однопроцессорных архитектурах и распределенных системах со слабой связью. Многие многопроцессорные системы UNIX основаны на существующих однопроцессорных вариантах ОС, поэтому для них решение об использовании тех или иных элементов синхронизации сильно зависит от соглашений при переносе на новую платформу. Система Mach и другие, базирующиеся на ней, не зависят от этих соглашений, поэтому разработчики могли выбирать элементы синхронизации на свое усмотрение.

7.13. Упражнения

1. Многие системы поддерживают неделимую функцию, которая выполняет перестановку значения регистра в значение, хранимое в памяти. Как эта функция может быть использована для реализации неделимой операции тестирования и установки?

2. Каким образом можно реализовать неделимую операцию тестирования и установки на машине, поддерживающую связанную загрузку и сохранение по условию?
3. Представьте, что соперничество при попытке обладания критическим участком кода, защищенным семафором, приводит к конвоированию семафора. Если такой участок поделить на две части, каждая из которых будет защищена отдельным семафором, уменьшит ли это вероятность возникновения конвоя?
4. Одним из методов предупреждения возникновения конвоирования является замена семафоров другим механизмом блокировки. Может ли это привести к увеличению степени риска устаревания нитей?
5. Чем отличается счетчик ссылок от совместного объекта блокировки?
6. Создайте объект блокировки ресурса на основе простого объекта mutex и условной переменной, в которой проверке подвергается состояние флага locked (см. раздел 7.7.3).
7. Нужно ли удерживать простой объект блокировки, защищающий условие, при сбросе флага (в примере, описанном в предыдущем упражнении)? В работе [15] описывается операция waitlock(), использование которой может улучшить алгоритм.
8. Каким образом условные переменные могут предупреждать возникновение проблемы потери сигнала пробуждения нити?
9. Создайте элемент event (событие), возвращающий ожидающим нитям значение статуса при совершении события.
10. Представьте объект, к которому часто происходит доступ для чтения и записи. В каких ситуациях целесообразнее защитить его простым объектом mutex, а в каких использовать блокировку чтения-записи?
11. Имеет ли возможность объект защиты чтения-записи блокировать нити? Создайте объект защиты чтения записи, который заставляет нити переходить в режим занятого ожидания при невозможности получить ресурс.
12. Опишите ситуацию, в которой вероятность возникновения взаимоблокировки уменьшается при повышении степени грануляции.
13. Опишите ситуацию, в которой вероятность возникновения взаимоблокировки уменьшается при понижении степени грануляции.
14. Необходимо ли защищать каждый ресурс или переменную ядра многопроцессорной системы объектом блокировки перед доступом к ним? Подсчитайте количество различных типов ситуаций, при которых нить может иметь доступ или изменять объект без его блокировки.
15. Программы управления (*monitors*) — это конструкции, поддерживаемые языками программирования, использующиеся для взаимного ис-

ключения участка кода. При каких ситуациях применение этих механизмов является наиболее естественным?

16. Создайте функции `upgrade()` и `downgrade()` для реализации объектов защиты чтения-записи, представленных в разделе 7.8.2.

7.14. Дополнительная литература

1. Bach, M, and Buroff, S., «Multiprocessor UNIX Operating Systems», AT&T Bell Laboratories Technical Journal, Vol. 63, Oct. 1984, pp. 1733–1749.
2. Birrell, A. D., «An Introduction to Programming with Threads», Digital Equipment Corporation Systems Research Center, 1989.
3. Campbell, M., Barton, R., Browning, J., Cervenka, D., Curry, B., Davis, T., Edmonds, T., Holt, R., Slice, R., Smith, T., and Wescott, R., «The Parallelization of UNIX System V Release 4.0», Proceedings of the Winter 1991 USENIX Conference, Jan. 1991, pp. 307–323.
4. Denham, J. M., Long, P., and Woodward, J. A., «DEC OSF/1 Version 3.0 Symmetric Multiprocessing Implementation», Digital Technical Journal, Vol. 6, No. 3, Summer 1994, pp. 29–54.
5. Digital Equipment Corporation, «VAX Architecture Reference Manual», 1984.
6. Dijkstra, E. W., «Solution of a Problem in Concurrent Programming Control», Communications of the ACM, Vol. 8, Sep. 1965, pp. 569–578.
7. Eykholt, J. R., Kleinman, S. R., Barton, S., Faulkner, R., Shivalingiah, A., Smith, M., Stein, D., Voll, J., Weeks, M., and Williams, D., «Beyond Multiprocessing: Multithreading the SunOS Kernel», Proceedings of the Summer 1992 USENIX Conference, Jun. 1992, pp. 11–18.
8. Goble, G. H., «A Dual-Processor VAX 11/780», USENIX Association Conference Proceedings, Sep. 1981.
9. Hoare, C. A. R., «Monitors: An Operating System Structuring Concept», Communications of the ACM, Vol. 17, Oct. 1974, pp. 549–557.
10. Hitz, D., Harris, G., Lau, J. K., and Schwartz, A. M., «Using UNIX as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers», Proceedings of the Winter 1990 USENIX Technical Conference, Jan. 1990, pp. 285–295.
11. Kelley, M. H., «Multiprocessor Aspects of the DG/UX Kernel», Proceeding of the Winter 1989 USENIX Conference, Jan. 1989, pp. 85–99.
12. Lee, T. P., and Luppi, M. W., «Solving Performance Problems on a Multiprocessor UNIX System», Proceedings of the Summer 1987 USENIX Conference, Jun. 1987, pp. 399–405.

13. National Semiconductor Corporation, «Series 32000 Instruction Set Reference Manual», 1984.
14. Peacock, J. K., Saxena, S., Thomas, D., Yang, F., and Yu, F., «Experiences from Multithreading System V Release 4», The Third USENIX Symposium of Experiences with Distributed and Multiprocessor Systems (SEDMS III), Mar. 1992, pp. 77–91.
15. Ruane, L. M., «Process Synchronization in the UTS Kernel», Computing Systems, Vol. 3, Summer 1990, pp. 387–421.
16. Sinkewicz, U., «A Strategy for SMP ULTRIX», Proceedings of the Summer 198S USENIX Technical Conference, Jun. 1988, pp. 203–212.
17. UNIX System Laboratories, «Device Driver Reference — UNIX SVR4.2», UNIX Press, Prentice-Hall, Englewood Cliffs, NJ, 1992.

Глава 8
Базовые элементы и интерфейс файловой системы

8.1. Введение

Операционная система должна предоставлять средства постоянного хранения и обработки информации. В UNIX понятие *файл* используется для обозначения контейнера для данных. *Файловая система* позволяет производить различные действия с файлами и обладать доступом к ним[1]. В этой главе описывается интерфейс взаимодействия между файловой системой и пользовательскими приложениями, а также базовые элементы, используемые ядром для поддержки различных файловых систем. Главы 9, 10 и 11 посвящены конкретным реализациям файловых систем, позволяющим получить доступ к данным, как на локальных, так и на удаленных машинах.

Интерфейс объединяет системные вызовы и утилиты, при помощи которых пользовательские программы могут производить различные операции над файлами. Команды и утилиты остаются неизменными уже много лет, подвергаясь лишь небольшой шлифовке и не теряя совместимости с предыдущими версиями. С другой стороны, базовые элементы файловой системы были детально пересмотрены. Первые ОС поддерживали только один тип файловой системы. Все файлы располагались локально и хранились на одном или нескольких физических дисках компьютера. Позже была произведена замена этой файловой системы на *интерфейс vnode/vfs*, позволяющий использовать на одной машине несколько различных типов файловых систем, которые могут быть как локальными, так и удаленными.

Ранние коммерческие варианты UNIX имели простую файловую систему, известную сейчас под названием *s5fs* (System V file system) [14]. Ее поддерживают все существующие реализации System V UNIX, а также все версии

[1] В UNIX термин «файловая система» понимается не только как управляющая каталогами и файлами программная оболочка, но и непосредственно как иерархия каталогов и файлов, в дальнейшем различать их следует по контексту повествования. — *Прим. ред.*

ОС от Berkeley до 4.2BSD. В ОС 4.2BSD разработчиками была представлена новая файловая система *FFS* (Fast File System) [10], имеющая лучшие характеристики производительности и большую функциональность по сравнению с s5fs. Система FFS получила широкое распространение, кульминацией которого было ее включение в SVR4. В главе 9 вы увидите описания s5fs и FFS, а также некоторых других специализированных файловых систем, основанных на архитектуре vnode/vfs.

ПРИМЕЧАНИЕ

Термины FFS и ufs (UNIX file system, файловая система UNIX) часто заменяют друг друга. Однако если быть более точными, FFS — это оригинальная реализация Fast File System от разработчиков из Беркли, а ufs является версией FFS с поддержкой элементов vnode/vfs. В этой книге мы будем придерживаться более точных значений приведенных терминов.

После появления возможности соединять компьютеры друг с другом сетью разработчики систем стали искать способы получения доступа к файлам на удаленных узлах. В середине 80-х годов конкурировало несколько технологий доступа, предлагающих совместное прозрачное использование файлов, находящихся на компьютерах, объединенных между собой. В главе 10 вы увидите описание трех наиболее распространенных вариантов систем: Network File System (NFS), Remote File Sharing (RFS) и Andrew File System (AFS).

В последние годы появились новые файловые системы, расширяющие возможности FFS либо применяемые специализированными приложениями. Большинство из них основано на таких сложных технологиях, как поддержка журналов, моментальные снимки (snapshots) и управление томами, используемых для увеличения производительности, надежности и работоспособности. Рассказ о некоторых современных файловых системах вы найдете в главе 11.

8.2. Интерфейс доступа пользователя к файлам

Ядро ОС UNIX позволяет пользовательским процессам взаимодействовать с файловой системой через строго определенный процедурный интерфейс. Интерфейс определяет представление файловой системы с точки зрения пользователя и семантику и действия всех относящихся к ней системных вызовов. Пользователь оперирует лишь некоторыми абстракциями, такими как *файлы*, *каталоги*, *дескрипторы файлов* и *файловые системы*.

Как мы уже говорили, на сегодняшний день существует несколько различных типов файловых систем, например s5fs или FFS. В каждой системе реализован одинаковый интерфейс, дающий приложениям одинаковое представление всех файлов. Каждая файловая система в отдельности может обла-

дать определенными ограничениями с точки зрения интерфейса. Например, s5fs позволяет использовать в имени файла до 14 символов, в то время как пользователи FFS могут создавать файлы с именами длиной до 255 символов.

8.2.1. Файлы и каталоги

Логически файл представляет собой место для хранения данных. Пользователь может создать файл и затем использовать его для хранения информации путем записи в него. Доступ к данным может быть как последовательным, так и произвольным. Ядро системы поддерживает различные операции управления, позволяющие называть файлы, использовать их и управлять доступом к ним. Ядро не интерпретирует содержание или структуру файла, трактуя его как простую последовательность байтов и предоставляя побайтовый доступ к его содержимому. Некоторые приложения требуют применения более сложной семантики, например индексированного доступа или доступа к отдельным записям. Такие механизмы могут быть созданы разработчиками самостоятельно на основе базисных элементов ядра.

С точки зрения пользователя файлы в системе UNIX организованы в виде иерархического древовидного пространства имен (см. рис. 8.1). Дерево состоит из файлов и каталогов («ветвей»), «листьями» которых являются файлы[1]. Каталог содержит информацию об именах файлов и других каталогов, находящихся внутри него. Каждое имя файла или каталога содержит любые символы ASCII, кроме «/» и NULL. Файловая система может ограничивать длину имен файлов. *Корневой* каталог называется /. Имена файлов должны отличаться друг от друга только в пределах одного каталога. К примеру, на рис. 8.1 в каталогах bin и etc находятся файлы с одинаковым именем passwd. Для уникальной идентификации файла необходимо указывать *полное имя* файла (или pathname[2]). Полное имя состоит из всех компонентов пути, начиная от корневого каталога до файла, разделенных между собой символом /. Следовательно, оба файла passwd на рисунке имеют одинаковое имя, но различаются полными именами: один из них называется /bin/passwd, в то время как второй — /etc/passwd. Символ «/» в системе UNIX используется для обозначения имени корневого каталога и разделителя составляющих полного имени файла.

В UNIX используется определение текущего рабочего каталога для каждого процесса, поддерживаемого как часть информации о его состоянии. Это позволяет пользователям оперировать с относительными именами файлов, интерпретируемых в зависимости от текущего каталога. Для этого применя-

[1] Существование твердой связи требует использования более точного определения дерева как прямого ациклического графа (если упустить элементы «..», служащие ссылками на родительский каталог). Однако с практической точки зрения сравнение файловой системы с деревом упрощает понимание и является более адекватным.

[2] Для pathname благозвучного русскоязычного термина еще не придумали, поэтому в дальнейшем это понятие будет обозначаться как «полное имя», или «составное имя». — *Прим. ред.*

ются два специализированных компонента пути файла: «.» (одна точка) указывает на текущий каталог сам по себе, а «..» (две точки) обозначает родительский каталог. Корневой каталог не имеет родительского, поэтому его компонент «..» ссылается сам на себя. На рис. 8.1 пользователь, для которого каталог /usr/local является текущим, может ссылаться на каталог lib при помощи имени /usr/lib, называемого *абсолютным именем* или ./lib, называемым *относительным именем* каталога. Процесс может изменить текущий каталог при помощи системного вызова chdir.

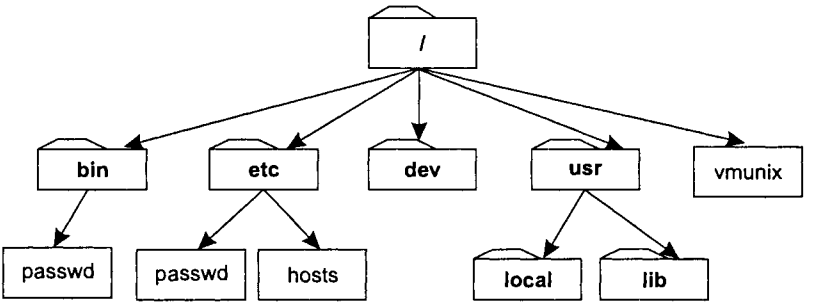


Рис. 8.1. Файлы организованы в виде дерева каталогов

Каталог файла называется *жесткой ссылкой*, или простой *ссылкой*, на этот файл. Каждый файл может обладать одной или несколькими ссылками на него, относящимися как к тому же, так и другим каталогам. Таким образом, файл не ограничен единственным каталогом и не имеет уникального имени. Имя не является атрибутом файла. Файл продолжает существовать до тех пор, пока счетчик ссылок не него не станет равным нулю. Все файловые ссылки эквивалентны и являются просто различными именами одного и того же файла. Доступ к нему можно осуществлять по любой ссылке, и чаще всего трудно сказать, какая из этих ссылок является изначальной. Современные варианты UNIX поддерживают еще один тип ссылки на файл, называемый *символическим* (см. раздел 8.4.1).

В каждом типе файловой системы используется свой внутренний формат каталогов. Так как создателям приложений необходимо переносить свои программы на другие системы, то для операций считывания содержимого каталогов в стандартах POSIX.1 были описаны следующие общие процедуры:

```
dirp = opendir (const char *filename);
direntp = readdir(dirp);
rewinddir (dirp);
status = closedir (dirp);
```

Эти процедуры были впервые представлены в системе 4BSD. На сегодняшний день они поддерживаются SVR4, а также большинством современных коммерческих вариантов UNIX. При вызове opendir библиотека производит ассоциацию с ним потока каталога и возвращает дескриптор потока пользо-

вателю. Объект потока поддерживает смещение на следующий элемент для чтения. Каждый вызов readdir возвращает один каталог и увеличивает смещение. Все элементы возвращаются в формате, независимом от файловой системы, определенными следующей структурой:

```
struct dirent {
   ino_t d_ino; /* номер индексного дескриптора (см. раздел 8.2 2 */
   char d_name[NAME_MAX + 1]; /* имя файла с завершающим нулевым
      символом */
};
```

Значение NAME_MAX зависит от типа файловой системы. ОС SVR4 поддерживает также вызов getdents, используемый для чтения элементов каталогов в независимом от системы формате. Представление элементов, возвращаемое getdents, отличается от структуры dirent. Следовательно, пользователи должны по возможности применять функции POSIX, обладающие большей переносимостью.

8.2.2. Атрибуты файлов

Система поддерживает отдельно от имен набор атрибутов для каждого файла. Атрибуты хранятся не в каталоге, а в специальной структуре на диске, называемой обычно *inode*. Слово inode произошло от сокращения *index node*, означающего *индексный дескриптор файла*. Формат и содержимое дескрипторов не совпадает в различных файловых системах. Для вывода атрибутов файла в формате, независимом от файловой системы, используются вызовы stat и fstat. Наиболее общими атрибутами файлов являются:

- тип. Система UNIX умеет распознавать несколько типов файлов, в том числе каталогов, файлов FIFO, символических ссылок, а также специализированных файлов, представляющих блочные или символьные устройства;
- количество жестких ссылок;
- размер в байтах;
- идентификатор устройства (device ID). Идентифицирует устройство, на котором размещен файл;
- номер индексного дескриптора. Каждому файлу или каталогу присваивается один дескриптор независимо от количества существующих ссылок. Каждый дескриптор на определенном разделе диска (или *логическом* диске; раздел 8.3.1) имеет уникальный *номер* (inode number). Таким образом, каждый файл идентифицируется уникальным образом при помощи его идентификатора устройства и номера индексного дескриптора. Эти идентификаторы не хранятся внутри дескрипторов.

Идентификатор устройства является характеристикой файловой системы, все файлы, находящиеся в одной и той же файловой системе, обладают одинаковым идентификатором устройства. Номер дескриптора хранится в элементах каталогов наряду с именем файла;

- идентификаторы пользователя и группы, являющейся владельцами файла;
- временные метки. Для каждого файла поддерживаются три метки: время последнего доступа, время последнего изменения и время изменения его атрибутов (не включающих другие временные метки);
- права и флаги режимов, описания которых будут приведены ниже.

С каждым файлом ассоциируются три типа прав: право на чтение (*read*), на запись (*write*) и на выполнение (*execute*). Все пользователи подразделяются также на три категории: владельца файла (*owner*)[1], пользователей группы, совпадающей с группой владельца (*group*) и всех остальных (*others*). Следовательно, все права на файл могут быть указаны при помощи 9 битов. Права на каталоги обрабатываются несколько по-другому. Доступ на запись в каталог позволяет создавать или удалять файлы, содержащиеся в нем. Права на выполнение разрешают иметь доступ к файлам каталога. Пользователь не может производить запись в каталог напрямую даже в том случае, если он обладает правами, разрешающими это действие. Содержимое каталога можно изменять только путем создания или удаления файлов.

Механизм прав доступа не только прост, но и достаточно примитивен. На сегодняшний день большинство производителей систем UNIX предлагают дополнительные средства защиты, добавляя их как в базовые версии своих ОС, так и в специализированные *защищенные* варианты[2]. Механизмы защиты систем обычно обеспечивают поддержку различных форм *списков контроля доступа* (ACL), позволяющих более детально указать распределение прав доступа на файлы [5].

Флаги режимов бывают трех различных типов: suid, sgid и sticky. Флаги suid и sgid относятся к выполняемым файлам. Если пользователь выполняет файл, для которого был установлен флаг suid, ядро изменит действительный идентификатор пользователя на ID владельца файла. Флаг sgid используется для аналогичного изменения действительного группового идентификатора (об идентификаторах подробнее см. в разделе 2.3.3). Так как флаг sgid не имеет никакого значения для неисполняемых файлов, он может перезаписы-

---

[1] Иногда владельца файла называют просто пользователем (*user*). Например, команда chmod использует сокращения u, g и o для обозначения пользователя, группы и остальных соответственно.

[2] Требования безопасности большинства систем UNIX основаны на наборе критериев оценки надежности компьютерных систем, опубликованных Министерством обороны США. Этот документ, получивший название «Оранжевой книги» [3], устанавливает несколько уровней безопасности операционных систем.

ваться для несколько иных целей. Если файлу не назначено прав на выполнение группой и флаг sgid является установленным, то над этим файлом разрешается производить принудительную блокировку файла/записи [15].

Флаг навязчивости sticky нужен также для выполняемых файлов и используется для запроса к ядру системы на сохранение «образа» программы в области свопинга (см. раздел 13.2.4) после завершения ее выполнения[1]. Системные администраторы часто устанавливают флаг sticky для наиболее часто выполняемых программ, увеличивая тем самым производительность системы. В большинстве современных реализаций UNIX все завершенные программы по возможности оставляются в области свопинга (при этом используется одна из вариаций алгоритма замены *недавно использовавшейся* информацией — LRU), таким образом, для них не требуется применение флага навязчивости.

Флаги sgid и sticky применяются немного по-другому для каталогов. Если флаг sticky установлен, а в каталог можно производить запись, то процесс обладает правом на удаление или изменение файла в этом каталоге только в том случае, если его действительный UID совпадает с идентификатором владельца файла или каталога, а также тогда, когда процесс обладает правами на запись файла. Если флаг sticky сброшен, то удалять или переименовывать файлы, находящиеся в каталоге, может любой процесс, обладающий правом на запись в этот каталог.

При создании файла значение действительного UID наследуется от создавшего его процесса. Идентификатор GID владельца при этом может иметь только два значения. В системе SVR3 файл наследовал действительный GID от своего создателя. В 4.3BSD наследуется GID каталога, в котором файл был создан. В системе SVR4 для выбора значения идентификатора используется флаг sgid родительского каталога. Если этот флаг был установлен для каталога, то при создании новых файлов значение GID наследуется от родительского каталога. Когда флаг sgid сброшен, новые файлы получают GID от своего создателя.

Для манипуляций с атрибутами файлов в UNIX предлагается специальный набор системных вызовов. Входным аргументом всех этих функций является полное имя файла. Вызовы link и unlink применяются соответственно для создания и удаления жесткой связи. Ядро удаляет файл только в том случае, если на него не остается ни одной ссылки и никто им не пользуется в текущий момент. Системный вызов utimes изменяет временные метки *доступа* и *изменения* для файла. Функция chown переназначает идентификаторы пользователя и группы владельца. Вызов chmod изменяет права и флаги режимов файла.

---

[1] Система старается не занимать память, в которую была загружена программа. При повторном запуске такую программу не нужно считывать с диска — достаточно передать на нее управление. — *Прим. ред.*

8.2.3. Дескрипторы файлов

Перед использованием файла процесс должен сначала открыть его для чтения или записи. Системный вызов open имеет следующий синтаксис:

```
fd = open (path, oflag, mode);
```

где path — это абсолютное или относительное полное имя файла, а mode указывает на права, ассоциируемые с файлом при его создании. Флаги, передаваемые при помощи переменной oflag, указывают на необходимость открытия файла после его создания на чтение (*read*), запись (*write*), чтение-запись (*read-write*), добавление (*append*) и т. д. Для создания файла также применяется системный вызов creat, который эквивалентен вызову open с флагами O_WRONLY, O_CREAT и O_TRUNC.

Каждый процесс имеет *маску создания файлов*, определенную по умолчанию, являющуюся битовой маской прав, *не* предоставляемых создаваемым файлам. Когда пользователь указывает параметр mode для функции open или creat, пользователь обнуляет разряды, указанные в маске по умолчанию. Для изменения маски, принятой по умолчанию, используется системный вызов umask. Пользователь может перезаписать маску уже после создания файла при помощи chmod.

При вызове функции open ядро создает *объект открытого файла*, соответствующий количеству открытий для этого файла. Ядро также выделяет *дескриптор файла*, являющийся ссылкой на объект открытого файла. Возвращаемой переменной вызова open и является дескриптор открываемого файла. Пользователь может открывать один и тот же файл неоднократно, его могут одновременно открывать и сразу несколько пользователей. В последнем случае ядро системы создаст новые объекты открытого файла и дескрипторы.

Дескриптор файла является объектом, относящимся к определенному процессу. Один и тот же дескриптор для двух процессов часто ссылается на разные файлы. Процесс использует дескрипторы для передачи ввода-вывода системным вызовам, таким как read или write. Ядру дескрипторы нужны для быстрого нахождения объекта открытого файла, а также других структур данных, ассоциируемых с этим файлом. Это дает возможность ядру производить различные действия, такие как анализ полного имени или управление доступом, на стадии вызова функции open, а не при проведении каждой операции ввода-вывода. Открытые файлы можно закрыть либо при помощи системного вызова close, либо это произойдет автоматически при завершении работы процесса.

Каждый дескриптор связан с независимым сеансом работы с файлом. Ассоциируемый с этим дескриптором объект открытого файла содержит контекст сеанса. Контекст включает в себя режимы, при которых было произведено открытие файла, а также *указатель смещения*, показывающий точ-

ку начала следующей операции чтения или записи. В системе UNIX по умолчанию доступ к файлам осуществляется последовательно. После открытия файла пользователем ядро производит инициализацию указателя смещения в значение «ноль». В дальнейшем каждая операция чтения или записи увеличит значение указателя на количество переданных данных.

Поддержка указателя смещения для объекта открытого файла позволяет ядру изолировать различные сеансы доступа к файлу друг от друга (рис. 8.2). Если два процесса открывают один и тот же файл[1], то операция чтения или записи одним из этих процессов приведет к сдвигу только собственного указателя смещения. Значение второго указателя при этом не изменяется. Такой подход обеспечивает прозрачное совместное использование файла несколькими процессами одновременно. Эта возможность системы должна применяться с осторожностью. В большинстве случаев доступ к одному и тому же файлу несколькими процессами влечет необходимость применения для его блокировки технологии синхронизации, описанной в разделе 8.2.6.

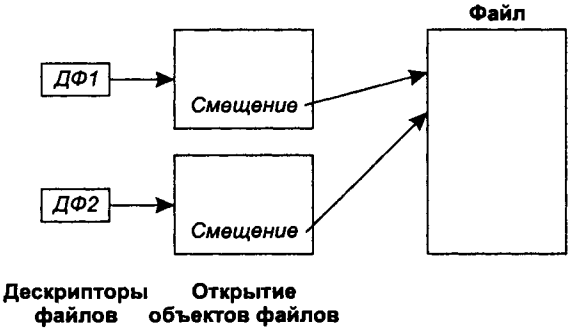


Рис. 8.2. Файл, открытый дважды

Процесс может дублировать дескрипторы при помощи системных вызовов dup или dup2. Эти функции создают новый дескриптор, указывающий на один и тот же объект открытого файла и, следовательно, разделяющий один и тот же сеанс (рис 8.3). Системный вызов fork производит создание копий всех файловых дескрипторов родительского процесса и передает их потомку. После возврата из fork предок и потомок разделяют между собой одинаковый набор открытых файлов. Эта форма совместного использования имеет фундаментальные отличия от нескольких экземпляров открытия файла. Так как два дескриптора разделяют между собой один и тот же сеанс работы, они оба видят файл одинаково и используют тождественный указатель смещения. Операция увеличения указателя смещения, произведенная над одним из дескрипторов, приведет к тому, что эти изменения (как правило) тут же окажутся видны и в остальных.

[1] Или если процесс открывает один и тот же файл дважды.

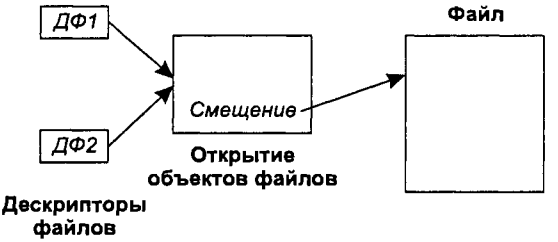


Рис. 8.3. Дескриптор, продублированный при помощи функций dup, dup2 или fork

Процесс может передавать дескриптор файла любому другому процессу, при этом достигается эффект передачи ссылки на объект открытого файла. Ядро производит копирование дескриптора в первую свободную ячейку таблицы дескрипторов получателя. В результате оба дескриптора разделяют между собой один и тот же объект и, следовательно, обладают единым указателем смещения. Обычно процесс, посылающий дескриптор, закрывает его после завершения операции отправки. Это не приводит к закрытию файла – даже в том случае, если адресат так и не получил дескриптор, поскольку ядро системы удерживает второй дескриптор во время проведения передачи.

Интерфейс передачи дескрипторов зависит от конкретной реализации UNIX. В системе SVR4 для этой цели используется вызов ioctl с командой I_SENDFD, при этом передача осуществляется через STREAMS (см. раздел 17.9). Процессы получают дескриптор при помощи команды I_RECVFD вызова ioctl. В системе 4.3BSD для передачи дескрипторов применяются вызовы sendmsg и recvmsg и соединение сокетов, осуществленное между двумя процессами (см. раздел 17.10.3). Передача дескрипторов может применяться в реализациях некоторых типов сетевых приложений. Один процесс, называемый сервером соединения, может настроить сетевое соединение для клиентского процесса, и затем передать дескриптор, представляющий соединение, этому клиенту обратно. В разделе 11.10 будет описана *переносимость* файловых систем в 4.4BSD, использующих развитие этой концепции.

8.2.4. Файловый ввод-вывод

Система UNIX позволяет осуществлять как произвольный, так и последовательный доступ к файлам. По умолчанию применяется последовательный доступ. Ядро поддерживает указатель смещения файла, который при первом открытии этого файла инициализируется в ноль. Он указывает на текущую позицию в файле, с которой будет производиться следующая операция ввода-вывода. Каждый раз, когда процесс считывает или записывает данные в файл, ядро сдвигает указатель на величину переданного объема данных. Произвольный доступ к файлу достигается с помощью системного вызова lseek, которому передается необходимое значение указателя смещения. Последующий вы-

зов функции read или write приведет к передаче данных начиная с указанной позиции.

Вызовы read и write имеют одинаковую семантику, поэтому мы приведем пример только одной из этих функций[1]:

```
nread = read (fd, buf, count);
```

где fd — это дескриптор файла, buf — указатель на буфер в пользовательском адресном пространстве, в который должно производиться чтение данных, а параметр count — количество считываемых байтов.

Ядро считывает данные из файла, ассоциированного с fd, начиная от смещения, сохраненного в объекте открытого файла. Возможно возникновение ситуации, когда количество считываемых байтов меньше, чем величина count. Это может произойти при достижении конца файла или в случае отсутствия доступных данных при обращении к файлам FIFO или устройствам. Например, при вызове read с терминала, работающего в *каноническом* режиме, выход из функции произойдет после ввода пользователем символа перевода каретки, даже в том случае, если строка содержит меньшее количество символов, чем было указано. Однако не существует каких-либо условий, при которых ядро системы могло бы передавать большее, чем значение count, количество байтов. Ответственность за емкость буфера buf, достаточную для помещения count байтов данных, несет пользователь. Вызов read возвращает количество переданных байтов (nread). Эта функция также производит смещение указателя на nread байтов, поэтому следующий вызов read или write начнет передачу данных с точки завершения действий предыдущей функции.

Хотя ядро позволяет нескольким процессам производить открытие и совместное использование файла, на самом деле эти операции последовательны. Например, если двум процессам необходимо одновременно произвести запись в один и тот же файл, ядро системы произведет вторую операцию записи только после завершения первой. Такой подход позволяет каждой операции работать с файлами, находящимися в непротиворечивом состоянии.

Файл может быть открыт в режиме добавления, для чего системному вызову open необходимо передать флаг O_APPEND. В этом случае перед вызовом write для указанного дескриптора ядро установит указатель смещения на конец файла. Если пользователь открывает файл в режиме добавления, то это не повлияет на операции с этим файлом, производимые через другие дескрипторы.

В многонитевых системах необходимо помнить о некоторых сложностях, связанных с возможностью совместного доступа к файлам. Например, в таких системах могут возникать ситуации, когда одна нить производит вызов

[1] Большинство программистов не применяет вызовы read и write напрямую. Вместо них используются стандартные библиотечные функции fread и fwrite, имеющие некоторые дополнительные возможности, такие как буферизация данных.

lseek перед тем, как другая нить вызовет read. В результате вторая нить начнет чтение с совершенно иного, отличного от заданного ею смещения. В некоторых ОС, например Solaris, поддерживаются специальные системные вызовы pread и pwrite, предлагающие неделимые операции позиционирования и чтения (записи). Более подробно эти функции обсуждались в разделе 3.6.6.

8.2.5. Ввод-вывод методом сборки-рассоединения

Системные вызовы read и write передают определенное количество байтов между логически непрерывным фрагментом файла и непрерывным диапазоном адресов памяти в адресном пространстве процесса. Однако многим приложениям необходимо производить неделимые операции чтения или записи данных в адресное пространство некоторого количества буферов, расположенных непоследовательно. Применение вызова read для этой цели является неэффективным, так как процессу приходится первоначально считывать все данные в единый буфер и только затем копировать их в нужные участки памяти. Система UNIX поддерживает два дополнительных вызова — readv и writev, производящие ввод-вывод методом сборки-рассоединения данных, при котором передача данных осуществляется из файла сразу в несколько буферов, расположенных в пользовательском адресном пространстве.

Примером использования метода сборки-рассоединения данных может служить сетевой протокол передачи файлов. Файлы из удаленного узла передаются по сети и записываются на локальный диск. Данные поступают в наборах сетевых пакетов, в каждом из которых находится только часть файла. Без применения описываемой технологии протоколу необходимо сохранять все полученные пакеты в одном буфере и затем записывать информацию из них на диск. Используя writev, протокол может произвести единственный простой запрос, который соберет данные из всех пакетов. Синтаксис функции writev приведен ниже:

```
nbytes = writev (fd, iov, iovcnt),
```

Формат readv аналогичен. Аргумент fd задает дескриптор файла, iov является указателем на массив пар <base, length> (начало, длина) в структуре iovec, описывающих набор буферов-источников. Аргумент iovcnt используется для передачи количества элементов в массиве. Как и в случае функции write, возвращаемой величиной вызова writev является число переданных байтов. Указатель смещения определяет начало данных в файле, ядро смещает его на величину nbytes перед завершением работы вызова.

На рис. 8.4 показано применение метода для записи файла. Ядро для управления операцией создает структуру uio и инициализирует ее, используя входные аргументы системного вызова и данные объекта открытого файла. Затем ядро передает указатель на структуру функциям нижнего уровня,

производящим ввод-вывод. Функция writev выполняет неделимую операцию передачи данных из всех заданных буферов в файл.

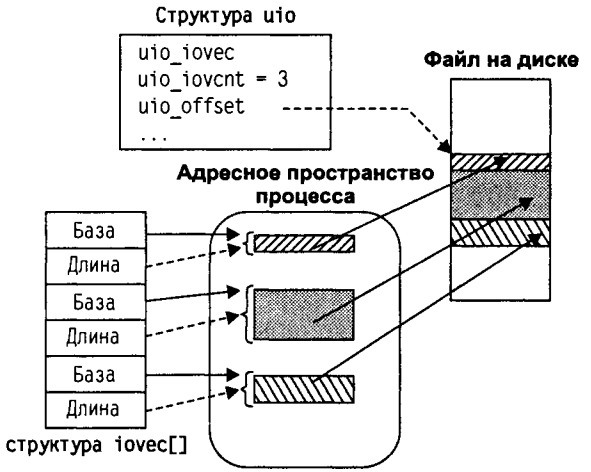


Рис. 8.4. Ввод-вывод методом сборки-разъединения данных

8.2.6. Блокировка файлов

В системах UNIX по умолчанию разрешен одновременный доступ к файлу нескольким процессам на чтение или запись. Каждый отдельный вызов read или write является неделимым, однако на уровне системы в целом синхронизация действий не производится. В результате если процесс производит чтение из файла порционно (применяя несколько вызовов read), другой процесс может между двумя операциями чтения внести изменения в этот файл.

Такой подход является абсолютно неудовлетворительным для большинства приложений, которым обычно требуется защищать файлы, к которым производится неоднократный доступ. Следовательно, система UNIX должна поддерживать некие средства блокировки файлов. Блокировка может быть рекомендательной или директивной. *Рекомендательная блокировка* производится без участия ядра и используется для защиты файла. При этом совместно работающие процессы должны явно проверять наличие блокировки. *Обязательная блокировка* производится ядром, которое и отсеивает операции, приводящие к конфликту. Запросы могут быть блокирующими либо неблокирующими. В последнем случае при невозможности получения объекта блокировки процесс получит код ошибки EWOULDBLOCK.

В системе 4BSD поддерживается вызов flock, реализующий рекомендательную блокировку отрытых файлов, но при этом разрешающий использовать как разделяемые, так и эксклюзивные объекты блокировки. Синхронизация доступа к файлам в ОС System V отличается от версии к версии. Система SVR2 поддерживает только рекомендательную блокировку, как для файлов,

так и для *записей* (диапазонов байтов внутри файлов). В SVR3 разработчики добавили возможность обязательной блокировки, но при этом необходимо предварительно разрешить применять к файлу рекомендательную блокировку через вызов chmod (см. раздел 8.2.2). Это средство совместимо на двоичном уровне с функциями XENIX. В систему SVR4 добавлены функции синхронизации файлов BSD, а также поддержка блокировки одного пишущего/множества читающих процессов. Для блокировки служит системный вызов fcntl, но большинство приложений использует более простой программный интерфейс, предлагаемый функцией библиотеки C, — lockf.

8.3. Файловые системы

Файловая иерархия системы UNIX видится как монолитная структура, но в реальности она представляет собой композицию нескольких отдельных деревьев, каждое из которых является отдельной, полной *файловой системой*. Одна из файловых систем настраивается как *корневая файловая система*, а ее корневой каталог становится *системным корневым каталогом*. Остальные файловые системы присоединяются к существующей структуре при помощи монтирования каждой новой файловой системы в каталог, находящийся в существующем дереве. После монтирования корневой каталог системы «покрывается» каталогом, на который система была смонтирована. Любой доступ к монтировочному каталогу преобразуется в доступ к корневому каталогу смонтированной файловой системы. Она остается видимой до тех пор, пока не будет произведена операция размонтирования.

На рис. 8.5 показана иерархия файлов, получаемая при композиции двух файловых систем. В примере система fs0 установлена как корневая файловая система машины, а fs1 смонтирована в каталоге /usr системы fs0. Каталог /usr называется монтировочным или точкой монтирования. Все попытки доступа к нему приведут к обращению к корневому каталогу монтированной на нем файловой системы.

Если каталог /usr содержит какие-либо файлы, они окажутся скрытыми в том случае, если на него будет смонтирована система fs1. Эти файлы окажутся недоступными для пользователей. При размонтировании fs1 файлы снова становятся видимыми и доступными для использования. Процедуры ядра для установления соответствия полных файловых имен должны обладать информацией о точках монтирования и корректно отрабатывать их пересечение в обоих направлениях. В оригинальной системе s5fs и различных реализациях FFS для отслеживания монтированных файловых систем применяется *таблица монтирования*. В современных системах UNIX для этой цели используются различные формы *списка vfs* (виртуальных файловых систем). О них мы расскажем чуть позже в разделе 8.9.

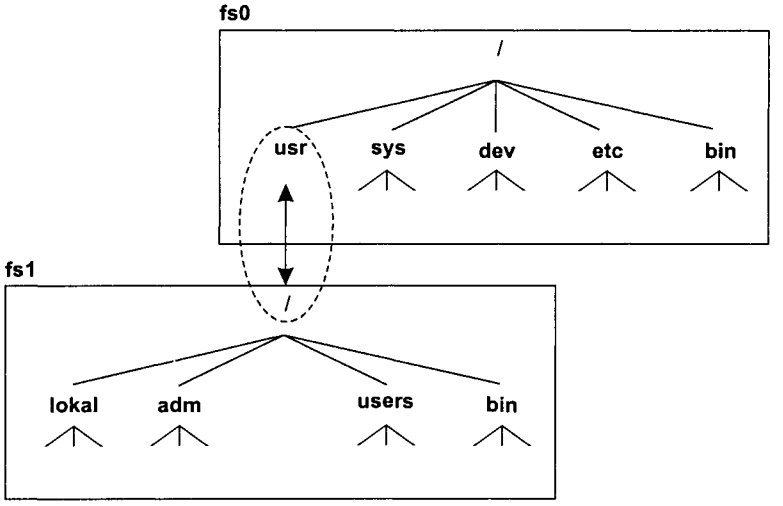


Рис. 8.5. Монтирование одной файловой системы в другую

Определение монтируемых подсистем скрывает подробности организации хранения информации от пользователя. Адресное пространство файлов является однородным, и пользователю не нужно указывать букву диска как часть имени файла (как это принято в системах MS-DOS или VMS). Файловые системы могут создаваться индивидуально для проведения резервного копирования, архивирования или восстановления. Системный администратор может регулировать уровень защиты каждой файловой системы, например, делая некоторые из них доступными только для чтения.

Монтируемые файловые системы накладывают некоторые ограничения на иерархию файлов. Размер файла может расти только до значения, определяемого величиной свободного пространства файловой системы, к которой он относится. Операции переименования и назначения жестких связей не могут «пересекать границу» файловой системы. Каждая файловая система должна располагаться на отдельном *логическом диске* и ограничивается размером этого диска.

8.3.1. Логические диски

Логический диск — это элемент, используемый для хранения информации, видимый ядром системы как линейная последовательность блоков фиксированного размера, доступных в произвольном порядке. Драйвер дискового устройства (см. раздел 16.6) производит отображение блоков на физические средства хранения. Для создания файловой системы на диске в UNIX используются утилиты newfs или mkfs. Каждая файловая система занимает один логический диск. Логический диск может содержать только одну файловую

систему. На некоторых логических дисках файловая система может отсутствовать, например, если диск используется подсистемой памяти для свопинга.

Поддержка логических дисков позволяет размечать физический объем устройства различными способами. В простейшем случае каждый логический диск отображает все физическое пространство диска полностью. Обычно диск разделяется на некоторое количество непрерывных *разделов*, каждый из которых является логическим устройством. В ранних системах UNIX встречался только такой способ разбиения дисков. В результате слово «раздел» и сейчас часто применяется для описания физического пространства файловой системы.

В современных системах UNIX поддерживаются и другие конфигурации хранения данных на дисках. Несколько дисков могут быть объединены в логический диск или том; таким образом, система может поддерживать файлы, имеющие размеры большие, чем объем одного диска. *Зеркалирование диска* (mirroring) позволяет проводить резервное копирование данных[1], что увеличивает надежность файловой системы. *Расщепленные диски* (stripe sets)[2] позволяют увеличивать пропускную способность системы путем чередования данных в наборе дисков[3]. Использование RAID-устройств совмещает в себе отказоустойчивость и высокую производительность, отвечающие требованиям различных типов установок [11].

8.4. Специальные файлы

Одной из отличительных черт системы UNIX является распространения понятия файла на все объекты, относящиеся к вводу-выводу, в том числе каталоги символических ссылок, аппаратные устройства (диски, терминалы и принтеры), псевдоустройства (системная память) а также элементы коммуникаций (каналы и сокеты). Доступ к каждому из перечисленных объектов производится путем обращения к дескриптору файла. Действия над специальными файлами осуществляются при помощи тех же системных вызовов, что и для работы с обычными файлами. Например, пользователь может послать данные на принтер простым открытием файла, ассоциированным с указанным принтером, предварительно сохранив информацию в этом файле.

С другой стороны, система UNIX использует простую байт-ориентированную модель при работе с файлами. Многие приложения требуют более

---

[1] Способ избыточной записи данных, при котором данные одного физического диска автоматически дублируются на другом; в случае присоединения двух дисков к одному контроллеру речь уже идет об удвоении дисков (disk duplexing). — *Прим. ред.*

[2] Разделы, распределенные по нескольким дискам. — *Прим. ред.*

[3] Надежность всей системы ниже надежности составляющих ее дисков. — *Прим. ред.*

сложных методик, таких как доступ к записям или доступ, основанный на индексации. Для таких приложений модель UNIX является недостаточной, и поэтому разработчики создают свои собственные методы доступа к файлам на базе этой модели. Некоторые объекты ввода-вывода не поддерживают всех файловых операций. Например, невозможно производить произвольный доступ или позиционирование по отношению к терминалам и принтерам. Часто приложения нуждаются в проверке, к файлу какого типа они производят доступ (обычно путем вызова fstat).

8.4.1. Символические ссылки

В ранних версиях UNIX, таких как SVR3 или 4.1BSD, поддерживались только жесткие ссылки. Хотя такие ссылки очень удобны, они обладают некоторыми недостатками. Жесткая ссылка не может пересечь границы файловой системы. Создание ссылок на каталоги часто разрешается только суперпользователю, который не особо «горит» желанием это делать. Дело в том, что невнимательно созданные ссылки способны повлечь создание замкнутой цепи в дереве каталогов, что приведет к возникновению проблем у многих утилит, таких как du или find, просматривающих дерево каталогов рекурсивно.

Единственной причиной, по которой система UNIX разрешает пользователям (привилегированным) создавать жесткие ссылки на каталоги, является отсутствие вызова mkdir в ранних версиях ОС (SVR2 и менее). Для создания каталога пользователь должен был вызывать функцию mknod, которая создавала специальный файл каталога, после чего двукратно вызывался link для добавления «.» и «..». Это приводило к появлению некоторых проблем, а также состязательности, поскольку три вышеперечисленные операции не являются неделимыми [1]. Вызов mkdir (и rmdir) появился в системе SVR3, но поддержка жестких связей с каталогами осталась в ОС для обратной совместимости со старыми приложениями.

При применении жестких ссылок также имеют место проблемы управления. Представьте, что пользователь X является владельцем файла с именем /usr/X/file1. Другой пользователь (Y) может создать на этот файл жесткую ссылку и вызывать ее как /usr/Y/link1 (см. рис. 8.6). Для этого пользователю Y достаточно иметь права на выполнение в каталоги, присутствующие в полном имени, и права на запись в каталог /usr/Y. Через некоторое время пользователь X вправе произвести удаление ссылки на file1, после чего считать, что файл был удален (обычно пользователи не проверяют счетчики ссылок на свои файлы). Однако файл продолжает существовать, так как на него имеется еще одна ссылка.

Конечно, владельцем ссылки /usr/Y/link1 является пользователь X, хотя она была создана Y. Если пользователь X защитит файл от записи, пользователь Y не сможет произвести в нем изменения. Более того, пользователю X не

обязательно желать, чтобы файл существовал постоянно. В системах, где использование диска лимитируется квотами, файл будет ограничивать доступное пространство пользователя X. Пользователь X не имеет возможности узнать местонахождение ссылки, если пользователь Y защитил каталог /usr/Y от чтения (или если X не знает номер индексного дескриптора файла).

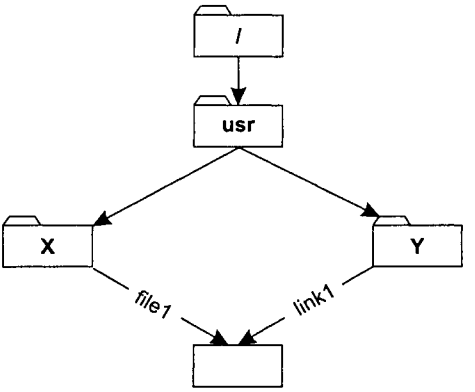


Рис. 8.6. Жесткие ссылки на файл

В системе 4.2BSD были впервые представлены символические ссылки, которые могут быть как абсолютными, так и относительными. Позже это новое средство было принято большинством разработчиков ОС, а также добавлено в файловую систему s5fs SVR4. Для создания символической ссылки служит вызов symlink. Ссылка — это специальный файл, указывающий на другой файл (или *связываемый* с файлом). Для идентификации файла как символической ссылки используются ее атрибуты. Внутри такого файла содержатся данные, показывающие путь к связываемому файлу. Многие системы позволяют сохранять короткие имена путей к ссылкам внутри их индексного дескриптора. Впервые такая возможность появилась в ULTRIX.

Полное имя символьной ссылки может быть как абсолютным, так и относительным. Процедуры преобразования путей опознают символьные ссылки и преобразуют их с целью получения имени связанного с ними файла. Если используется относительное имя, то оно интерпретируется в зависимости от каталога, в котором находится ссылка. Хотя символические ссылки обрабатываются незаметно для большинства программ, некоторые утилиты должны обладать возможностью их обнаружения и обработки. Это можно сделать при помощи системного вызова lstat, который запрещает преобразование последней символьной ссылки на полное имя. Таким образом, если ссылка mylink указывает на файл myfile, то вызов lstat(mylink, ...) возвратит атрибуты mylink, а вызов stat(mylink, ...) покажет атрибуты myfile. Имея возможность обнаружения ссылок при помощи lstat, пользователь в состоянии просмотреть содержимое ссылки, вызвав функцию readlink.

8.4.2. Каналы и файлы FIFO

Каналы и файлы FIFO — это элементы системы, представляющие потоки, функционирующие по принципу «первым вошел, первым вышел». Их основные отличия между собой заключаются в методе создания. Файл FIFO создается путем вызова mknod, после чего он может открываться и использоваться любым процессом, знающим его имя и обладающим соответствующими полномочиями. Он продолжает существовать до тех пор, пока не будет удален принудительно. Канал создается вызовом pipe, который возвращает дескрипторы чтения и записи. Процесс, создавший канал, имеет возможность передавать эти дескрипторы своим потомкам через вызов fork, таким образом разделяя между ними канал. У одного канала может быть несколько получателей и отправителей данных.[1] После того как у канала не остается ни одного активного читающего или пишущего процесса, ядро автоматически удаляет его.

Операции ввода-вывода для файлов FIFO и канал очень схожи. Одни процессы добавляют данные в конец элемента, в то время как процессы-получатели извлекают данные из его начала. После прочтения данных происходит их удаление из конвейера, после чего канал оказывается недоступным даже для других получателей. Ядром определен параметр PIPE_BUF (равный по умолчанию 5 120 байт), который ограничивает объем данных, хранящихся внутри конвейера. Если запись информации приводит к переполнению файла FIFO или канала, читающий процесс будет заблокирован до тех пор, пока из элемента не будет удалено достаточное количество данных (при помощи их считывания). Если процесс попытается за один прием произвести запись данных большего объема, чем значение PIPE_BUF, ядро не сумеет гарантировать неделимость операции записи. В этом случае данные могут оказаться перемешаны с информацией, записанной другими процессами.

Чтение из файла FIFO отличается немногим. Если запрашиваемый размер данных оказывается большим, чем текущий объем элемента, ядро прочтет тот объем информации, который содержится в нем, и возвратит процессу количество считанных байтов. Если элемент в текущий момент не содержит данных, читающий процесс будет заблокирован до тех пор, пока другой процесс не произведет запись в канал. Флаг O_NDELAY (устанавливаемый через вызов open для файлов FIFO или fcntl) изменяет режим работы канала или файла FIFO на неблокирующий. В этом случае чтение и запись завершатся в любом случае без приостановки работы процессов, при этом будет передано максимально возможное количество данных.

Канал поддерживают счетчики количества текущих читающих и пишущих процессов. После того как последний пишущий закроет канал, ядро системы разбудит все ожидающие процессы, которые смогут прочесть данные, оставшиеся в канале. После освобождения канала они получат при очередном вызове read значение 0, что будет интерпретировано ими как конец файла. После

[1] Разделять доступ к каналу могут только те потомки, которые были созданы уже после его появления.

того как последний процесс-получатель закроет канал, ядро пошлет сигнал SIGPIPE всем заблокированным процессам-отправителям. Все последующие вызовы write для этого канала возвратятся с ошибкой EPIPE.

Изначальная реализация каналов использовала файловую систему. Каждому каналу присваивался индексный дескриптор и список блокировки. Во многих вариантах системы BSD для реализации каналов были использованы сокеты (см. раздел 17.10.3). В ОС SVR4 каналы и файлы FIFO построены на STREAMS. Более подробно о них вы прочтете в разделе 17.9. Каналы в SVR4 являются двунаправленными. Они поддерживают отдельные потоки данных, передаваемые в каждом направлении. Такой вариант более подходит для большинства приложений, требующих полнодуплексного механизма взаимодействия процессов.

8.5. Базовые файловые системы

Традиционное ядро UNIX имеет монолитную файловую организацию и может использовать только одну файловую систему — s5fs. Система FFS от разработчиков 4.2BSD предоставила производителям ОС альтернативу. Однако поддерживаемая в те времена базовая структура операционных систем не позволяла s5fs и FFS функционировать вместе. Большинство производителей остановило свой выбор на FFS, но некоторые из них поддерживали s5fs из соображений обратной совместимости. В любом случае, ни одно из этих решений не являлось удовлетворительным.

Более того, обе файловые системы рассчитаны на общие приложения разделения времени, поэтому многие разработчики программ нашли их недостаточно подходящими для своих нужд. Например, приложениям работы с базами данных нужна более совершенная поддержка обработки транзакций. Приложениям, использующим объемные файлы (как правило, только считываемые), необходимо предоставить средство выделения программе непрерывных участков памяти, увеличивающее производительность операций последовательного чтения. Первые версии UNIX не поддерживали добавление новых файловых систем независимыми производителями. Для этого требовалось вносить большой объем изменений в ядро. Это ограничение определяло операционную систему UNIX как базовую платформу для достаточно большого диапазона различных типов приложений.

С течением времени все больше увеличивалась необходимость поддержки файловых систем других ОС. Ее отсутствие ограничивало возможности UNIX, установленной на персональном компьютере, не позволяя иметь доступ к разделам DOS, находящимся на той же машине, или гибких дисков, размеченных под DOS.

Более важен тот факт, что развитие компьютерных систем привело к необходимости разделения файлов между несколькими компьютерами. В середине 80-х годов разработчики увлеклись созданием распределенных файло-

вых систем. В результате появились такие системы, как *Remote File Sharing* (*RFS*) от корпорации AT&T, *Network File System* (*NFS*) от Sun Microsystems и другие, предоставляющие возможность прозрачного доступа к файлам на удаленных узлах.

Развитие компьютерных технологий поставило UNIX перед необходимостью проведения кардинальных изменений в базовой структуре файловой системы с целью поддержки нескольких типов файловых систем одновременно. В результате появились различные архитектуры, такие как *переключение файловых систем* (file system switch) корпорации AT&T [12], *vnode/vfs* компании Sun Microsystems [8] и *gnode* корпорации Digital Equipment Corporation [13]. Долгое время эти технологии боролись между собой за доминирование на рынке операционных систем. Корпорация AT&T добавила технологии vnode/vfs и NFS в операционную систему SVR4, чем положила конец противостоянию, сделав их стандартами де-факто.

Интерфейс vnode/vfs претерпел значительные изменения по сравнению с оригинальной реализацией. Хотя это фундаментальное средство поддерживается основными вариантами UNIX, на практике интерфейс и реализация в каждой ОС отличаются друг от друга. В этой главе наиболее подробно будет описан интерфейс операционной системы SVR4. Краткий рассказ о других реализациях можно найти в разделе 8.11.

8.6. Архитектура vnode/vfs

Интерфейс vnode/vfs разработан компанией Sun Microsystems как базовая структура, поддерживающая различные файловые системы. Описываемая технология получила широкое распространение и стала частью System V Unix в SVR4.

8.6.1. Цели, поставленные перед разработчиками

При разработке архитектуры vnode/vfs выдвигались следующие основные требования:

- необходимость одновременной поддержки несколько различных типов файловых систем, в том числе для UNIX (s5fs или ufs) и других ОС (например, DOS, A/UX и т. д.);
- поддержка содержания в разных разделах одного диска различных типов файловых систем. После монтирования разделы должны быть представлены для пользователя в обычном виде, как однородная файловая система. Пользователь при этом должен видеть дерево файлов полностью и не беспокоиться насчет различий в представлении на дисках каждого поддерева системы;

- система должна обеспечивать разделение файлов по сети. При этом удаленная файловая система обязана быть доступна пользователю точно так же, как и система на локальной машине;
- производители могут создавать собственные типы файловых систем и добавлять их в ядро в качестве модулей.

Основной целью разработки было создание структуры ядра для доступа и действий с файлами, а также четко определенный интерфейс между ядром и остальными модулями, в которых реализованы файловые системы.

8.6.2. Немного о вводе-выводе устройств

Первые версии UNIX имели только один тип файловой системы, но при этом поддерживали различные типы файлов. Кроме обычных файлов система UNIX работает с различными устройствами, доступ к которым организован при помощи специальных файлов устройств. В каждом драйвере имеется собственная реализация процедур ввода-вывода низкого уровня, но при этом интерфейс взаимодействия с пользователем является универсальным, что позволяет осуществлять доступ к файлам устройств точно так же, как и к обычным файлам.

Таким образом, поддержка нескольких файловых систем зависит от базовых конструкций устройств ввода-вывода. Более подробно работа с устройствами ввода-вывода будет показана в разделе 16.3, здесь же мы остановимся только на части интерфейса, относящейся к нашей теме. В системе UNIX все устройства поделены на две категории: блочные и символьные. Существуют небольшие различия в интерфейсе взаимодействия с ядром для каждой категории устройств, но при этом используются одни и те же базовые конструкции.

Система UNIX требует, чтобы каждое символьное устройство поддерживало стандартный набор операций. Такие операции собраны в структуре cdevsw, являющейся совокупностью указателей функций.

```
struct cdevsw {
   int (*d_open)();
   int (*d_close)();
   int (*d_read)();
   int (*d_write)();
   ..
} cdevsw[];
```

То есть cdevsw[] представляет собой глобальный массив структур cdevsw. Он называется переключателем символьных устройств. Каждый тип устройства поддерживает собственный набор функций, реализующих его интерфейс. Например, линейный принтер может иметь функции lpopen(), lpclose() и т. д. Каждое устройство ассоциировано с уникальным *основным номером устройства*. Этот номер используется как индекс в глобальном массиве cdevsw[], предоставляя тем самым каждому устройству собственный элемент в переключателе. Поля каждого элемента указывают на функции, поддерживаемые устройством.

Представим, что пользователь вызывает read по отношению к файлу символьного устройства. В традиционном варианте UNIX обработка этого вызова происходит по алгоритму, описанному ниже.

1. Использовать дескриптор файла для получения объекта открытого файла.
2. Проверить по объекту, не открыт ли файл на чтение.
3. Получить указатель на *индексный дескриптор в оперативной памяти* (in-core inode). Такие дескрипторы (также называемые дескрипторами in-memory) являются структурами данных файловой системы, поддерживающими в памяти атрибуты используемых файлов. Более подробно прочесть о них можно в разделе 9.3.1.
4. Заблокировать индексный дескриптор, тем самым установив последовательный характер доступа к файлу.
5. Проверить поле режима индексного дескриптора с целью обнаружения, что считываемый файл является символьным устройством.
6. Использовать основной номер устройства (хранящийся в индексном дескрипторе) для запроса необходимого элемента таблицы символьных устройств и получения для него элемента структуры cdevsw. Элемент представляет собой массив указателей на функции, реализующие специфические для данного устройства операции.
7. Из структуры cdevsw получить указатель на процедуру d_read для используемого устройства.
8. Вызвать операцию d_read для проведения обработки специфичного для конкретного устройства запроса на чтение. Код при этом выглядит следующим образом:

   ```
   result = (*(cdevsw[major].d_read))(...);
   ```

   где major — основной номер устройства.
9. Разблокировать индексный указатель и произвести возврат функции чтения.

Как мы видим, большинство шагов алгоритма являются независимыми от конкретного устройства. Шаги 1–4 и 9 как правило применяются и для обычных файлов, следовательно, они независимы от типа файлов. Взаимодействие между ядром и устройствами производится на этапах 7–9, при этом необходимые функции извлекаются из таблицы cdevsw. Операции, зависящие от конкретно используемого устройства, производятся только на шаге 8.

Вспомним, что поля таблицы cdevsw, такие как d_read, определяют абстрактный интерфейс. Каждое устройство реализует их при помощи различных функций, например lread() для линейного принтера или ttread() для терминала. Основной номер устройства используется как ключ, при помощи которого общая для всех операция d_read преобразуется в функцию, специфичную для устройства.

Используемые принципы можно расширить и на проблему поддержки нескольких файловых систем. Необходимо разделить коды файловой подсистемы на две части: зависимую от файловой системы и независимую от нее. Интерфейс между ними должен быть определен как набор общих функций, вызываемых независимым от файловой системы кодом для проведения манипуляций с файлами и операций доступа. Участок кода, зависящий от файловой системы, реализует специфические функции. Он неодинаков и пишется для каждого типа файловой системы по отдельности. Базовые конструкции предоставляют механизмы добавления новых файловых систем, а также используются для преобразования абстрактных операций в функции, специфичные для обрабатываемых файлов.

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД (ОТСТУПЛЕНИЕ)

Интерфейс vnode/vfs создан на основе концепций объектно-ориентированного программирования (ООП). Позже такой подход был использован и для других участков ядра UNIX, таких как управление памятью, коммуникации при помощи сообщений и планирование процессов. Ознакомимся вкратце с основами ООП в части применения для разработки ядра систем UNIX. Такие технологии обычно реализуются при помощи объектно-ориентированных языков программирования типа C++, но разработчики UNIX выбрали для этого язык C из соображений единообразия с кодами остальной части ядра.

Объектно-ориентированный подход основан на понятиях класса и объекта. Класс является сложным типом данных, состоящим из полей данных членов класса и набора их функций. Объект — это элемент класса. Функции членов класса оперируют с индивидуальными объектами этого класса. Обобщенно каждый член класса (данные или функция) бывает либо *открытым* (public), либо *закрытым* (private). Внешние пользователи класса видят только общедоступные его элементы. Закрытые данные или методы могут быть доступны только другим функциям того же класса.

На основе любого класса можно создать один или более порожденных классов, называемых классами-наследниками (рис. 8.7). *Класс-наследник* сам по себе может являться базовым для последующих классов, порожденных от него. Так формируется иерархия классов. Наследник приобретает все атрибуты (данные и функции) своего базового класса. Более того, в нем могут быть изменены некоторые функции базового класса, — таким образом, класс-потомок может обладать собственными реализациями этих функций.

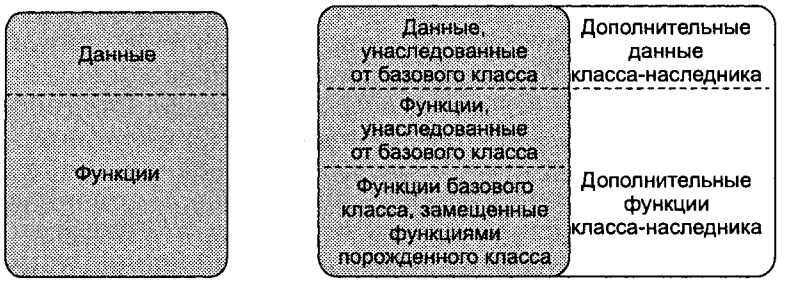


Рис. 8.7. Взаимосвязь между базовым классом и его наследниками

Так как порожденный класс обладает всеми атрибутами базового класса, объект такого типа также является и объектом типа базового класса. Например, класс directory (каталог) может являться порожденным классом file (файла). Это означает, что любой каталог является и файлом. Конечно, обратное утверждение не будет верным: не каждый файл представляет собой каталог. Точно так же указатель на объект каталога является и указателем на объект файла. Атрибуты, добавленные в класс-наследник, невидимы базовым классом. Следовательно, указатель на базовый объект нельзя использовать для доступа к данным или функциям, специфичным для порожденного класса.

Часто базовый класс используется для задания основных абстракций и определения интерфейса. Специфичные реализации функций для каждого члена класса оформляются в порожденных классах. Таким образом, в классе file можно определить функцию create(), но при вызове ее пользователем для произвольного файла загрузятся различные процедуры в зависимости от того, является ли файл обычным, каталогом, символической ссылкой, файлов устройства и т. д. Мы вправе вообще не определять функцию create() для произвольного файла. Такая функция называется *виртуальной функцией*.

Реализовать вышеописанные объекты можно при помощи языков ООП. Например, на C++ можно определить абстрактный базовый класс как класс, содержащий как минимум одну виртуальную функцию. Так как базовый класс не будет иметь реализации для этой функции, она не может быть вызвана. Она используется только для порожденных классов, в которых предлагаются различные реализации виртуальных функций. Все объекты являются представителями одного или другого класса-потомка, но пользователь волен управлять ими при помощи указателя базового класса, не обладая знаниями, к какому конкретному классу-наследнику они относятся. Если для объекта вызывается виртуальная функция, то определение, какую из специфических функций необходимо вызвать, происходит автоматически, в зависимости от класса-наследника данного объекта.

Как уже упоминалось ранее, языки, такие как C++ или SmallTalk, обладают встроенными конструкциями для описания таких понятий, как классы или виртуальные функции. В языке C эти элементы приходится реализовывать самостоятельно. В следующем разделе мы расскажем об интерфейсе vnode/vfs, в основу которого положен объектно-ориентированный подход.

8.6.3. Краткий обзор интерфейса vnode/vfs

В ядре системы UNIX файл представлен абстракцией *vnode* (или virtual node, виртуальным узлом), файловая система представлена понятием *vfs* (virtual file system). Обе абстракции могут быть реализованы как абстрактный базовый класс, внутри подклассов которого описаны специфические реализации для различных файловых систем, например s5fs, ufs, NFS или FAT (файловая система MS-DOS).

Класс vnode системы SVR4 показан на рис. 8.8. Поля данных базового класса vnode содержат информацию, не зависящую от определенного типа файловой системы. Функции членов класса могут быть поделены на две категории. В первой категории находится набор *виртуальных функций*, определяющих зависимый от файловой системы интерфейс. В каждой отдельной файловой системе должны иметься собственные реализации для таких функций. Ко второй категории относится набор *высокоуровневых утилит*, используемых другими подсистемами ядра для осуществления действий с файлами. Такие функции вызывают для выполнения задач низкого уровня зависимые от файловой системы процедуры.

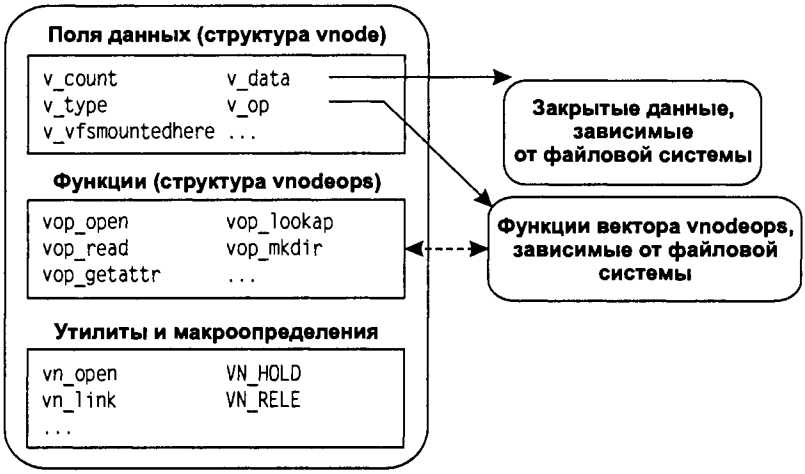


Рис. 8.8. Организация vnode

Базовый класс vnode имеет два поля, используемые для реализации подклассов. Первое поле — v_data, является указателем (типа caddr_t) на закрытую структуру данных, в которой хранятся зависящие от файловой системы данные vnode. Для файлов s5fs или ufs это традиционная структура индексных дескрипторов inode. В системе NSF используется структура rnode, в tmpfs применяется tmpnode и т. д. Так как доступ к этим структурам происходит не напрямую, а при помощи v_data, они непрозрачны для базового класса vnode, а их поля видимы только для функций, являющихся внутренними по отношению к специфичным файловым системам.

Поле v_op указывает на структуру vnodeops, состоящую из набора указателей на функции реализации виртуального интерфейса к vnode. Поля v_data и v_op заполняются при инициализации vnode, обычно на стадии выполнения системных вызовов open или create. Когда участок кода, не зависящий от файловой системы, вызывает виртуальную функцию для произвольного vnode, ядро изменяет указатель v_op и вызывает конкретную функцию, относящуюся к определенной реализации файловой системы. Например, операция VOP_CLOSE позволяет вызвавшему ее процессу закрыть файл, ассоциированный с vnode. Доступ происходит при помощи макроса, такого как приведен ниже:

```
#define VOP_CLOSE(vp, ...) (*((vp)->v_op->vop_close))(vp, ...)
```

где многоточие обозначает другие входные аргументы процедуры close. Приведенный макрос гарантирует, что после соответствующей инициализации vnode применение операции VOP_CLOSE приведет к вызову процедуры ufs_close() для файла ufs, nfs_close() для файла NFS и т. д.

Базовый класс vfs также обладает двумя полями, vfs_data и vfs_op, позволяющими осуществлять взаимосвязь с данными и функциями, реализующими определенные файловые системы. Компоненты vfs показаны на рис. 8.9.

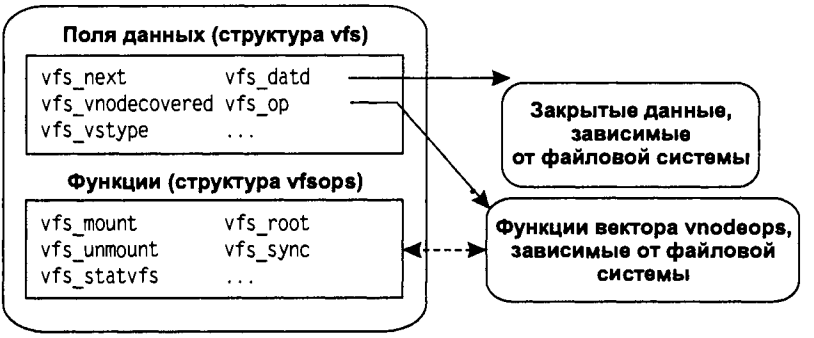


Рис. 8.9. Организация vfs

В языке C базовый класс реализуется как структура, а также набор глобальных функций (и макросов) ядра, определяющих публичные невиртуальные функции. Базовый класс содержит указатель на другую структуру, состоящую из набора указателей на функции, по одному для каждой виртуальной функции. Указатели v_op и v_data (для класса vfs указатели vfs_op и vfs_data) позволяют соединить подклассы, предоставляя таким образом доступ к функциям и данным, зависящим от конкретной файловой системы.

8.7. Краткий обзор реализации

Следующие разделы посвящены более подробному анализу интерфейса vnode/vfs, а также описанию реализации различных файловых операций.

8.7.1. Цели, стоящие перед разработчиками

Для создания гибкого интерфейса, который можно было бы эффективно использовать для работы с различными типами файловых систем, перед разработчиками были поставлены следующие цели:

- каждая операция должна выполняться в зависимости от состояния текущего процесса, который может быть переведен в режим сна, если функции необходимо приостановить работу в ожидании ресурса или события;
- многие операции должны использовать последовательный доступ к файлам. Следовательно, они могут блокировать структуры данных на уровне, зависимом от конкретной файловой системы. Операции обязаны производить освобождение ресурсов перед завершением своей работы;
- интерфейс не должен задействовать информацию о состоянии (stateless). Он не должен использовать глобальные переменные, такие как поля области u, для передачи информации о состоянии между процессами;

- реентерабельность интерфейса. Это требование позволяет не привлекать глобальные переменные, такие как u_error или u_rval1, для хранения кодов ошибок или возвращаемых значений. Фактически все операции передают коды ошибок как возвращаемые значения;
- реализации файловых систем должны уметь задействовать общие ресурсы системы, такие как буферный кэш (но не в принудительном порядке);
- пригодность интерфейса для использования серверной частью удаленной файловой системы с целью обработки клиентских запросов;
- не рекомендуется применение статических таблиц фиксированных размеров. Вместо этого должно применяться динамическое размещение (везде, где это возможно).

8.7.2. Открытые файлы и объекты vnode

Понятие vnode обозначает фундаментальную абстракцию, представляющую используемый файл в ядре системы. Объект vnode определяет интерфейс доступа к файлам и заменяет все операции с ними на функции, специфичные для определенной файловой системы. Существует два способа доступа ядра к vnode. Системный вызов, относящийся к вводу-выводу, может вызвать объект vnode при помощи дескриптора файла (более подробно читайте об этом ниже). Вторым способом является преобразование процедурами полных имен файлов для обнаружения vnode-структур данных, зависящих от конкретной файловой системы. Более подробно о преобразовании полных имен вы узнаете из раздела 8.10.1.

Для того чтобы прочесть файл или произвести запись в него, процесс должен сначала его открыть. Системный вызов open возвращает файловый дескриптор. Такой дескриптор, являющийся обычно целым числом, применяется как идентификатор файла и соответствует независимому сеансу, или потоку, для этого файла. В дальнейшем процесс передает полученный дескриптор вызовам read или write.

На рис. 8.10 показаны основные структуры данных. Файловый дескриптор является объектом процесса, содержащим указатель на *объект открытого файла* (struct file), а также набор флагов. Поддерживаемыми флагами являются: FCLOSEXEC (для указания ядру на необходимость использования дескриптора при вызове процессом exec) и U_FDLOCK (применяется для блокировки файла).

Объект открытого файла хранит контекст, управляющий сеансом работы с файлом. Если файл открыт несколькими пользователями (либо один пользователь открыл файл несколько раз), то каждый экземпляр будет обладать собственным объектом открытого файла. Поля такого объекта содержат:

- смещение, от которого начнется следующая операция чтения или записи;
- счетчик ссылок, содержащий количество файловых дескрипторов, указывающих на файл. Счетчик обычно равен единице, но значение может быть и выше, если дескрипторы дублированы при помощи dup или fork;

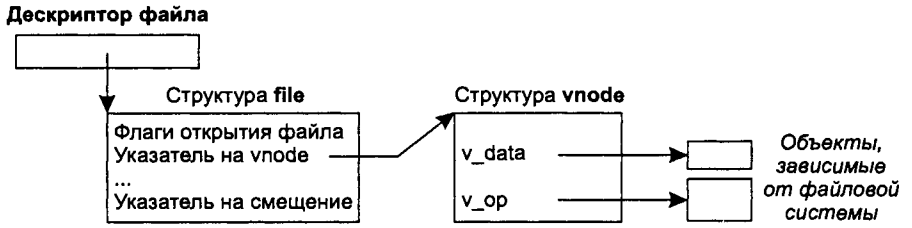


Рис. 8.10. Структуры данных, независимые от файловой системы

- указатель на vnode файла;
- режим, в котором был открыт файл. Ядро проверяет режим при каждой операции ввода-вывода. Таким образом, если пользователь открыл файл только для чтения, он не сможет произвести запись в него, используя дескриптор, даже если пользователь обладает необходимыми полномочиями.

В традиционных системах UNIX применяется статическая таблица дескрипторов фиксированного размера, располагаемая в области u. Размер таблицы (обычно это 64 элемента) ограничивает количество файлов, открытых одним пользователем. В современных вариантах UNIX таблица дескрипторов не имеет ограничений по размеру и может вырасти до огромных размеров[1].

В некоторых реализациях ОС, например в SVR4 или SunOS, дескрипторы размещаются в блоках (обычно) по 32 элемента. Блоки хранятся в связанном списке, первый блок находится в области u процесса. Это усложняет задачу удержания дескриптора. Вместо использования его в качества индекса к таблице ядру приходится сначала находить нужный блок и затем производить к нему запрос. При незначительном увеличении сложности кода такая схема позволяет снять ограничение на количество файлов, открываемых одним процессом.

В современных операционных системах, основанных на SVR4, таблица дескрипторов размещается динамически и расширяется по мере необходимости при помощи вызова kmem_alloc(), производящего либо расширение существующей таблицы, либо копирование ее в новое место, имеющее свободное пространство для увеличения объема таблицы. Этот способ позволяет увеличивать таблицу дескрипторов динамически и производить их быстрое преобразование, что достигается путем копирования таблицы при ее размещении.

8.7.3. Структура vnode

Объект vnode представлен структурой данных, приведенной ниже:

```
struct vnode {
  u_short v_flag; /* флаги V_ROOT и т. д. */
```

[1] Размер таблицы ограничен RLIMIT_NOFILE.

```
    u_short v_count; /* счетчик ссылок */
    struct vfs *vfsmountedhere; /* для точек монтирования */
    struct vnodeops *v_op; /* вектор операций vnode */
    struct vfs *vfsp; /* файловая система, к которой относится
       структура */
    struct stdata *v_stream; /* указатель на ассоциированный поток,
       если таковой существует */
    struct page *v_page; /* резидентный список страниц */
    enum vtype v_type; /* тип файла */
    dev_t v_rdev; /* идентификатор устройства для файлов устройств */
    caddr_t v_data; /* указатель на закрытую структуру данных */
    ...
};
```

Отдельные поля структуры более подробно будут разобраны в следующих разделах главы.

8.7.4. Счетчик ссылок vnode

Поле v_count структуры vnode содержит счетчик ссылок, при помощи которого определяется, как долго структура будет поддерживаться ядром. Структура vnode размещается и ассоциируется с файлом при первом доступе к нему. Позже указатели (или ссылки) на vnode могут поддерживать и другие объекты, желающие получить доступ к этому файлу при помощи таких указателей. Это означает, что ядро будет поддерживать структуру vnode и не сможет произвести ассоциацию ее с другим файлом до тех пор, пока существуют такие ссылки.

Счетчик ссылок является одним из общих свойств структуры vnode. Манипуляции с ним производятся фрагментом кода, независящим от конкретной файловой системы. Для инкремента и декремента счетчика используются макросы VN_HOLD и VN_RELE. Если значение счетчика станет равным нулю, это будет означать, что файл неактивен и его структура vnode может быть освобождена или передана другому файлу.

Важно понимать разницу между получением ссылки (или удержанием) и блокировкой. Блокировка объекта защищает его от доступа определенным образом, зависящим от того, является ли блокировка эксклюзивной или позволяющей чтение-запись. Удержание ссылки на объект дает гарантию постоянства этого объекта. Код, зависящий от файловой системы, блокирует структуру vnode на короткие промежутки времени, обычно равные длительности одной операции с нею. Ссылка, как правило, удерживается в течение большого временного интервала, занимаемого не несколькими операциями над vnode, а несколькими системными вызовами. Ниже приведены примеры операций, при которых происходит запрос ссылки на vnode.

- Запрос ссылки на vnode (то есть инкремент счетчика ссылок) происходит при открытии файла. Закрытие файла приводит к освобождению ссылки (или декременту счетчика).
- Процесс удерживает ссылку на текущий рабочий каталог. Если процесс изменяет свой рабочий каталог, то он запрашивает на новый каталог ссылку и освобождает ссылку на предыдущий.
- При монтировании новой файловой системы запрашивается ссылка на каталог, являющийся точкой монтирования. Освобождение такой ссылки происходит при размонтировании системы.
- Процедуры преобразования полных имен запрашивают ссылки на каждый промежуточный каталог, с которым они сталкиваются. Удержание ссылки происходит в течение поиска каталога, освобождение — после получения ссылки на следующий компонент полного имени.

Счетчики ссылок гарантируют постоянство vnode, а также файла, с которым эта структура была ассоциирована. Если один процесс удаляет файл, открытый другим процессом (или тем же процессом дважды), то такой файл не удалится с диска физически. Будет удален элемент каталога для такого файла, что не даст возможности другим процессам открывать его. Файл же продолжает при этом существовать до тех пор, пока счетчик его структуры vnode не станет равным нулю. Процесс, обладающий в текущий момент этим открытым файлом, будет иметь доступ к нему до того момента, пока сам не закроет файл. Такой подход эквивалентен *маркировке файла для удаления*. После освобождения последней ссылки независимый от файловой системы код вызовет операцию VOP_INACTIVE для завершения удаления файла. Например, для файлов ufs или s5fs при этом производится освобождение индексного дескриптора и блоков данных.

Описанная возможность очень полезна для создания временных файлов. Приложения, такие как компиляторы, используют временные файлы для хранения результатов промежуточных фаз периода выполнения. Такие файлы должны удаляться при ненормальном завершении работы приложения. Приложение следит за этим посредством открытия файла и последующего его немедленного отсоединения. Тогда счетчик ссылок становится равным нулю, и ядро удаляет элемент каталога. Это защищает от возможности видеть такие файлы и иметь доступ к ним другим пользователям. Пока файл открыт, счетчик, содержащийся в памяти, равен единице. Пока файл существует, приложение может производить чтение или запись в него. При закрытии файла, независимо от того, случилось ли это принудительно, либо произвольно при завершении работы, счетчик ссылок обнуляется. Тогда ядро завершает процесс удаления и освобождает блоки данных и индексный дескриптор. Многие версии UNIX поддерживают стандартную библиотечную функцию tmpfile, создающую временный файл.

8.7.5. Объект vfs

Объект vfs (структура vfs) соответствует файловой системе. Ядро размещает по одному объекту vfs для каждой активной файловой системы. Структура данных vfs имеет следующий вид:

```
struct vfs {
   struct vfs vfs_next;/* следующая структура vfs в списке */
   struct vfsops *vfs_op; /* вектор операций */
   struct vnode *vfs_vnodecovered; /* структура vnode, монтированная на
      этой файловой системе */
   int vfs_fstype; /* индекс типа файловой системы */
   caddr_t vfs_data; /* закрытые данные */
   dev_t vfs_dev; /* идентификатор устройства */
...
};
```

На рис. 8.11 продемонстрированы взаимосвязи между объектами vnode и vfs для ОС, содержащей две файловые системы. Вторая файловая система монтирована в каталоге /usr корневой файловой системы. Глобальная переменная rootvfs указывает на заголовок связанного списка всех объектов vfs. Объект vfs для корневой файловой системы находится в заголовке списка. Поле vfs_vnodecovered указывает на структуру vnode, с которой было произведено связывание файловой системы.

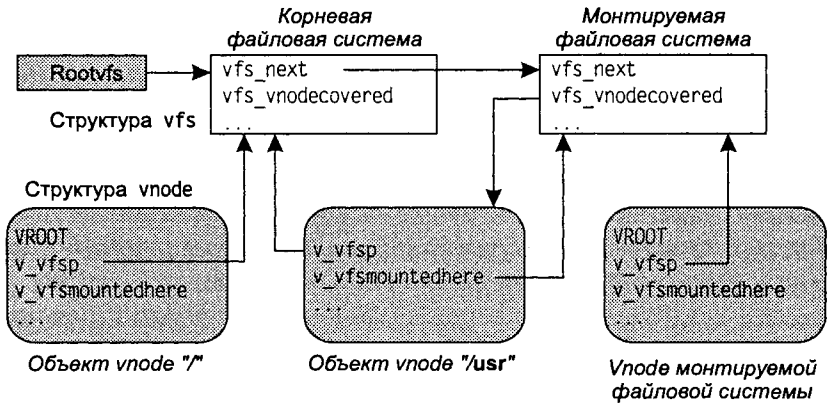


Рис. 8.11. Взаимосвязи между объектами vnode и vfs

Поле v_vfsp каждой структуры vnode указывает на объект vfs, к которому он относится. Корневые объекты vnode каждой файловой системы имеют установленный флаг VROOT. Если объект vnode является точкой монтирования, то его поле v_vfsmountedhere ссылается на объект vfs файловой системы, на которую он смонтирован. Помните о том, что корневая файловая система никуда не монтируется и не покрывается какой-либо структурой vnode.

8.8. Объекты, зависящие от файловой системы

В этом разделе вы прочтете описание объектов интерфейса vnode/vfs, зависящих от файловой системы, а также реализации доступа к этим объектам операциями уровня, независимого от файловой системы.

8.8.1. Закрытые данные каждого файла

Объект vnode является абстрактным. Он не может существовать в изоляции, новый экземпляр vnode всегда в контексте конкретного файла. Файловая система, к которой относится конкретный файл, предлагает собственную реализацию интерфейса абстрактного объекта vnode. Поля данных v_op и v_data связывают его с зависимой от файловой системы частью кода. Поле v_data указывает на закрытую структуру данных, хранящую информацию о файле, зависящую от файловой системы. Структуры данных используются по-разному в зависимости от конкретной файловой системы, к которой относится обрабатываемый файл. Например, файлы s5fs или ufs используют структуры индексных дескрипторов[1], в NFS применяется rnodes и т. д.

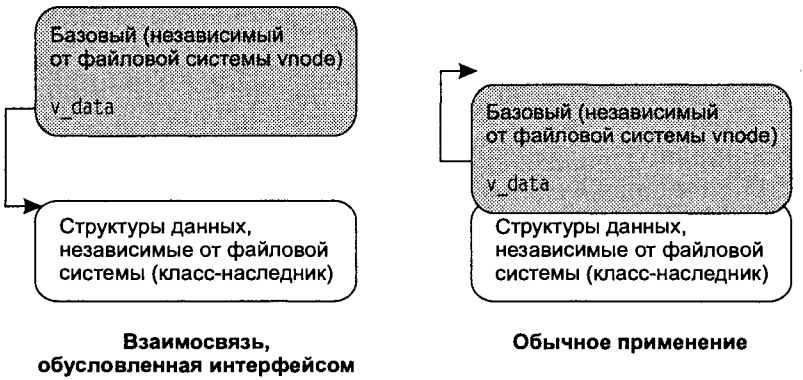


Рис. 8.12. Интерфейс vnode, зависящий от файловой системы

Поле v_data является непрозрачным указателем. Это означает, что участок кода, не зависящий от файловой системы, не может иметь прямого доступа к объекту, зависимому от нее. Однако зависящий от системы код вправе иметь доступ к базовому объекту vnode и использует эту возможность. В любом случае необходим способ нахождения vnode через закрытый объект данных. Так как оба объекта всегда размещаются вместе, более эффективно скомбинировать их в одном. В реализации ссылок на уровне vnode объект vnode

[1] Структуры индексных дескрипторов для этих двух файловых систем значительно отличаются друг от друга.

просто-напросто является частью объекта, зависящего от файловой системы. Заметим, что описанный метод является превалирующим в существующих реализациях файловых систем. Он подходит для поддержания отдельных структур данных vnode и участка кода, зависящего от файловой системы, если произвести соответствующим образом инициализацию поля v_data. Эти взаимосвязи показаны на рис. 8.12.

8.8.2. Вектор vnodeops

Интерфейс vnode определяет набор операций над типовым файлом. Независимый от файловой системы код производит манипуляции с файлом, используя только эти операции. Он не может получить прямой доступ к объектам, зависящим от системы. Структура vnodeops, используемая для определения набора операций, описана ниже.

```
struct vnodeops {
   int (*vop_open)();
   int (*vop_close)();
   int (*vop_read)();
   int (*vop_write)();
   int (*vop_ioctl)();
   int (*vop_getattr)();
   int (*vop_setattr)();
   int (*vop_access)();
   int (*vop_lookup)();
   int (*vop_create)();
   int (*vop_remove)();
   int (*vop_link)();
   int (*vop_rename)();
   int (*vop_mkdir)();
   int (*vop_rmdir)();
   int (*vop_readdir)();
   int (*vop_symlink)();
   int (*vop_readlink)();
   int (*vop_inactive)();
   int (*vop_rwlock)();
   int (*vop_rwunlock)();
   int (*vop_realvp)();
   int (*vop_getpage)();
   int (*vop_putpage)();
   int (*vop_map)();
   int (*vop_poll)();
   ...
};
```

В каждой файловой системе приведенный интерфейс реализован по-своему. Файловая система предоставляет набор функций для операций, указан-

ных в нем. Например, в файловой системе ufs операция VOP_READ реализована как чтение файла с локального диска, в то время как в NFS эта команда производит запрос к удаленному файл-серверу для получения данных. Таким образом, каждая файловая система поддерживает собственный экземпляр структуры vnodeops. Например, в системе ufs определен объект

```
struct vnodeops ufs_vnodeops = {
   ufs_open;
   ufs_close;
   ...
};
```

Поле v_op структуры vnode указывает на структуру vnodeops для типа файловой системы, ассоциированной с файлом. Как показано на рис. 8.13, все файлы одной файловой системы разделяют между собой единственный экземпляр структуры и обладают доступом к одному и тому же набору функций.

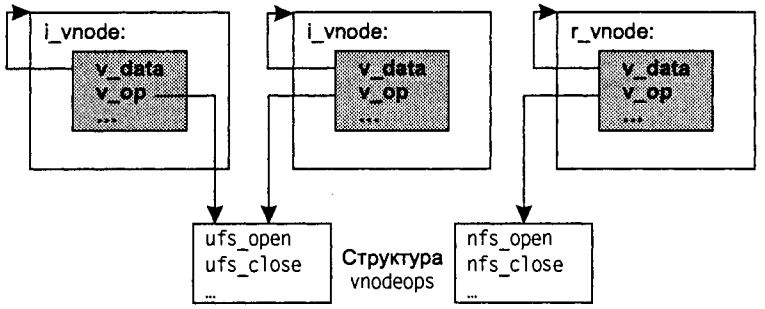


Рис. 8.13. Объекты vnode, зависящие от файловой системы

8.8.3. Зависящая от файловой системы часть уровня vfs

Как и vnode, объект vfs имеет указатели на свои закрытые данные и на вектор операций. Поле vfs_data указывает на непрозрачную структуру данных файловой системы. В отличие от vnode сам объект vfs и его закрытая структура данных обычно размещаются отдельно. Поле vfs_op указывает на структуру vfsops, описываемую следующим образом:

```
struct vfsops {
   int (*vfs_mount)();
   int (*vfs_unmount)();
   int (*vfs_root)();
   int (*vfs_statvfs)();
   int (*vfs_sync)();
...
};
```

Каждый тип файловой системы имеет собственную реализацию перечисленных функций. Таким образом, используется один экземпляр структуры vfsops для каждого типа файловой системы: ufs_vfsops для ufs, nfs_vfsops для NFS и т. д. На рис. 8.14 показаны структуры данных уровня vfs для системы, содержащей две файловые системы ufs и одну NFS.

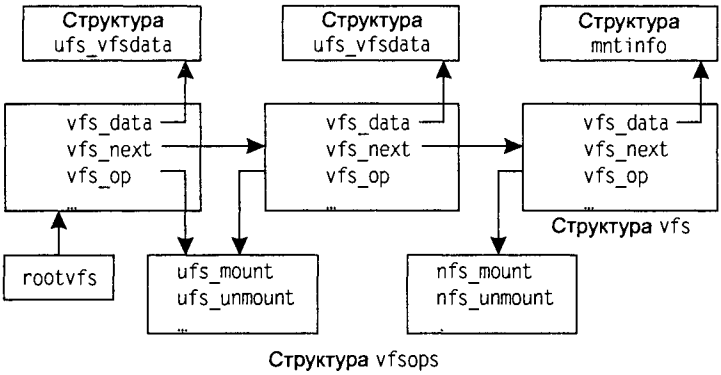


Рис. 8.14. Пример структур данных уровня vfs

8.9. Монтирование файловой системы

Разработчики интерфейса vfs изменили системный вызов mount для поддержки существования нескольких файловых систем одновременно. Синтаксис вызова mount в SVR4 выглядит следующим образом:

```
mount (spec, dir, flags, type, dataptr, datalen);
```

где spec является именем файла устройства, представляющим файловую систему, dir — задает полное имя каталога, являющегося точкой монтирования, type — строка, определяющая типа файловой системы, dataptr — указатель на дополнительные аргументы, зависящие от файловой системы, а datalen является общей длиной блока этих дополнительных параметров. В данном разделе мы опишем, как системный вызов mount реализуется ядром.

8.9.1. Виртуальный переключатель файловых систем

Если в системе сконфигурированы файловые системы различных типов, то встает вопрос корректного соединения vnode и vfs в таких реализациях. Ядру необходим некий механизм, подсказывающий ему, как правильно осуществлять доступ к интерфейсным функциям каждой файловой системы.

В ОС SVR4 для этой цели используется технология, получившая название виртуального переключателя файловых систем, являющегося глобальной таб-

лицей, в которой содержится по одному элементу для каждого типа файловой системы. Элементы таблицы описываются при помощи структуры vfssw.

```
struct vfssw {
   char vfs_name; /* имя типа файловой системы */
   int (*vfs_init)(); /* адрес процедуры инициализации */
   struct vfsops *vfs_vfsops; /* вектор операций vfs для этой файловой
      системы */
   ...
} vfssw[]
```

8.9.2. Реализация вызова mount

Системный вызов mount запрашивает объект vnode для каталога, являющегося точкой монтирования при помощи функции lookuppn(). Вызов удостоверяется, что структура vnode представляет каталог, и никакая другая файловая система не была раньше смонтирована на этот каталог. (Заметим, что функция lookuppn() запрашивает ссылку на каталог, которая освобождается только после размонтирования системы.) Затем происходит поиск в таблице vfssw[] на предмет обнаружения элемента, совпадающего с именем type.

После обнаружения необходимого элемента переключателя ядро загружает для него операцию vsw_init. Этот вызов является операцией, зависящей от конкретной используемой файловой системы. Он производит размещение структур данных и ресурсов, необходимых для функционирования файловой системы. Затем ядро размещает новую структуру vfs и инициализирует ее.

1. Добавляет структуру в связанный список с головным элементом rootvfs.
2. Задает в поле vfs_op ссылку на вектор vfsops, указанный в элементе переключателя.
3. Задает в поле vfs_vnodecovered ссылку на vnode каталога, являющегося точкой монтирования.

Затем ядро сохраняет указатель на структуру vfs в поле v_vfsmountedhere структуры vnode покрываемого каталога. Процесс монтирования завершается вызовом VFS_MOUNT применительно к vfs, которая выполняет часть действий вызова mount, зависящих от типа подключаемой файловой системы.

8.9.3. Действия операции VFS_MOUNT

Каждая файловая система обладает собственной реализацией операции VFS_MOUNT. Эта функция должна выполнять следующие операции:

- проведение проверки привилегий на операцию;
- размещение и инициализация закрытого объекта данных файловой системы;
- сохранение указателя на него в поле vfs_data объекта vfs;

- доступ к корневому каталогу файловой системы и инициализация его структуры vnode в памяти. Единственным способом доступа ядра к корню монтированной файловой системы является применение операции VFS_ROOT. Часть vfs, зависящая от типа файловой системы, должна поддерживать информацию, необходимую для обнаружения корневого каталога.

Локальные файловые системы обычно реализуют VFS_MOUNT путем чтения метаданных файловой системы (таких как *суперблок* для s5fs) с диска, в то время как распределенные варианты систем могут посылать запрос на монтирование файл-серверу.

8.10. Операции над файлами

В этом разделе будет рассмотрена реализация некоторых файловых операций в архитектуре vnode/vfs. Вы увидите описание системных вызовов open и read, а начнем мы разговор с методики преобразования полных имен файлов.

8.10.1. Преобразование полных имен

Независимая от файловой системы функция lookuppn() преобразует полное имя файла и возвращает указатель на vnode выбранного файла. Функция также запрашивает ссылку на объект vnode. Начальная точка поиска зависит от того, является ли полное имя относительным или абсолютным. Если имя – относительное, то вызов lookuppn() приступит к поиску в текущем каталоге, получая указатели vnode из поля u_cdir области u. Если задано абсолютное имя, процесс начнется с корневого каталога. Указатель на vnode для такого каталога является глобальной переменной, имеющей имя rootdir[1].

Функция lookuppn() запрашивает ссылку на начальный объект vnode (увеличивая тем самым для него значение счетчика ссылок) и затем входит в цикл, за каждую итерацию которого происходит просмотр одного полного имени. Цикл включает в себя определенную последовательность шагов.

Проверка, относится ли объект vnode к каталогу (до тех пор, пока не будет достигнут последний компонент). Эта информация содержится в поле v_type структуры vnode.

---

[1] Изменить корневой каталог (а также некоторые другие настройки) процесс может при помощи вызова chroot. Это повлияет на интерпретацию ядром абсолютных имен для процесса. Вызывать chroot умеют большинство сценариев входа в систему в зависимости от того или иного процесса. Эта возможность может быть использована также и администраторами системы, ограничивающими доступ пользователей к дереву каталогов. В этом случае lookuppn() проверяет поле u.u_rdir области u; если оно окажется равным нулю, то следующим шагом станет проверка переменной rootdir.

1. Если найденный компонент окажется «..», а текущий каталог является системным корневым каталогом — переход к следующему компоненту. Корневой каталог является прародителем самого себя.
2. Если найденный компонент окажется «..», а текущий каталог является корнем монтированной файловой системы — получение доступа к каталогу, являющемуся *точкой монтирования*. Флаг VROOT всех корневых каталогов является установленным. Поле v_vfsp указывает на структуру vfs файловой системы, в свою очередь содержащую указатель на точку монтирования в поле vfs_vnoderecovered.

   Вызов для этого объекта vnode операции VOP_LOOKUP. В результате произойдет вызов функции lookup для данной файловой системы (например, s5lookup(), ufs_lookup() и т. д.). Функция ищет в каталоге компонент и, если находит таковой, возвращает указатель на vnode этого файла (размещая структуру в ядре, если это не было сделано ранее). Функция также запрашивает ссылку на объект vnode.
3. Если компонент не найден, проверка, не является ли он последним. Если так, возврат с удачным кодом завершения (то есть вызывающий может создать файл), а также возврат указателя на родительский каталог без освобождения ссылки на него. В противоположном случае завершение выполнения с ошибкой ENOENT.
4. Если новый компонент является точкой монтирования (поле v_vfs_mountedhere != NULL), следование за указателем на объект vfs монтированной файловой системы и вызов vfs_root для возврата *корневого объекта vnode* файловой системы.
5. Если новый компонент является символической ссылкой (v_type == VLNK), вызывается VOP_SYMLINK для преобразования символической ссылки. Добавление оставшейся части полного имени в содержимое ссылки и начало новой итерации. Если ссылка содержит абсолютное имя, процедура должна продолжиться с корневого системного каталога.

   Вызывающий lookuppn() может передать функции флаг, запрещающий определение символической ссылки для последнего компонента полного имени. Эта возможность используется для некоторых системных вызовов, например, lstat, которым не нужно просматривать символические ссылки в конце полных имен. Общее количество символических ссылок, просматриваемых функцией lookuppn(), ограничивается глобальной переменной MAXSYMLINKS (по умолчанию равной 20). Переменная защищает от зависания в бесконечном цикле при неверно заданных символических ссылках, например в том случае, когда /x/y является ссылкой на /x.
6. Освобождение каталога, в котором только что был завершен поиск. Ссылка на него была запрошена при проведении операции VOP_LOOKUP. С начальной точки ссылка в принудительном порядке была получена функцией lookuppn().

7. В завершение — переход в начало цикла и продолжение поиска со следующего компонента каталога, представленного уже новым объектом vnode.
8. Завершение выполнения, если компонентов больше не осталось. Если поиск оказался удачным, не освобождать ссылку на последнем объекте vnode и вернуть указатель на него процессу, инициировавшему процедуру поиска.

8.10.2. Кэш просмотра каталогов

Кэш подстановки имен файлов — это основополагающий ресурс ОС, доступный любым реализациям файловых систем, желающим воспользоваться им. Он представляет собой кэш объектов (построенный по принципу LRU, то есть в порядке «последнего недавно использовавшегося»), содержащих указатель на vnode каталога, имени файла в этом каталоге, а также указатель на vnode для этого файла. Кэш производит быстрое установление местонахождения элементов путем объединения их в хэш-таблице, основываясь на имени текущего файла и каталога.

Если файловая система применяет кэш подстановки имен, то функция `lookup` (реализующая операцию `VOP_LOOKUP`) станет при поиске нужного имени файла сначала просматривать кэш. Если такой файл будет обнаружен в нем, функция просто увеличит счетчик ссылок на его объекте vnode и завершит выполнение. Такой подход дает возможность найти объект без полномасштабного поиска в каталогах, не производя операции чтения с диска. Применение кэша является удобным, так как многие разработчики программ чаще всего производят одни и те же запросы по отношению к одним и тем же файлам и каталогам. Если компонент не будет найден в кэше, функция продолжит поиск в каталогах диска. После обнаружения компонента на диске функция добавит его в кэш для дальнейшего использования.

Поскольку файловая система может получить доступ к vnode не только при просмотре каталогов, то некоторые из ее действий способны привести к тому, что данные элемента кэша станут неверными. Например, пользователь может отсоединить файл, а ядро, в свою очередь, передать его vnode новому файлу. Если затем произвести поиск старого файла, результатом станет запрос неверного элемента кэша и получение указателя на vnode совершенно другого объекта. Таким образом, становится очевидным, что кэш имен должен обладать некоей методикой проверки правильности вхождений. Кэш подстановки имен реализован как в системе 4.3BSD, так и в ОС SVR4, однако в каждой из них применяются различные средства решения описанной проблемы.

В системе 4.3BSD не используется интерфейс vnode/vfs. Кэш просмотра каталогов хранится в индексном дескрипторе файла, находящемся в памяти. Каждый индексный дескриптор представляет собой сгенерированный номер, также называемый *характеристикой* (capability), увеличивающийся каждый

раз при назначении дескриптора новому файлу. Кэш основан на рекомендациях. При попытке добавления в него файловая система производит копирование генерируемого номера индексного дескриптора файла в этот элемент. Функция обработки кэша сверяет полученный номер с текущим сгенерированным номером индексного дескриптора. Если они не совпадают, это означает, что данный элемент кэша является неверным.

В системе SVR4 каждый элемент кэша удерживает ссылку на vnode кэшируемого файла и освобождает ее, если запись исключается или переназначается. Хотя описанный метод и дает гарантию правильности соответствия записей кэша, он обладает некоторыми недостатками. К примеру, ядро должно поддерживать некоторые неактивные объекты vnode только потому, что существуют элементы кэша, обладающие ссылками на них. Применяемый метод также защищает другие компоненты ядра от получения эксклюзивного права на использование файла или устройства.

8.10.3. Операция VOP_LOOKUP

Операция VOP_LOOKUP представляет собой интерфейс доступа к специфичной для файловой системы функции поиска компонента имени файла в каталоге. Функция вызывается при помощи макроса, подобного приведенному ниже.

```
error = VOP_LOOKUP (vp, compname, &tvp, ...);
```

где vp — указатель на родительский каталог vnode, compname — компонент имени. В случае удачного завершения переменная tvp должна указывать на объект vnode компонента compname, а его счетчик ссылок окажется увеличенным на единицу.

Как и другие операции интерфейса vnode/vfs, применение VOP_LOOKUP приведет к вызову функции lookup, специфичной для используемой файловой системы. Функция поиска обычно начинает просмотр с кэша имен. Если она находит необходимое вхождение, то следующим шагом увеличивает значение счетчика ссылок и возвращает указатель на vnode искомого компонента. Если в кэше нет данных об объекте поиска, функция продолжит просмотр в родительском каталоге. Локальные файловые системы производят такой поиск путем просмотра элементов каталогов в цикле блок за блоком. В распределенных файловых системах запрос на поиск посылается серверному узлу.

Если каталог содержит искомый компонент, функция поиска проверит, не находится ли его объект vnode уже в памяти. В каждой файловой системе применяется собственный метод отслеживания объектов, расположенных в памяти. Например, в ufs результатом поиска каталога станет номер индексного дескриптора, который в дальнейшем используется системой как индекс в таблице хэширования, по которому и ведется поиск индексного дескриптора. Дескриптор, хранящийся в памяти, содержит в себе vnode. Если функция поиска обнаруживает объект vnode в памяти, то следующим шагом она производит инкремент его счетчика ссылок и возвращает его вызвавшему процессу.

Часто возникает ситуация, когда процедура поиска находит соответствие компоненту, но его объект vnode не находится в памяти. В этом случае она должна запросить и проинициализировать vnode, точно так же, как и закрытую, зависящую от файловой системы структуру данных. Обычно vnode является частью такой структуры, следовательно, происходит размещение обоих элементов в едином блоке. Оба объекта инициализируются путем чтения атрибутов файла. Функция устанавливает поле v_op структуры vnode в значение, указывающее на вектор vnodeops для данной файловой системы, после чего на vnode запрашивается ссылка. Последним действием процедуры поиска является добавление нового вхождения в кэш просмотра каталогов и помещение его в конец списка LRU кэша.

8.10.4. Открытие файла

В список входных аргументов вызова open входят полное имя, набор флагов и привилегий, устанавливаемых для файла, если его необходимо создать. Набор флагов включает в себя O_READ, O_WRITE, O_APPEND, O_TRUNC (обрезать до нулевой длины), O_CREAT (создать новый файл, если таковой не существует) и O_EXCL (используется совместно с O_CREAT для возврата ошибки, если файл с таким именем уже существует). Реализация вызова open почти полностью находится на уровне, независимом от файловой системы. Алгоритм функции приведен ниже.

1. Запрос файлового дескриптора (см. раздел 8.2.3). Если выполнение функции open завершается успешно, ее выходным аргументом является индекс, идентифицирующий дескриптор в списке блоков.
2. Запрос объекта открытого файла (struct file) и сохранение указателя на него в дескрипторе файла. В системе SVR4 размещение таких объектов происходит динамически. В более ранних реализациях для этой цели используется статическая таблица фиксированного размера.
3. Вызов lookuppn() для получения имени пути и возврата vnode на открываемый файл. Операция lookuppn() также возвращает указатель на vnode родительского каталога.
4. Проверка vnode (при помощи операции VOP_ACCESS) на обладание процессом, вызвавшим функцию open, полномочиями, необходимыми для доступа к файлу в выбранном режиме.
5. Проверка и исключение неверных операций, таких как попытка открытия каталога или активного выполняемого файла на запись (в противоположном случае пользователь, вызвавший программу, может получить неожиданные результаты).
6. Если файл не существует, проверка, не указан ли флаг O_CREAT. Если так, то исполняется VOP_CREATE по отношению к родительскому каталогу для создания файла. В противном случае выполнение завершается с кодом ошибки ENOENT.

7. Вызов VOP_OPEN по отношению к объекту vnode для проведения действий, зависящих от файловой системы. Обычно такая процедура не делает ничего, но в некоторых системах именно на этой стадии необходимо производить дополнительные действия. Например, в файловой системе *specfs*, обрабатывающей все файлы устройств, иногда необходимо вызвать процедуру open определенного драйвера устройства.
8. Если указана опция O_TRUNC, вызывается VOP_SETATTR для установки размера файла, равного нулю. Необходимую очистку, например освобождение блоков данных файла, производит код, зависящий от конкретной файловой системы.
9. Инициализация объекта открытого файла. Сохранение в нем указателя на vnode и флага режима открытия файла. Установка его счетчика ссылок в значение «единица», а указателя смещения — в нулевую позицию.
10. Возврат индекса дескриптора файла пользователю.

Помните о том, что операция lookuppn() производит инкремент счетчика ссылок vnode и инициализирует его указатель v_op. Это гарантирует последующим системным вызовам получение доступа к файлу при помощи его дескриптора (так как объект vnode остается в памяти), а также выполнение нужных функций, зависящих от файловой системы.

8.10.5. Файловый ввод-вывод

Для того чтобы произвести операции ввода-вывода с файлом, пользователь предварительно должен его открыть и только после этого вызывать функции read или write (readv или writew для ввода-вывода сборкой-разъединением) по отношению к дескриптору, возвращенному open. Независимый от файловой системы код помещает параметры вызова в структуру uio, как это описывалось в разделе 8.2.5. В случае применения вызовов read или write структура uio будет указывать на массив iovec[], состоящий из одного элемента. Ядро использует дескриптор файла для обнаружения объекта открытого файла и проверки на то, что данный файл был открыт в нужном режиме доступа. Если режим окажется правильным, ядро отложит указатель vnode объекта открытого файла для обнаружения vnode.

В системе UNIX вызовы, осуществляющие ввод-вывод, должны выполняться последовательно. Если два пользователя одновременно вызывают read или write по отношению к одному и тому же файлу, ядро прежде завершает одну операцию и только затем начинает выполнение второй. Следовательно, ядро блокирует vnode перед началом выполнения read или write и освобождает объект после завершения ввода-вывода. В системе SVR4 блокировка и освобождение производятся при помощи операций VOP_RWLOCK и VOP_RWUNLOCK. После этого система вызывает функцию VOP_READ или VOP_WRITE для выпол-

нения действий, зависящих от конкретной файловой системы. Большая часть этих действий производится на зависимом от файловой системы уровне. Более подробно о них вы прочтете в разделах 9.3.3 (для s5fs) и 10.6.3 (для NFS). В системе SVR4 операции ввода-вывода файлов схожи с обработкой подсистемы виртуальной памяти, что будет обсуждаться позже, в разделе 14.8.

8.10.6. Атрибуты файлов

Для изменения или запроса специфических атрибутов файлов, таких как идентификатор владельца или права (см. раздел 8.2.2), существует несколько различных системных вызовов. В ранних версиях UNIX такие вызовы производили прямое чтение или запись в поля индексного дескриптора, хранящегося в памяти. Если было необходимо, атрибуты копировались в дескриптор, находившийся на диске. Действия функций сильно зависели от конкретной реализации системы. Так как архитектура vnode/vfs поддерживает большое количество типов файловых систем, каждя из которых может иметь различные методы хранения структур метаданных в памяти или на диске, то можно сказать, что она предлагает общий интерфейс взаимодействия.

Для чтения и записи атрибутов файла применяются операции VOP_GETATTR и VOP_SETATTR, при этом используется независимый от файловой системы объект под названием vattr. Хотя эта структура и содержит информацию, идентичную с содержимым индексного дескриптора s5fs или ufs, ее формат является более общим и не зависит от какой-либо одной файловой системы. Это дает возможность специфическими реализациям преобразовывать информацию между общей структурой и собственными структурами метаданных.

8.10.7. Права пользователя

Некоторым системным файловым операциям необходимо проверять наличие права пользователя на доступ к файлу в выбранном режиме. Управление доступом производится путем проверки идентификаторов группы и пользователя, вызвавшего функцию. Перечисленные идентификаторы обычно хранятся в области u вызывающего процесса. В современных системах UNIX такая информация собрана в *мандате* (credentials) (структура cred), который передается принудительно (через указатель) большинству файловых операций.

Каждый процесс обладает мандатом, статично размещенным в области u или структуре proc. Для проведения операций над локальными файлами мы передаем указатель на этот объект, что не отличается от трактовки прямой передачи информации из области u, принятой в более ранних версиях системы. Преимуществом нового подхода является поддержка удаленных файловых операций, которые выполняются серверными процессами по запросу клиентов. Следовательно, в этом случае полномочия определяются правами клиента, а не процесса на сервере. Таким образом, сервер может динамиче-

ски размещать структуры прав для каждого запроса клиента и инициализировать их в зависимости от его идентификаторов пользователя и группы.

Так как права могут передаваться от одной операции к другой и должны поддерживаться до завершения действий, ядро ассоциирует с каждым мандатом счетчик ссылок, освобождая структуру, если его значение станет равным нулю.

8.11. Анализ

Интерфейс vnode/vfs представляет мощную архитектуру программирования. Он позволяет сосуществовать в одной ОС сразу нескольким файловым системам. Производители ОС имеют возможность добавлять новые файловые системы в ядро в виде модулей. Базовые элементы, построенные на объектно-ориентированном подходе, эффективно разграничивают файловую систему от остальной части ядра. Это способствовало развитию нескольких интересных реализаций файловых систем. При установке SVR4 в типовом комплекте поддерживаются следующие типы файловых систем:

s5fs	Оригинальная файловая система System V
FFS	Berkeley Fast File System (быстрая файловая система), адаптированная для поддержки интерфейса vfs/vnode
vxfs	Журнальная файловая система Veritas, обладающая несколькими дополнительными возможностями
specfs	Файловая система для специальных файлов устройств
NFS	Network File System (сетевая файловая система)
RFS	Remote File Sharing File System (система разделения удаленных файлов)
fifofs	Система для файлов FIFO («первым зашел, первым вышел»)
/proc	Файловая система, в которой каждый процесс представлен в виде файла
bfs	Boot File System (загрузочная файловая система)

Многие варианты UNIX, например Solaris, дополнительно поддерживают файловую систему FAT MS-DOS. Такая возможность позволяет обмениваться данными между машинами DOS и UNIX при помощи гибких дисков. Более подробно о файловых системах вы прочтете в следующих главах книги.

Архитектура vnode/vfs, представленная в системе SunOS, получила в дальнейшем широкое распространение. Однако при ее рассмотрении необходимо поговорить и о недостатках, а также посмотреть, как они были исправлены в других вариантах системы UNIX. Большинство проблем архитектуры происходит от применяемого метода преобразования полных имен. Рассмотрению проблем vnode/vfs и их решению в современных реализациях UNIX будет посвящена оставшаяся часть главы.

8.11.1. Недостатки реализации в системе SVR4

Одной из главных проблем, влияющих на производительность системы, является работа функции lookuppn(), просматривающей только один компонент за раз. При этом для каждого компонента вызывается зависимая от файловой системы функция VOP_LOOKUP. Такой подход не только приводит к чрезмерным перегрузкам, но и в случае применения для удаленных файловых систем требует большого объема взаимодействий между клиентом и сервером. Этот метод был выбран компанией Sun в первую очередь из соображений поддержки ее Network File System (NFS), где процедура поиска ограничена одним компонентом за один проход. Так как это не является жизненно важным для удаленных файловых систем, оптимальным решением будет предоставление выбора просматриваемой части имени файла за один прием каждой файловой системе по отдельности.

Вторая проблема возникает из-за того, что операция подстановки имен не сохраняет информацию о состоянии по причине самой природы протокола NFS. Поскольку операция не блокирует родительский каталог, не существует гарантии правильности полученного результата через любой промежуток времени. Приведем пример возникновения такой ситуации.

Представьте, что пользователь вводит запрос на создание файла /a/b. Операция создания занимает два этапа. Сначала ядро производит просмотр имен для определения, не существует ли уже файл с таким именем. Если такой файл не обнаружен, ядро создает его при помощи функции VOP_CREATE с параметром объекта vnode для /a. Проблемным является временной промежуток между возвратом функции подстановки (с результатом ENOENT) и вызовом функции создания файла, так как в это время каталог /a не блокируется. Следовательно, любой другой процесс может создать в нем файл с тем же именем. Для корректности можно заставить операцию VOP_CREATE просматривать каталог заново, однако это приведет к дополнительной загрузке системы.

Одним из методов избавления от дополнительных ненужных действий является изменение работы функции преобразования, которая не станет транслировать последний компонент в том случае, если после ее запуска будет вызвана операция create или delete. Это уменьшает модульность интерфейса, так как операции не зависят друг от друга.

Существуют иные варианты реализации поддержки нескольких файловых систем в UNIX. Каждый из них основан на концепции существования единого интерфейса, реализованного по-своему конкретными файловыми системами. Различия между ними происходят от различных целей, поставленных перед разработчиками, расхождений в базовых операционных системах, а также от специфики файловых систем, для поддержки которых создается новый интерфейс. Одними из самых ранних альтернатив явились системы

file system switch в ОС SVR3 UNIX корпорации AT&T [12] и *generic file system* (gfs) в системе ULTRIX [13]. Обе архитектуры сохранили концепцию применения индексного дескриптора как базового объекта, представляющего файл, но разделили его на зависящую и независящую от файловой системы части. Современные варианты UNIX, такие как 4.4BSD и OSF/1, поддерживают альтернативные модели интерфейса vnode. Они будут описаны в следующих разделах.

Однако в интерфейсе vnode/vfs имеются и другие, более фундаментальные проблемы. Хотя его дизайн и позволяет создавать модульные файловые системы, он все-таки недостаточно гибок для этой цели. Невозможно написать код любой, пусть даже самой простейшей файловой системы без использования исходных кодов самой ОС. Существует множество запутанных взаимосвязей между подсистемами работы с файлами и обработкой памяти, часть из которых будет рассмотрена позже, в разделе 14.8. Более того, интерфейс не остается неизменным в различных вариантах UNIX, и даже между различными реализациями одной и той же ОС. В результате, несмотря на существование небольшого количества новых полнофункциональных файловых систем, основанных на описываемом интерфейсе (таких как Episode или BSD-LFS, описание которых вы найдете в главе 11), создание файловой системы остается весьма сложной задачей. Более подробно об этом вы можете прочесть в разделе 11.11, где показано описание нового интерфейса *стековых vnode* (stackable vnode), обладающего лучшими возможностями для построения файловых систем по сравнению с vnode/vfs.

8.11.2. Модель 4.4BSD

Разработчики реализации интерфейса vnode/vfs для 4.4BSD [7] попытались преодолеть проблемы варианта SunOS/SVR4 путем улучшения функционирования операции подстановки при помощи встраивания в нее возможностей интерфейса *namei* ОС 4.3BSD (и GFS). Новая версия позволила производить блокировку объектов vnode сразу на время проведения нескольких операций и передавать информацию о состоянии между зависящими друг от друга действиями многоступенчатых системных вызовов.

Просмотр путей в системе 4.4BSD производится при помощи процедуры namei(), которая, в свою очередь, вызывает процедуру поиска vnode текущего каталога. Для преобразования ей передается полное имя пути. Зависимая от файловой системы процедура подстановки может преобразовывать сразу один или более компонентов в одиночный вызов, но не обладает возможностью пересечения точки монтирования. Некоторые реализации процедуры могут преобразовать одновременно лишь один компонент (например, в файловой системе NFS), в то время как реализации s5fs или ufs способны преобразо-

вывать имя пути целиком (если в имени не встречается точка монтирования). После достижения такой точки функция namei() производит необходимые действия и передает оставшуюся часть имени пути следующей операции преобразования.

Аргументы функции подстановки собраны в структуру nameidata. Структура содержит несколько дополнительных полей для передачи информации о состоянии и для внешних данных, возвращаемых функцией. Она может передаваться другим операциям, таким как create или symlink, позволяя взаимосвязанным задачам использовать данные без занесения переменных в стек.

Одно из полей структуры nameidata содержит обоснование преобразования полных имен. Если производится создание или удаление файла, то последняя по счету операция заблокирует объект vnode родительского каталога. Функция также вернет дополнительную информацию через структуру nameidata, например, о расположении файла в текущем каталоге (если файл был обнаружен), о первой свободной ячейке каталога (для дальнейшего создания файла). Затем структура nameidata передается операциям create или delete. Так как родительский каталог остается заблокированным функцией преобразования, ее содержимое не может быть изменено, следовательно, функциям создания или удаления файла не придется повторно выполнять последнюю итерацию поиска. После завершения выполнения операции произойдет освобождение родительского каталога.

Иногда после завершения операции ядро не разрешает производить создание или удаление файла (например, если вызывающий процесс не обладает необходимыми полномочиями). В этом случае ядро загружает операцию abortop для сброса блокировки родительского каталога.

Несмотря на то, что интерфейс запоминает полученную информацию, и блокировка может проводиться сразу на несколько операций, действия по установке и освобождению объектов блокировки производятся на уровне, зависящем от файловой системы. Таким образом, интерфейс может поддерживать одновременно файловые системы, которые могут как сохранять, так и не сохранять информацию о состоянии, не производя избыточные действия, если это возможно. Например, в файловой системе с сохранением (такой как NFS) можно не производить блокировку родительского каталога и также отказаться от поиска последнего компонента, если сразу после преобразования имен будут запущены вызовы create или delete.

Основной проблемой такого подхода является последовательность проведения всех операций над каталогом, так как блокировка объекта удерживается на протяжении всей длительности проведения операции, даже в том случае, если процесс приостанавливается в ожидании завершения ввода-вывода. Реализация вызова использует эксклюзивный объект блокировки, запрещающий выполнять параллельно даже операции чтения. В частности, в системах разделения времени это может привести к возникновению больших задер-

жек при доступе к наиболее часто используемым каталогам, таким как / и /etc. Для многопроцессорных систем описанная методика последовательной работы операций является неприемлемой.

Еще одним изменением, внесенным разработчиками новой версии интерфейса vnode/vfs, является поддержка кэша каталогов и смещения последнего, удачного завершившегося поиска имен каждого процесса. Это увеличивает производительность операций, производящих действия над всеми файлами в каталоге. Операция namei() использует кэш во время поиска последнего компонента имени. Если родительский каталог совпадает с указанным для предыдущего вызова namei(), поиск начнется от смещения, указанного в кэше (вместо начала каталога).

Интерфейс файловой системы 4.4BSD предлагает и другие интересные возможности, например *стековую организацию vnode* (stackable vnode) и *объединенное монтирование* (union mount), которые будут описаны позже, в разделах 11.11–11.12.

8.11.3. Средства системы OSF/1

В системе OSF/1 были решены проблемы производительности, связанные с избыточными операциями над каталогами, при этом возможность сохранения информации о состоянии в реализации интерфейса vnode осталась. Более того, представленный интерфейс корректно работает как на однопроцессорных, так и многопроцессорных системах. Такой результат был достигнут применением передачи всей информации о состоянии между всеми взаимосвязанными процессами как рекомендации (hint). Рекомендации ассоциируются с временными отметками файлов, которые в дальнейшем нужны для проверки правильности последующих операций. Файловая система может использовать рекомендации по своему усмотрению с целью предупреждения лишней проверки на отсутствие изменений данных между операциями.

Метаданные файла защищаются посредством взаимоисключающего объекта блокировки (mutex), поддерживающего многопроцессорную обработку. Такой объект реализуется в виде циклической блокировки (spin lock) и удерживается в течение коротких промежутков времени. В частности, блокировка происходит только на этапе выполнения участка кода, зависящего от файловой системы. Объект никогда не удерживается при проведении нескольких операций, что означает возможность обмена метаданными между этими операциями. Изменения отслеживаются посредством ассоциации с каждым объектом синхронизации временных меток. Такие метки представляют собой простые счетчики, инкрементируемые при каждом изменении объекта. Функция подстановки кэша 4BSD, основанная на рекомендациях, использует ту же схему (см. раздел 8.10.2).

Приведем небольшой пример. Рассмотрим операцию создания файла. При поиске последнего компонента имени операция поиска блокирует родительский каталог и проверяет, не находится ли в нем заданный файл. Если файл с таким именем не будет обнаружен, функция определяет смещение каталога, начиная с которого можно поместить новый элемент. Затем происходит отмена блокировки родительского каталога и возврат полученной информации вызвавшему процессу (вместе с временной меткой). После этого ядро вызывает direnter() для создания нового элемента родительского каталога и передает ему смещение и сохраненную временную метку. Функция direnter() сравнит полученное значение метки с текущим значением. Если значения совпадут, это будет означать, что каталог не был изменен в течение преобразования, и новое имя может быть вставлено в выбранную точку смещения без повторной проверки каталога. Если временные метки окажутся разными, то каталог был изменен, и необходимо еще раз провести операцию подстановки.

Описанные улучшения были произведены в первую очередь из-за необходимости поддержки многопроцессорных платформ, однако преимущества интерфейса проявляются и в случае применения в однопроцессорной системе. Изменения могут стать причиной возникновения состязательности, но система защищена от них при помощи методов, описанных в [9]. Модель OSF/1 объединяет в себе преимущества архитектур с сохранением и несохранением информации и поддерживает различные файловые системы, такие как AFS (см. раздел 10.15) и Episode (см. раздел 11.8).

8.12. Заключение

Интерфейс vnode/vfs является мощным механизмом разработки модулей и добавления новых файловых систем в ядро UNIX. Он позволяет ядру производить действия с абстрактным представлением файлов, называемым vnode, и выполнять зависимый от файловой системы код на отдельном уровне, доступном посредством четко очерченного интерфейса. Разработчики могут строить собственные файловые системы, реализующие этот интерфейс. Процесс создания новой системы схож с написанием драйвера устройства.

Важно знать, что между реализациями интерфейса vnode/vfs существуют большие различия. Хотя многие реализации, например, представленые в системах SVR4, BSD и OSF/1, основаны на одних и тех же общих принципах, они значительно разнятся между собой в специфике интерфейсов (то есть наборами операций и их аргументами, а также форматами структур vnode и vfs) и правилах, касающихся состояний, синхронизации и т. д. Это означает, что разработчики файловых систем должны производить большие изменения в кодах для поддержки совместимости своих систем с различными реализациями интерфейса vfs.

8.13. Упражнения

1. Какие существуют преимущества в представлении файлов как потока байтов? В каких ситуациях такая модель является неподходящей?
2. Представьте, что программа производит повторяющиеся вызовы readdir для получения списка содержимого каталога. Что произойдет в том случае, если пользователь создаст или удалит файл в этом каталоге между двумя вызовами?
3. Почему пользователи не имеют права производить запись в каталог напрямую?
4. Почему атрибуты файла не сохраняются в самом элементе каталога?
5. Почему каждый процесс обладает собственной маской создания, заданной по умолчанию? Где хранятся такие маски? Почему ядро системы не устанавливает режим в вызовах open или creat непосредственно?
6. В силу чего пользователь не может производить запись в файл, открытый в режиме «только для чтения», даже если он обладает привилегиями, необходимыми для такого действия?
7. Ниже приведен командный сценарий под названием myscript.

```
date
cat /etc/motd
```

   Что произойдет в результате выполнения команды

```
myscript > result.log
```

   Как при этом производится разделение дескрипторов файлов?
8. Какие преимущества дает существование отдельного системного вызова lseek вместо передачи стартового смещения каждой операции read или write? Какие недостатки у этой схемы?
9. Почему функция read может вернуть меньшее количество байтов, чем было указано при ее вызове?
10. Какими преимуществами обладает произвольный ввод-вывод? Для каких приложений этот метод доступа является наиболее предпочтительным?
11. Чем отличаются друг от друга рекомендательная и принудительная блокировка? Для каких типов приложений предпочтительнее использовать объекты блокировки байтовых областей[1]?
12. Текущим рабочим каталогом пользователя является /usr/mnt/kaumu. Если администратор произведет монтирование новой системы в каталоге /usr/mnt, как это повлияет на пользователя? Будет ли пользователь

[1] Записей файла. — *Примеч. ред.*

иметь возможность видеть файлы, расположенные в каталоге kaumu? Каков будет результат выполнения команды pwd? Какое нестандартное поведение можно ожидать от других вызываемых команд?

13. Перечислите недостатки применения символических ссылок по сравнению с жесткими ссылками.
14. Почему жесткие ссылки не имеют возможности объединять разные файловые системы?
15. Какие проблемы могут возникать при некорректном применении жестких ссылок на каталоги?
16. Какие действия производит ядро системы, когда значение счетчика ссылок на vnode становится равным нулю?
17. Расскажите об основных преимуществах и недостатках кэша подстановки имен файлов, основанного на счетчиках и на рекомендациях.
18. В работах [2] и [6] описываются две различные реализации, получающие и освобождающие объекты vnode динамически. Могут ли такие системы использовать кэш имен, основанный на рекомендациях?
19. Приведите пример бесконечного цикла, возникающего при неверно заданной символической ссылке. Каким образом происходит его обработка функцией lookuppn()?
20. Почему операция VOP_LOOKUP за один проход просматривает только одно имя?
21. Система 4.4BSD позволяет процессу блокировать объект vnode на время нескольких операций над ним, происходящих внутри одного системного вызова. Что произойдет в том случае, если процесс будет принудительно завершен по сигналу во время удержания блокировки? Каким образом ядро обрабатывает такие ситуации?

8.14. Дополнительная литература

1. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
2. Barkley, R. E., and Lee, T. P., «A Dynamic File System Inode Allocation and Reallocation Policy», Proceedings of the Winter 1990 USENIX Technical Conference, Jan. 1990, pp. 1–9.
3. Department of Defense, «Trusted Computer System Evaluation Criteria», DOD 5200.28–STD, Dec. 1985.
4. Ellis, M. A., and Stroustrup, B., «The Annotated C++ Reference Manual», Addison-Wesley, Reading, MA, 1990.

5. Fernandez, G., and Allen, L., «Extending the UNIX Protection Model with Access Control Lists», Proceedings of the Summer 1988 USENIX Technical Conference, Jun. 1988, pp. 119–132.
6. John, A., «Dynamic Vnodes — Design and Implementation», Proceedings of the Winter 1995 USENIX Technical Conference, Jan. 1995, pp. 11–23.
7. Karels, M. J. and McKusick, M. K., «Toward a Compatible Filesystem Interface», Proceedings of the Autumn 1986 European UNIX Users' Group Conference, Oct. 1986, pp. 481–496.
8. Kleiman, S. R., «Vnodes: An Architecture for Multiple File System Types in Sun UNIX», Proceedings of the Summer 1986 USENIX Technical Conference, Jun. 1986, pp. 238–247.
9. LoVerso, S., Paciorek, N., Langerman, A., and Feinberg, G., «The OSF/1 UNIX Filesystem (UFS)», Proceedings of the Winter 1991 USENIX Conference, Jan. 1991, pp. 207–218.
10. McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S., «A Fast File System for UNIX», ACM Transactions on Computer Systems, Vol. 2, (Aug. 1984), pp. 181–197.
11. Patterson, D. A., Gibson, G. A., and Katz, R. H., «A Case for Redundant Arrays of Inexpensive Disks (RAID)», Proceedings of the 1988 ACM SIGMOD Conference of Management of Data, Jun. 1988.
12. Rifkin, A. P., Forbes, M. P., Hamilton, R. L., Sabrio, M., Shah, S., and Yueh, K. «RFS Architectural Overview», Proceedings of the Summer 1986 USENIX Conference, Jun. 1986, pp. 248–259.
13. Rodrigues, R., Koehler, M., and Hyde, R., «The Generic File System», Proceedings of the Summer 1986 USENIX Technical Conference, Jun. 1986, pp. 260–269.
14. Thompson, K., «UNIX Implementation», The Bell System Technical Journal, Vol. 57, No. 6, Part 2, pp. 1931–1946.
15. UNIX Systems Laboratories, «Operating System API Reference: UNIX SVR4.2», UNIX Press, Prentice-Hall, Englewood Cliffs, NJ, 1992.

Глава 9
Реализации файловых систем

9.1. Введение

В предыдущей главе книги был описан интерфейс vnode/vfs, предоставляющий базовую структуру, поддерживающую различные типы файловых систем и определяющую способы взаимодействия между файловой системой и остальной частью ядра. Современные варианты UNIX используют многие типы файловых систем, которые можно разделить на две большие категории. В локальных файловых системах осуществляется обработка и хранение на устройствах, подключенных к машине напрямую. Распределенные файловые системы позволяют пользователям иметь доступ к файлам, находящимся на удаленных компьютерах. В этой главе будут рассмотрены различные реализации локальных систем. О распределенных системах вы узнаете из 10 главы. Глава 11 расскажет о некоторых новых файловых системах, включающих такие дополнительные возможности, как журналы, обработку томов и высокий уровень доступности.

Большинство современных вариантов UNIX работает с двумя локальными системами, применяемыми для общих целей. Это — System V file system (s5fs) и Berkeley Fast File System (FFS). Первая из них, s5fs [23], является оригинальной файловой системой, поддерживаемой UNIX изначально. Ее можно найти во всех вариантах System V, а также в некоторых современных коммерческих вариантах UNIX. Файловая система FFS была разработана в лабораториях Беркли и впервые представлена в 4.2BSD. Система имела большую производительность, надежность и функциональность по сравнению с s5fs. Система FFS получила высокое признание среди производителей ОС, кульминацией которого стало ее включение в SVR4 (SVR4 поддерживает три типа файловых систем: s5fs, FFS и VxFS, журнальную файловую систему Veritas).

Во времена появления FFS базовая структура UNIX могла использовать одновременно только один тип файловой системы. Производители ОС встали перед выбором, какую из них выбрать. Интерфейс vnode/vfs, представ-

ленный компанией Sun Microsystems [9], позволил сочетать несколько файловых систем на одной машине. Для интеграции существующих файловых систем с интерфейсом vnode/vfs необходимо было произвести в них некоторые изменения. Обновленная версия FFS известна сейчас под именем *файловой системы UNIX* (*ufs*, UNIX file system)[1]. Подробное описание s5fs можно найти в [2], рассказ о FFS представлен в книге [12]. В этой главе вы увидите краткий анализ и сравнение этих файловых систем. Описание s5fs и FFS включено в книгу для полноты материала и изложения читателю базовых концепций, упрощающих понимание архитектуры более сложных файловых систем, обсуждаемых в следующих главах.

В UNIX понятие файл включает в себя различные абстракции, в том числе сетевые соединения через сокеты или STREAMS, механизмы взаимодействия процессов, такие как каналы или FIFO, а также блочные и символьные устройства. На этом была построена архитектура vnode/vfs, в которой файлы и файловые системы являются базовыми элементами, предоставляющими модульный интерфейс взаимодействия с остальной частью ядра. Такой подход привел к созданию специализированных файловых систем. Во многие из них не внесено практически никаких изменений в работу с файлами, однако интерфейс подвергнут полной переработке с целью добавления специализированных функций. С несколькими различными реализациями таких систем вы познакомитесь в этой главе.

Здесь вы также прочтете о *буферном кэше* UNIX. В ранних реализациях системы, таких как SVR3 или 4.3BSD, кэш применялся при всех операциях ввода-вывода. В современных вариантах, таких как SVR4, функции работы с памятью и обработки ввода-вывода являются интегрированными. Доступ к файлам производится путем отображения их в адресное пространство ядра. В этой главе будут представлены некоторые детали архитектуры, однако более подробно о ней можно будет прочесть в главе 14, посвященной работе с виртуальной памятью в SVR4. Традиционный механизм буферного кэша использует блоки *метаданных*. Термин «метаданные» применяется для обозначения атрибутов и служебной информации файла или файловой системы. До того как буферный кэш стал частью конкретных файловых систем, он являлся глобальным ресурсом, используемым всеми файловыми системами.

Глава начнется с рассказа s5fs и описания принятой в ней организации хранения данных на диске и в ядре. Файловая система FFS имеет некоторые отличия от s5fs, однако базовые операции в обеих системах реализованы сходным образом. Обсуждение FFS будет посвящено ее отличиям от ориги-

[1] Изначально термин ufs имел различные толкования в зависимости от используемой системы. В реализациях, основанных на System V, он употреблялся для обозначения первоначальной файловой системы UNIX, известной сейчас под именем s5fs. Системы на основе BSD применяют термины ufs и s5fs аналогично обозначениям, принятым в этой книге. Неудобства, связанные с путаницей терминов, были преодолены с появлением SVR4, в которой был такой же вариант их толкования.

нальной системы UNIX. Если это не оговорено специально, все общие алгоритмы, приводимые для s5fs в разделе 9.3, также относятся и к системе FFS.

9.2. Файловая система System V (s5fs)

Файловая система располагается на одном логическом диске или в разделе (см. раздел 8.3.1). Каждый логический диск может содержать в себе лишь одну файловую систему. Всякая файловая система является независимой и содержит собственный корневой каталог, подкаталоги, файлы, а также ассоциированные с ними данные и метаданные. Файловое дерево, видимое пользователем, формируется из одной или нескольких файловых систем.

На рис. 9.1 представлено размещение раздела s5fs на диске. Раздел может быть логически представлен как одномерный массив блоков. Размер дискового блока начинается от 512 байт и увеличивается по степени с основанием 2 (различные реализации системы поддерживают блоки размерами 512, 1024 или 2048 байт). Это число представляет собой единицу дробления дискового пространства при размещении файла и объема данных для операций ввода-вывода. Физический номер блока (или просто номер блока) является индексом в массиве блоков и используется для уникальной идентификации блока на разделе диска. Этот номер преобразуется дисковым драйвером в номера цилиндра, дорожки и секторы на диске. Преобразование зависит от физических характеристик конкретного диска (общего количества цилиндров и дорожек, количества секторов на одной дорожке и т. д.), а также от местонахождения раздела на диске.

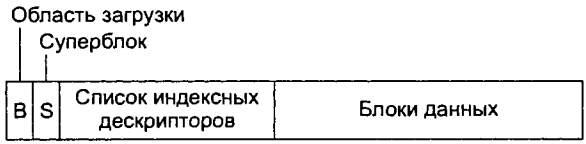


Рис. 9.1. Файловая система s5fs. Размещение на диске

В начале раздела находится *загрузочная область*, содержащая код, необходимый для *начальной загрузки* (и инициализации) операционной системы. В принципе, для хранения подобной информации достаточно одного раздела диска, однако каждый имеющийся на диске дополнительный раздел должен по возможности иметь такую область (пустую). После области загрузки на диске располагается суперблок, содержащий атрибуты и метаданные файловой системы.

Далее размещается *список индексных дескрипторов*, являющийся одномерным массивом индексных дескрипторов. Такой дескриптор имеется у каждого файла. Более подробно об индексных дескрипторах читайте в разделе 9.2.2. Каждый дескриптор идентифицируется при помощи *номера*, эквивалентного

индексу в списке дескрипторов. Размер дескриптора составляет 64 байт. В одном дисковом блоке могут быть размещены несколько дескрипторов. Начальные смещения для суперблока и списка индексных дескрипторов одинаковы во всех разделах системы. Следовательно, номер дескриптора может быть легко преобразован в номер блока и смещение от начала этого блока. *Список индексных дескрипторов имеет фиксированный размер (устанавливаемый при создании файловой системы на разделе), который ограничивает максимально допустимое число файлов в этом разделе.* После таблицы дескрипторов на диске следует область данных. В ней расположены блоки данных для хранения файлов и каталогов, а также *блоки косвенной адресации,* содержащие указатели на блоки данных файлов, которые будут описаны в разделе 9.2.2.

9.2.1. Каталоги

Каталог в системе s5fs представляет собой специальный файл, содержащий список файлов и подкаталогов. В нем находятся записи фиксированного размера (по 16 байт). В первых двух байтах размещается номер индексного дескриптора, следующие 14 отведены для имени файла. Размер одной записи ограничивает общее количество файлов на разделе диска до 65 535 (так как 0 не может являться номером индексного дескриптора) и имя файла 14 символами. Если имя файла меньше 14 символов, то оно должно быть завершено символом NULL. Так как каталог тоже является файлом, он имеет и собственный индексный дескриптор с полем, идентифицирующим его как каталог. Первыми двумя элементами каталога являются «.» (собственно этот каталог) и «..» (родительский каталог). Если номер индексного дескриптора равен нулю, это означает, что файл, на который он указывает, больше не существует. Корневой каталог раздела, а также элемент «..» всегда имеет номер дескриптора, равный 2. При помощи этого значения файловая система может идентифицировать корневой каталог. Пример каталога в s5fs показан на рис. 9.2.

73	.
38	..
9	file1
0	deletedfile
110	subdirectory_1
65	archana

Рис. 9.2. Файловая система s5fs. Структура каталога

9.2.2. Индексные дескрипторы

Каждый файл обладает собственным *индексным дескриптором* (inode), ассоциированным с ним. Индексный дескриптор содержит служебную информацию, или метаданные файла. Он хранится на диске вместе со списком индексных дескрипторов. Если файл открыт или каталог является текущим, ядро

сохраняет данные о них из дисковой копии дескриптора в структуре данных, размещенной в памяти, также называемой индексным дескриптором. Эта структура имеет некоторые дополнительные поля, не повторенные на диске. Несмотря на определенную неоднозначность, мы используем термин *дескриптор на диске* (on-disk inode) для указания на структуру данных, размещенных на диске (dinode), а термин *дескриптор в памяти* (in-core inode) – для обозначения структуры, размещаемой в оперативной памяти (inode). Поля структуры dinode представлены в табл. 9.1.

Таблица 9.1. Поля структуры dinode

Поле	Размер в байтах	Описание
di_mode	2	Тип файла, привилегии и т. д.
di_nlinks	2	Количество жестких ссылок на файл
di_uid	2	Идентификатор пользователя-владельца
di_gid	2	Идентификатор группы пользователя-владельца
di_size	4	Размер файла в байтах
di_addr	39	Массив адресов блоков
di_gen	1	Генерируемый номер (инкрементируется каждый раз при запросе индексного дескриптора для нового файла)
di_atime	4	Время последнего доступа
di_mtime	4	Время последней модификации
di_ctime	4	Время последнего изменения индексного дескриптора (кроме изменений полей di_atime и di_mtime)

Поле di_mode подразделяется на несколько битовых полей (рис. 9.3). Первые четыре бита определяют тип файла, который может иметь значения: IFREG (обычный файл), IFDIR (каталог), IFBLK (блочное устройство), IFCHR (символьное устройство) и т. д. Последние девять битов определяют права на чтение, запись и выполнение для владельца, членов группы владельца и остальных пользователей соответственно.

Рис. 9.3. Структура поля di_mode

Поле di_addr требует дополнительных разъяснений. Файлы в системе UNIX расположены на диске не непрерывно. При увеличении размера файла ядро запрашивает для него новые блоки, которые могут находиться в любом удобном месте диска. Такой подход дает возможность легко увеличивать или уменьшать размеры файла без фрагментации диска, наследуемой от последовательных схем размещения. Однако это не защищает полностью от пробле-

мы фрагментации, так как последний блок каждого файла может содержать неиспользованное пространство. В среднем на каждый файл теряется около половины блока дискового пространства.

Описанная методика размещения требует поддержки файловой системой карты расположения на диске каждого блока файла. Такой список организован в виде массива физических адресов блоков. Логический номер блока вместе с файлом являются индексом к этому массиву. Большинство файлов, однако, являются не очень большими [21], поэтому поддержка объемного массива приведет лишь к ненужной трате дискового пространства. Более того, хранение массива адресов блоков диска в отдельном блоке потребует дополнительной операции чтения при осуществлении доступа к файлу, что отрицательно повлияет на производительность системы.

Решение, представленное в ОС UNIX, предлагает хранить небольшой список блоков прямо в индексном дескрипторе и использовать дополнительные блоки для файлов больших размеров. Этот метод весьма эффективен для маленьких файлов, но недостаточно гибок для больших файлов. Схема размещения представлена на рис. 9.4. Поле di_addr, занимающее 39 байт, содержит 13-элементный массив, каждый элемент которого хранит в себе 3-байтовый номер физического блока. Элементы от 0 до 9 содержат номер 0–9 блоков данных файла. Таким образом, для файла, умещающегося в 10 и менее блоков, вся адресная информация хранится в индексном дескрипторе. Элемент 10 представляет собой номер блока косвенной адресации с содержимым в виде массива номеров блоков. Элемент 11 указывает на блок двойной косвенной связи, содержащий номера других «косвенных» блоков. И последний, 12-й элемент, указывает на «трижды косвенный» блок с номерами «дважды косвенных» блоков.

В этой схеме при размере блока в 1024 байт первые 10 блоков можно адресовать непосредственно, еще 256 блоков через один блок косвенной связи, еще 65 536 (256×256) блоков через двойной косвенный блок и еще 16 777 216 (256×256×256) блоков — через блок тройной косвенной адресации.

Файлы UNIX могут содержать *пустоты*. Пользователь может создать файл, перейти к большой величине смещения (установить указатель смещения в объекте открытого файла при помощи lseek, см. раздел 8.2.4) и произвести туда запись. Пространство перед этим смещением не содержит каких-либо данных, то есть является «дыркой» в файле. Если процесс попытается прочесть данные из такой области, то увидит байты со значением NULL.

Пустоты иногда бывают такой величины, что перекрывают сразу же несколько блоков. Предоставлять дисковое пространство таким блокам неразумно. Вместо этого ядро устанавливает соответствующие элементы массива di_addr, либо блока косвенной адресации в ноль. Если пользователь попытается прочесть такой блок, ядро вернет его, заполненный нулями. Дисковое пространство будет занято только в том случае, если будет совершена попытка записи в такой блок.

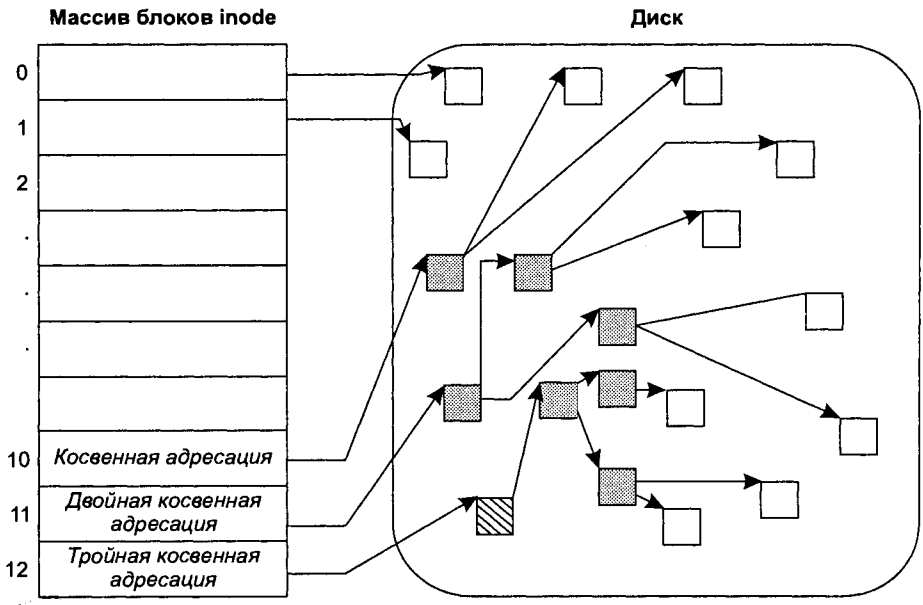


Рис. 9.4. Файловая система s5fs. массив адресов блоков в индексном дескрипторе

Отказ от размещения пустот в дисковом пространстве имеет некоторые важные последствия. Процесс может неожиданно оказаться перед фактом отсутствия места на диске при попытке записи в пустоты. Если произвести копирование файла с пустотами, новый экземпляр будет иметь страницы на диске, заполненные нулями. Это происходит из-за того, что операция копирования требует чтения содержимого файла и записи его в файл назначения. При чтении пустоты ядро создает страницы, заполненные нулями, которые и копируются без дальнейшей интерпретации. *Такое свойство может стать причиной проблем при использовании утилит резервного копирования и упаковщиков типа* tar *или* cpio, *работающих на уровне файлов, а не непосредственно с дисками. Системный администратор может при резервном копировании системы столкнуться с тем, что на другом диске, равном по объему, не хватит места для ее копии.*

9.2.3. Суперблок

Суперблок содержит метаданные применительно к файловой системе. В каждой файловой системе имеется один суперблок, располагающийся в начале диска. Ядро читает данные из суперблока при монтировании файловой системы и хранит их в памяти до тех пор, пока система не будет размонтирована. Суперблок содержит следующую информацию:

- размер файловой системы в блоках;
- размер списка индексных дескрипторов в блоках;
- количество свободных блоков и индексных дескрипторов;

- ✦ список свободных блоков;
- ✦ список свободных индексных дескрипторов.

Так как файловая система может иметь большое количество свободных блоков и индексных дескрипторов, неразумно поддерживать их полный список в суперблоке. В случае индексных дескрипторов суперблок включает в себя только частичный список. Если он становится пустым, ядро сканирует диск на предмет нахождения новых свободных индексных дескрипторов (для которых di_mode == 0) для добавления их в список.

Однако такой подход невозможен для списка свободных блоков, так как не существует другого способа определения незанятости блока кроме проверки его содержимого. Таким образом, файловая система должна постоянно поддерживать полный список всех свободных блоков на диске. Как показано на рис. 9.5, такой список занимает несколько дисковых блоков. Суперблок содержит первую часть списка, удаление и добавление новых блоков происходит с конца. Первый элемент списка указывает на блок, содержащий следующий элемент списка и т. д.

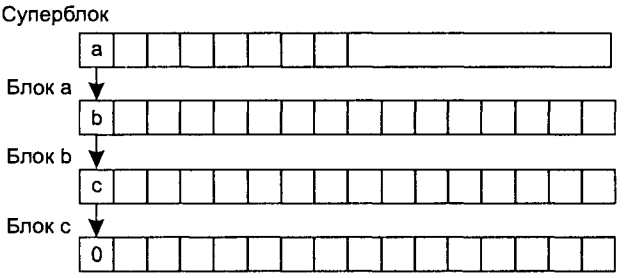


Рис. 9.5. Файловая система s5fs. список свободных блоков

В какой-то момент времени процедура размещения блоков может обнаружить, что список в суперблоке состоит только из одного элемента. Переменная, хранящаяся в этом элементе, является номером блока, содержащего следующий элемент списка свободных блоков (блок A на рис. 9.5). Тогда процедура копирует список из этого блока в суперблок и дополнительный блок становится свободным. Такой подход дает определенные преимущества, поскольку пространство, необходимое для хранения списков свободных блоков, напрямую зависит от общего количества свободного места в разделе диска. Следовательно, для заполненных дисков место на хранение списка свободных блоков не тратится.

9.3. Организация ядра системы s5fs

Индексный дескриптор является фундаментальным объектом файловой системы s5fs. Он представляет собой закрытую структуру данных, ассоциированную с объектом vnode системы s5fs. Как это уже упоминалось ранее, дескрипторы

в памяти отличаются по составу от дескрипторов, размещенных на диске. В этом разделе будет рассказано об индексных дескрипторах, находящихся в оперативной памяти, а также о том, как в системе s5fs реализованы различные операции на основе этих объектов.

9.3.1. Индексные дескрипторы в памяти

Индексный дескриптор в памяти представлен структурой inode, включающей в себя не только все поля дисковой структуры дескриптора, но и некоторые дополнительные поля, такие как:

- *vnode* (поле i_vnode индексного дескриптора, содержащее vnode файла);
- *идентификатор устройства*, относящийся к разделу диска местонахождения файла;
- *номер индексного дескриптора* файла;
- *флаги*, используемые для синхронизации и работы с кэшем;
- указатели на дескрипторы, содержащиеся в *списке свободных индексных дескрипторов*;
- указатели на дескрипторы, содержащиеся в *таблице хэширования*. Ядро хэширует дескрипторы по номерам, что позволяет их при необходимости быстро найти;
- номер *последнего прочитанного блока*.

На рис. 9.6 продемонстрирован простой пример организации индексных дескрипторов в s5fs для случая четырех таблиц хэширования.

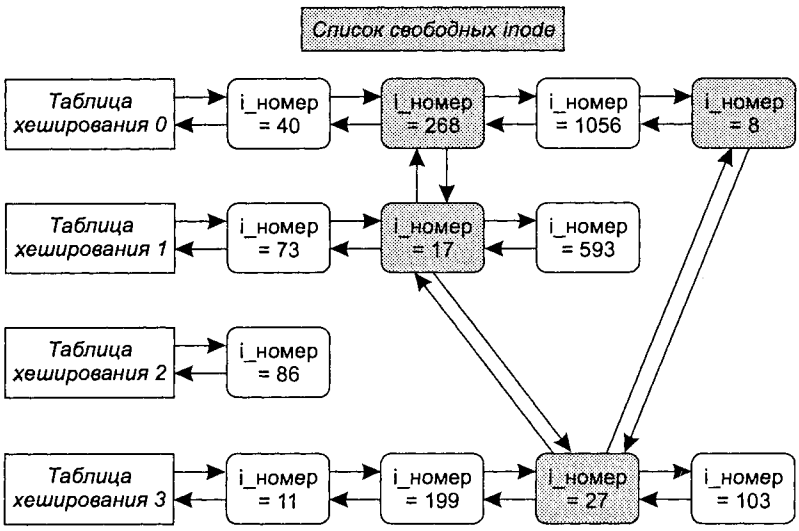


Рис. 9.6. Организация индексных дескрипторов, размещаемых в памяти

Массивы дисковых блоков также обрабатывается по-разному. Массив di_addr[] дескриптора на диске использует для номера каждого блока элемент размером в три байта, дескрипторы в памяти для той же цели применяют четырехбайтовые переменные. Эти различия исходят из необходимости балансировки между занимаемым местом и производительностью работы. Экономия места более важна для дескриптора на диске, в то время как скорость обработки критична прежде всего для дескриптора в памяти.

9.3.2. Получение индексных дескрипторов

Просмотр на независимом от файловой системы уровне выполняется при помощи функции lookuppn(). Как это уже описывалось ранее в разделе 8.10.1, функция просматривает за один проход только один компонент, вызывая для этой цели операцию VOP_LOOKUP. При осуществлении поиска в каталоге s5fs происходит вызов функции s5lookup(), которая начинает свою работу с проверки кэша подстановки имен (см. раздел 8.10.2). Если функция не находит искомый элемент в кэше, то поиск заданного имени файла будет осуществляться путем считывания каталога (по одному блоку за один прием).

Если каталог содержит необходимый элемент, функция s5lookup() запросит для него номер индексного дескриптора. Затем она вызовет iget() для обнаружения дескриптора. Для этого функция произведет поиск индексного дескриптора по его номеру в таблице хэширования. Если дескриптор не будет найден, функция iget() запросит новый индексный дескриптор и инициализирует его путем чтения дискового дескриптора. При копировании его полей в дескриптор, размещаемый в памяти, функция расширяет элементы массива di_addr[] до границы, кратной четырем байтам. Затем индексный дескриптор помещается в соответствующую таблицу хэширования. Функция также осуществляет инициализацию объекта vnode и устанавливает значение его поля v_op на вектор s5vnodeops, поле v_data — на сам индексный дескриптор и поле v_vfsp — на указатель vfs, к которой относится файл. На последнем этапе работы функции iget() происходит возврат указателя на дескриптор операции s5lookup(), которая в свою очередь возвращает указатель на vnode функции lookuppn().

Следует отметить, что iget() является функцией файловой системы s5fs, используемой только для размещения и инициализации индексных дескрипторов и vnode. Например, при создании нового файла операция s5create() производит размещение незадействованного номера индексного дескриптора (из списка свободных индексных дескрипторов суперблока) и далее вызывает iget() для копирования этого дескриптора в память.

9.3.3. Файловый ввод-вывод

Входным аргументом системных вызовов read и write является дескриптор файла (индекс, возвращаемый при его открытии), а также адрес пользовательского буфера, являющийся источником (для чтения) или приемником

(для записи) данных, и счетчик количества передаваемых байтов. Смещение в файле контролируется функциями для объекта открытого файла, ассоциированного с его индексным дескриптором. По завершении операции ввода-вывода значение смещения изменяется на количество переданных байтов данных. Это означает, что последующие вызовы read или write начнут свою работу с той позиции в файле, где был завершен ввод-вывод предыдущего по счету вызова. В случае использования произвольного ввода-вывода пользователю необходимо перед чтением или записью вызвать lseek() для перемещения указателя в нужную позицию файла.

Независимый от файловой системы код (см. раздел 8.6) использует файловый дескриптор как индекс в таблице индексных дескрипторов для получения указателя на объект открытого файла (структуры file), после чего проверяет корректность режима открытия данного файла. Если режим совпадает, ядро запрашивает указатель на vnode их структуры file. Перед началом ввода-вывода ядро вызывает операцию VOP_RWLOCK для запрещения одновременного доступа к файлу. В системе s5fs защита файла реализована при помощи объекта эксклюзивной блокировки по отношению к дескриптору файла[1]. Применение блокировки гарантирует неизменность данных при чтении или записи (в рамках одного системного вызова), а также то, что все вызовы write будут исходить от одной нити. Затем ядро производит запуск VOP_READ или VOP_WRITE, указанных в объекте vnode. Это приведет к последующему вызову зависимых от файловой системы функций s5read() или s5write() соответственно.

В более ранних реализациях системы процедуры ввода-вывода использовали буферный кэш, специальную область памяти, зарезервированную для блоков файловой системы. В SVR4 была произведена унификация ввода-вывода файлов с операциями управления виртуальной памятью, в этой системе буферный кэш применяется только для хранения блоков метаданных. Ранняя версия буферного кэша будет рассмотрена в разделе 9.12. В этом разделе мы вкратце опишем операции, реализованные в SVR4. Подробную информацию о виртуальной памяти можно получить из раздела 14.8.

Рассмотрим ввод-вывод на примере операции read. Функция s5read() преобразует первоначальное смещение в номер логического блока файла и смещение от начала этого блока. Затем функция производит чтение данных постранично[2], отображая блоки в виртуальное адресное пространство ядра. Для копирования данных в пользовательское адресное пространство вызывается функция uiomove(), которая, в свою очередь, запускает процедуру copyout() для передачи текущих данных. Если страница не находится в физической памяти, либо ядро системы не обладает корректным адресом для ее преобра-

[1] Во многих системах UNIX для этой цели используется объект блокировки одного пишущего и множества читающих процессов, что дает возможность более гибко реализовать параллелизм работы ОС.

[2] Страница — это понятие, относящееся к обработке памяти. Страница может содержать один блок, часть блока или несколько блоков.

зования, функция copyout() сгенерирует ошибку страничной памяти. Обработчик ошибок идентифицирует файл, к которому относится эта страница, и выполнит операцию VOP_GETPAGE по отношению к его vnode.

В файловой системе s5fs такая операция реализована при помощи функции s5getpage(). Эта функция начинает работу с вызова bmap() для преобразования номера логического блока в физический номер блока на диске. Затем она производит проверку нахождения страницы объекта vnode (на который указывает поле v_page) в памяти. Если страница не будет обнаружена в памяти, функция s5getpage() запросит свободную страницу и вызовет дисковый драйвер для осуществления операции чтения данных с диска.

Вызывающий процесс будет находиться в режиме ожидания до тех пор, пока операция чтения не завершится. После прочтения блока дисковый драйвер произведет пробуждение процесса, который, в свою очередь, продолжит копирование данных copyout(). Перед копированием данных в пользовательское пространство операция copyout() убедится в наличии прав пользователя на запись в буфер, в который необходимо передать данные. Если не производить такой проверки, то при случайном или преднамеренном указании неверного адреса можно столкнуться с крупными проблемами. Например, если пользователь укажет адрес ядра, то ядро системы «затрет» собственные коды или структуры данных.

Операция s5read() заканчивает работу после прочтения всех данных либо при возникновении ошибки. После этого независимый от файловой системы код освобождает объект vnode (при помощи VOP_RWUNLOCK), увеличивает значение указателя смещения в структуре file на количество прочитанных байтов и завершает свое выполнение. Возвращаемой переменной функции read является общее количество прочитанных байтов. Обычно эта переменная равна запрошенному количеству байтов, кроме случаев достижения конца файла или возникновения ошибки при чтении.

Системный вызов write функционирует сходным образом с несколькими отличиями. Измененные блоки не записываются на диск сразу же. Они хранятся в памяти и переписываются из кэша позже, в зависимости от его эвристических способностей. Кроме этого, вызов write может увеличивать размер файла, что требует выделения новых дисковых блоков, а иногда и блоков косвенной связи для хранения дисковых адресов. Если запись данных производится только во фрагмент блока, ядру все равно необходимо считать блок с диска целиком, изменить его участок и затем перезаписать блок обратно на диск.

9.3.4. Запрос и возврат индексных дескрипторов

Индексный дескриптор остается активным до тех пор, пока его счетчик не станет равным нулю. В этом случае независимый от файловой системы код обратится к операции VOP_INACTIVE, которая освободит индексный дескриптор. В SVR2 свободные дескрипторы помечались как недействующие, таким

образом, они могли быть заново прочитаны с диска при необходимости. Этот подход недостаточно эффективен, из-за чего в более поздних версиях UNIX дескрипторы принято по возможности кэшировать. Если дескриптор становится неактивным, ядро помещает его в список свободных дескрипторов, но оставляет его действующим. Существует возможность найти этот дескриптор при помощи функции iget(), так как он хранится в корректной таблице хэширования до тех пор, пока не будет использован заново.

В каждой файловой системе имеется *таблица индексных дескрипторов* фиксированного размера, ограничивающая общее количество активных дескрипторов в ядре. В SVR3 механизм кэширования дескрипторов использует алгоритм замены последнего, недавно использовавшегося элемента. Ядро освобождает дескрипторы, размещенные в конце списка, и размещает другие, взятые из его начала. Такая схема является стандартной методикой работы кэша и оптимальным решением для кэша дескрипторов из-за того, что именно неактивные дескрипторы используются чаще всего.

Если файл часто используется, то его индексный дескриптор является *прикрепленным* (pinned) к таблице. При доступе к файлу происходит кэширование его страниц в памяти. После того как файл становится неактивным, вероятно нахождение некоторых его страниц в памяти. Такие страницы могут быть обнаружены через *список страниц* объекта vnode (с использованием поля v_page структуры). Система управления страницами производит хэширование также на основе указателя на vnode и смещения страницы в файле. Когда ядро использует индексный дескриптор заново, его предыдущие страницы могут уже не быть идентичными данным файла. Если процессу нужна страница, сформированная из данных этого файла, ядру необходимо будет прочесть ее с диска заново, даже в том случае, если страница уже находится в памяти.

ПРИМЕЧАНИЕ

Говорят, что объект закреплен в памяти, если его нежелательно удалять или освобождать. Объекты со счетчиками ссылок являются прикрепленными до тех пор, пока не освободится последняя ссылка. Процесс также может закрепить часть своего адресного пространства в памяти при помощи системного вызова mlock.

В любом случае для повторного использования лучше применять дескрипторы, не имеющие страниц, кэшированных в памяти. При достижении счетчиком ссылок на vnode нуля, ядро вызывает операцию VOP_INACTIVE для освобождения vnode и его закрытого объекта данных (в рассматриваемой файловой системе — индексного дескриптора). При освобождении дескриптора ядро проверяет список страниц vnode. Освобожденный дескриптор помещается в начало списка, если его список страниц пуст, либо в конец списка, если какие-либо страницы файла до сих пор находятся в памяти. Если индексный дескриптор будет оставаться неактивным, система управления страницами произведет удаление страниц из памяти.

В работе [3] описываются правила размещения и возврата индексных дескрипторов, позволяющие регулировать загруженность системы управлением количеством дескрипторов в памяти. Вместо применения таблицы фиксированного размера файловая система производит размещение дескрипторов динамически при помощи диспетчера памяти ядра. Это позволяет увеличивать или уменьшать количество индексных дескрипторов в системе по мере необходимости. Системному администратору больше не нужно угадывать наиболее подходящее число дескрипторов при установке системы.

Если функция iget() не может обнаружить дескриптор в своей таблице хэширования, то она выбирает первый дескриптор из списка. Если окажется, что он имеет страницы в памяти, функция возвратит такой дескриптор обратно в список и вызовет диспетчер памяти ядра для размещения новой структуры inode. Алгоритм проверки списка свободных дескрипторов можно сделать более универсальным, если производить поиск объектов, не обладающих страницами в памяти, однако реализация, описываемая в этом разделе, является простой, но достаточно эффективной. Ее единственным недостатком является размещение в памяти несколько большего количества дескрипторов, чем это реально требуется.

Эксперименты с применением многопользовательского теста загрузки, работающего в режиме разделения времени, показали, что новый алгоритм сокращает использование *системного времени* (проведенного процессором в режиме ядра) с 16 до 12 %. Хотя описанный алгоритм был изначально реализован для s5fs, он оказался настолько универсальным, что был взят на вооружение и в других файловых системах, в том числе и в FFS.

9.4. Анализ файловой системы s5fs

Файловая система s5fs отличается простотой дизайна. Однако это свойство создает определенные проблемы, связанные с надежностью, производительностью и функциональностью. В этом разделе мы обсудим некоторые отрицательные стороны s5fs, которые стали причиной разработки новой быстрой файловой системы в BSD.

Основным камнем преткновения в отношении надежности s5fs является суперблок. Он содержит важнейшую информацию обо всей файловой системе, в том числе список свободных блоков и информацию о размере списка свободных индексных дескрипторов. В каждой файловой системе имеется только одна копия суперблока. Если она окажется поврежденной, вся файловая система станет непригодной к использованию.

На производительность работы файловой системы негативно влияют сразу же несколько вещей. В s5fs все индексные дескрипторы группируются в начале файловой системы, остальная часть диска обычно занимается блоками данных файлов. Операция доступа к файлу включает в себя чтение его

дескриптора и данных, поэтому такое разделение приводит к длительной задержке при проведении поиска информации на диске, что увеличивает общее время ввода-вывода. Дескрипторы располагаются на диске в произвольном порядке. Система никак не группирует связанные между собой дескрипторы, например относящиеся к файлам одного каталога. Следовательно, вызов операций, производящих доступ ко всем файлам в каталоге (например, `ls -l`) приводит к необходимости чтения блоков из произвольных мест диска.

Размещение блоков на диске также не является оптимальным. При создании новой файловой системы (с использованием программы `mkfs`) производится оптимальная настройка расположения блоков на диске в порядке последовательного размещения. Однако по мере создания и удаления файлов новые блоки помещаются в список уже в произвольном порядке. После некоторого периода использования файловой системы порядок расположения блоков на диске становится абсолютно произвольным. Это уменьшает скорость доступа к файлам, так как логически смежные блоки могут физически находиться далеко друг от друга.

Еще одним немаловажным фактором, влияющим на производительность системы, является размер дискового блока. В SVR2 использовались блоки размером в 512 байт, в SVR3 их длина была увеличена до 1024 байт. Увеличение размера блока позволяет считывать больший объем данных при каждой операции доступа к диску, что увеличивает производительность. В то же время это наращивает потери дискового пространства, так как в каждом файле не используется примерно около половины размера блока. Перечисленные проблемы показывают необходимость существования более гибкого механизма предоставления дискового пространства для размещения файлов.

Файловая система s5fs имеет важные функциональные ограничения. Короткие имена файлов до 14 символов не беспокоили пользователей ранних вариантов UNIX, однако для мощных коммерческих систем такое условие является абсолютно неприемлемым. Некоторые приложения создают имена файлов автоматически, добавляя часто при этом определенные расширения к уже существующим файлам. При использовании всего 14 символов сложно добиться какой-либо степени эффективности. Максимальное число индексных дескрипторов на одну файловую систему в s5fs равняется 65 535, что также ограничивает функциональность системы.

Все вышеназванные ограничения привели к началу разработки новой файловой системы в лабораториях Беркли. Такая система получила название FFS (Fast File System) и впервые была представлена в 4.2BSD[1]. В следующих разделах мы расскажем о ее наиболее важных возможностях.

---

[1] На самом деле файловая система FFS была представлена уже в версии 4.1bBSD, являющейся тестовой реализацией для внутреннего использования в Беркли. Она также была частью 4.1cBSD, еще одной промежуточной версии ОС, распространенной примерно в ста экземплярах.

9.5. Файловая система FFS

В FFS (Fast File System или быстрой файловой системе) были разрешены многие проблемы, существовавшие в s5fs. В следующих разделах вы прочтете рассказ об архитектуре новой файловой системы и о том, как ее новые возможности позволили увеличить надежность, производительность и функциональность ОС в целом. В системе FFS представлены все возможности, имеющиеся в s5fs. Большинство алгоритмов обработки системных вызовов и структур данных ядра остались неизменными. Основные отличия новой файловой системы от s5fs проявляются в разметке дисков, дисковых структурах и методах размещения свободных блоков. В систему FFS для поддержки этих возможностей были добавлены дополнительные системные вызовы.

9.6. Структура жесткого диска

Для того чтобы понять, какие факторы повлияли на увеличение производительности дисковых операций, важно иметь представление о расположении данных на дисках. На рис. 9.7 показан формат обычных жестких дисков, производившихся в начале и середине 80-х годов. Такой диск имеет несколько *дисков-тарелок*, с каждой из которых ассоциируется *головка чтения-записи*[1]. Каждая тарелка содержит несколько дорожек, являющихся концентрическими окружностями. Нумерация дорожек ведется от внешней, которая является нулевой. Каждая дорожка разбивается на *секторы*, которые также нумеруются последовательно. Размер одного сектора (обычно равняющийся 512 байтам) определяет степень детализации при операциях ввода-вывода с диска. Термин «цилиндр» описывает множество дорожек всех тарелок, по одной на тарелку, на одном и том же расстоянии от осевого перпендикуляра к диску. Таким образом, нулевой цилиндр содержит нулевые дорожки каждой тарелки и т. д. В большинстве конструкций все головки передвигаются одновременно. Следовательно, в один момент времени все головки работают с одинаковыми дорожками и одинаковыми номерами секторов каждого диска-тарелки.

Система UNIX видит жесткий диск как одномерный массив блоков. Количество секторов на блок обычно равняется некоторой степени числа 2, но мы для простоты объяснения будем считать, что каждому сектору диска соответствует один блок. Если процессу UNIX необходимо прочесть данные из определенного блока, драйвер устройства произведет преобразование его номера в номер логического сектора и произведет по нему вычисление физических номеров дорожки, головки и сектора. При использовании такой схемы сначала

[1] Обычно использовались обе части тарелки, поэтому для каждой ее стороны существовала отдельная головка.

узнается номер сектора, затем головки и только поток номер цилиндра (дорожки). Таким образом, каждый цилиндр содержит последовательный набор номеров блоков. После вычисления местонахождения необходимого блока драйверу необходимо передвинуть головки диска к соответствующему цилиндру. *Перемещение головки* является наиболее длительной процедурой ввода-вывода. Ее продолжительность напрямую зависит от расстояния, на которое нужно произвести перемещение головки. После установки головки в нужную позицию необходимо дождаться, пока вращающийся диск не окажется в позиции, при которой искомый сектор не окажется под головкой. Требуемое для этого время называется *временем ожидания*. После того как соответствующий сектор попадет под головку диска, можно начать операцию передачи данных. При этом скорость передачи обычно равняется времени передвижения головки через сектор. Таким образом, при оптимизации ввода-вывода необходимо уделить внимание минимизации количества и протяженности операций перемещения головок, а также уменьшению времени ожидания путем улучшения методики размещения блоков на диске.

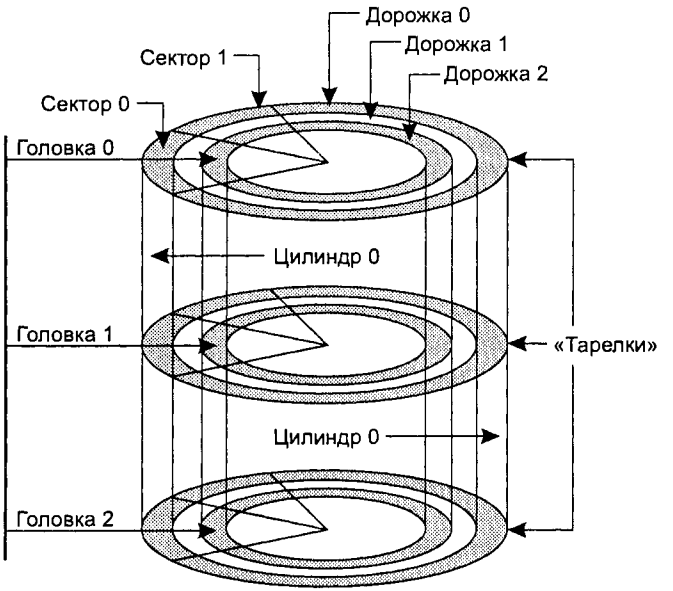


Рис. 9.7. Внутренняя организация жесткого диска

9.7. Хранение данных на диске

Раздел диска занимает несколько последовательных цилиндров на нем. Файловая система располагается на отформатированном разделе. В файловой системе FFS каждый раздел поделен на одну или несколько групп цилинд-

ров, каждая из которых содержит небольшой набор последовательных цилиндров. Это позволяет UNIX хранить относящиеся друг к другу данные в одной группе цилиндров, что уменьшает количество перемещений по диску. Более подробно об этом можно прочесть в разделе 9.7.2.

Информация традиционного суперблока поделена на две структуры. Суперблок FFS содержит информацию о файловой системе в целом. Здесь находятся данные о количестве, размерах и расположении групп цилиндров, размере блоков, общем количестве блоков и индексных дескрипторах и т. д. Данные суперблока остаются неизменными до тех пор, пока файловая система не будет перестроена заново. Более того, каждая группа цилиндров обладает структурой данных, содержащей сводную информацию о группе, в том числе и списки свободных индексных дескрипторов и свободных блоков. Суперблок находится в начале раздела (после загрузочной области). Данные, хранящиеся в суперблоке, очень важны ля системы, следовательно, они должны быть защищены от дисковых ошибок. Каждая группа цилиндров содержит дубликат суперблока. Файловая система FFS располагает такие копии на неодинаковом смещении в каждой группе цилиндров. Поэтому в системе не существует некоей единственной дорожки, цилиндра или диска, объемлющего сразу все копии суперблока. Пространство между началом группы цилиндра и копией суперблока используется для размещения блоков данных (кроме первого цилиндра группы).

9.7.1. Блоки и фрагменты

Как это уже упоминалось ранее, увеличение размера блоков приводит к увеличению производительности путем предоставления возможности передачи большего объема данных в течение одной операции ввода-вывода, но влечет неэкономное использование дискового пространства (в среднем на каждый файл теряется около половины блока). Разработчики FFS постарались придумать некое решение, являющееся компромиссным: они поделили блоки на фрагменты. В FFS все блоки внутри одной файловой системы должны иметь одинаковый размер, но разные файловые системы на одной и той же машине вправе при этом использовать другие размеры блоков. Размер блока может быть больше в два раза или равняться минимальному значению 4 096. В большинстве реализаций верхней границей длины блока определено значение в 8 192 байт. Это намного больше, чем принятые в s5fs блоки «габаритами» в 512 или 1024 байт. Кроме увеличения пропускной способности, поддержка блоков больших размеров позволила адресовать файлы объемом больше, чем 2^{32} байт (4 Гбайт) при помощи блоков косвенной адресации всего двух уровней. В системе FFS не применяются блоки тройной косвенной связи, хотя некоторые ее реализации используют такие блоки для хранения файлов, превышающих по объему значение в 4 Гбайт.

В обычной файловой системе UNIX содержится большое количество файлов, не требующих применения эффективных методик хранения [21]. Для таких файлов размер блока в 4 Кбайт является слишком расточительным. В системе FFS проблема потери пространства была решена при помощи разделения каждого блока на один или несколько фрагментов. Размер фрагмента является фиксированным для каждой файловой системы. Он устанавливается при создании новой системы. Количество фрагментов на один блок можно установить равным 1, 2, 4 или 8, таким образом, нижняя граница размера фрагмента будет равняться 512 байт, что совпадает с размером сектора диска. Каждый фрагмент доступен и адресуется индивидуально. Это требует замены списка свободных блоков битовыми переменными, описывающих состояние связанных с ними фрагментов.

Файл в системе FFS строится из целых дисковых блоков за исключением последнего, который может содержать только один или несколько последовательных фрагментов. Файловый блок должен быть полностью заполнен в пределах одного дискового блока. Даже если два соседних дисковых блока имеют достаточное количество свободных фрагментов, чтобы разместить в них файловый блок, они не могут быть скомбинированы для этой цели. Более того, если последний блок файла содержит более одного фрагмента, то такие фрагменты должны располагаться в одном и том же блоке последовательно.

Приведенная схема уменьшает потери при размещении файлов на дисках, но требует проведения периодического копирования данных файлов. Представим некоторый файл, имеющий последний блок, занимающий единственный фрагмент. Остальные фрагменты блока могут быть ассоциированы с другими файлами. Если размер файла увеличится более чем на один фрагмент, то для его записи необходимо отыскать новый блок, имеющий два последовательных свободных фрагмента, при этом в первый из них копируется информация из предыдущей позиции, а второй фрагмент заполняется новыми данными. Если файл растет небольшими порциями, то его фрагменты необходимо копировать несколько раз, что влияет на производительность. Эта проблема была частично решена в системе FFS тем, что делить на фрагменты можно только «прямые» блоки.

Из всего сказанного следует, что максимальная производительность достигается при записи приложениями целого блока за раз (если это возможно). Различные файловые системы на одной и той же машине могут иметь различные размеры блоков. Для получения атрибутов файловой системы можно использовать системный вызов stat. Одним из атрибутов, возвращаемых stat, является рекомендация по наилучшему размеру элемента для операций ввода-вывода. Применительно к FFS она содержит размер блока. Такая информация используется стандартной библиотекой ввода-вывода, а также приложениями, имеющими собственные средства обработки ввода-вывода.

9.7.2. Правила размещения

Суперблок файловой системы содержит список свободных блоков и список свободных индексных дескрипторов. Добавление или удаление элементов производится в конце этих списков. Их упорядочивание осуществляется только при создании файловой системы: список свободных блоков отсортирован в порядке их смежности, список свободных дескрипторов упорядочен последовательно. Однако через некоторый период времени использования файловой системы оба списка становятся произвольными, то есть управление размещением блоков или индексных дескрипторов в s5fs не осуществляется.

В противоположность оригинальной файловой системе UNIX, в FFS производится группировка связанных между собой данных на диске и оптимизируется последовательный доступ. Система FFS обладает более мощными средствами по управлению распределением дисковых блоков и индексных дескрипторов, а также каталогов. Правила размещения основаны на концепции группы цилиндров и требуют от файловой системы знаний о многих параметрах диска. Эти правила можно вкратце описать несколькими тезисами.

- Система пытается помещать индексные дескрипторы файлов, расположенных в одном каталоге, в одну и ту же группу цилиндров. Существует множество команд (наилучшим примером которых служит `ls -l`), обращающихся ко всем дескрипторам одного каталога за небольшой промежуток времени. Пользователи операционной системы часто имеют дело с ограниченным кругом объектов, работая с одними и теми же файлами, расположенными в одном каталоге (являющемся для пользователя текущим), до того как перейдут в другой каталог.
- Система создает новый каталог в группе цилиндров, отличной от группы родительского каталога. Это помогает распределять данные на диске унифицированно. Процедура размещения выбирает новую группу цилиндров из групп с наибольшим количеством свободных индексных дескрипторов. Из таких групп будет использована та, которая связана с минимальным количеством каталогов.
- Система пытается располагать блоки данных файла в той же группе цилиндров, где находится и его индексный дескриптор, так как наиболее часто доступ осуществляется и к данным, и к дескриптору файла.
- Для защиты от заполнения группы цилиндров одним файлом большого объема система меняет группу цилиндров, если размер файла превышает 48 Кбайт и еще раз по превышению размера в 1 Мбайт. Величина, равная 48 Кбайт, выбрана исходя из того, что элементы «прямого» блока описывают в индексном дескрипторе файла как раз

первые 48 Кбайт (при использовании 4096-байтовых блоков)[1]. Выбор новой группы цилиндров основан на количестве его свободных блоков.

- Система старается по возможности использовать для размещения файла последовательные блоки. При последовательном чтении файла между операцией чтения одного блока и завершением процедуры ввода-вывода ядром имеется некоторая задержка. Поскольку диск не перестает вращаться, за этот период времени его головка может оказаться над сектором, находящимся на некотором расстоянии от того, из которого производилось чтение ранее. Программы, реализующие оптимизацию чтения, пытаются определять количество секторов, которые следует пропустить, чтобы в начале следующей операции чтения головка диска оказалась точно над необходимым сектором. Такое число получило название *фактора задержки вращения* (rotdelay) или *фактора чередования секторов* (interleave).

Система должна уметь балансировать усилия, направленные на установление местонахождения данных и на распределение информации по диску. Если уделять слишком много внимания локализации, все данные могут оказаться записанными в одну группу цилиндров. В максимально вырожденном случае мы имеем на диске всего одну группу, как в файловой системе s5fs. От возникновения подобной ситуации защищает правило создания подкаталогов в новых группах цилиндров и требование разделения файлов большого объема.

Реализация правил размещения в FFS является весьма эффективной при небольшом объеме занятости диска. При увеличении загруженности до 90% преимущества методики становятся практически незаметными. Если на диске мало свободных блоков, системе становится значительно труднее находить блоки, расположенные оптимально. Файловая система FFS поддерживает параметр зарезервированного свободного места, обычно устанавливаемый в значение 10%. Использовать пространство из этого резерва может только суперпользователь.

9.8. Новые возможности FFS

Файловая система имеет отличную от принятой в s5fs организацию хранения данных, поэтому переход на новую систему потребует вывода и восстановления информации, находящейся на дисках. Так как системы s5fs и FFS все равно несовместимы друг с другом, разработчики последней решили добавить в нее несколько нововведений, не имеющихся в s5fs.

[1] Количество «прямых» блоков в массиве дисковых адресов в файловой системе FFS было увеличено с 10 до 12.

Длинные имена файлов

В файловой системе FFS структура каталогов позволяет использовать для файлов имена длиной более 14 символов. Как показано на рис. 9.8, длина записи для каталогов в системе FFS может быть различна. Постоянная часть элемента состоит из полей номера индексного дескриптора, размера переменной части и размера имени файла. После этих параметров записывается имя файла, заканчивающееся нулевыми символами. Остаток области имени заполняется нулевыми байтами до границы, кратной 4. Максимальная длина имени файла не превышает 255 символов. При удалении файла система FFS объединяет освобожденное пространство с предыдущим элементом (рис. 9.8, *б*). Сам каталог поделен на 512-байтовые порции. Элемент каталога не может занимать несколько таких кусков. С целью поддержки создания переносимых кодов в стандартную библиотеку были добавлены функции, позволяющие производить независимый от файловой системы доступ к информации о каталоге (см. раздел 8.2.1).

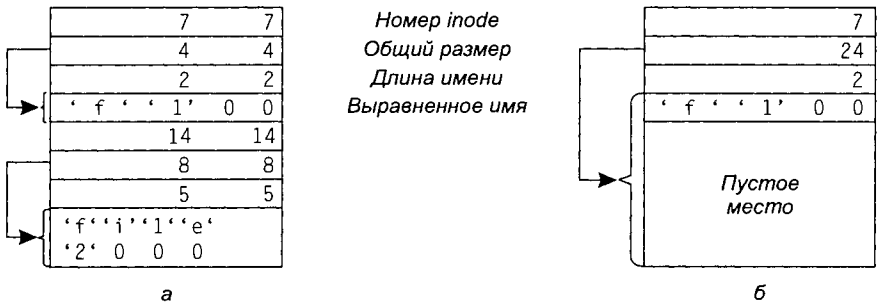


Рис. 9.8. Каталог в файловой системе FFS: *а* — начальное состояние; *б* — после удаления file2

Символические ссылки

Применение символических ссылок (см. раздел 8.4.1) позволяет преодолеть многие ограничения, характерные для жестких ссылок. Символическая ссылка представляет собой файл, указывающий на другой файл, называемый *адресатом*. Для идентификации файла как символической ссылки используется поле type индексного дескриптора. Содержимым такого файла является имя пути к адресату. Такое имя может быть как абсолютным, так и относительным. Процедуры преобразования имен каталогов умеют распознавать и обрабатывают символические ссылки. Если заданное имя является относительным, то процедура интерпретирует его в зависимости от каталога, в котором расположен файл ссылки. Несмотря на то, что обработка символических ссылок происходит незаметно для большинства программ, некоторым утилитам необходимо знать об их существовании. Для этой цели применяется систем-

ный вызов lstat, который не преобразует последнюю символическую ссылку в имени, а также вызов readlink, возвращающий содержимое (то есть адресата) ссылки. Создание ссылок осуществляется при помощи команды symlink.

Другие возможности

В операционную систему 4.2BSD добавлен вызов rename, производящий неделимую операцию переименования файлов или каталогов, которая ранее требовала выполнения последовательности из команд unlink и link. В систему также был привнесен механизм квот, при помощи которого можно ограничить количество файловых ресурсов, доступных каждому пользователю. *Квоты* можно применять по отношению как к индексным дескрипторам, так и к блокам. Ограничение бывает *мягким* (soft) — предупредительным — и *жестким* (hard), требующим вмешательства ядра.

Позже некоторые из перечисленных средств были добавлены в обновленную версию файловой системы s5fs. *В ее реализации для SVR4 поддерживаются символические ссылки и неделимая операция переименования. Однако в этой системе нельзя задать квоты и использовать длинные имена файлов.*

9.9. Анализ файловой системы FFS

Файловая система FFS обладает большей производительностью по сравнению с s5fs. Измерения на машине VAX/750, оснащенной адаптером UNIBUS, показали [11] увеличение пропускной способности чтения с 29 Кбайт/с в s5fs (при размере блока 1 Кбайт) до 221 Кбайт/с в FFS (размер блока — 4 Кбайт, фрагментов — 1 Кбайт). Коэффициент использования CPU при этом увеличился с 11 до 43%. На том же компьютере пропускная способность при записи выросла с 48 до 142 Кбайт/с, а использование процессора — от 29 до 43%.

Еще одним важным фактором, требующим упоминания, являются потери дискового пространства. В среднем на каждом файле теряется около половины блока (в s5fs), либо около половины фрагмента (в FFS). Если размер одного фрагмента FFS равен размеру блока s5fs, то потери одинаковы. Основным преимуществом применения больших блоков является использование меньшего объема диска для указания на все блоки крупного файла. Для размещения такого файла в FFS требуется лишь небольшое количество блоков косвенной связи. В противовес этому системе необходимо больше свободного места для отслеживания свободных блоков и фрагментов. Это свойство практически сводит к нулю преимущества больших блоков. Таким образом, при использовании фрагментов, имеющих размер блока s5fs, потери дискового пространства FFS остаются почти такими же, как и в оригинальной файловой системе UNIX.

Резерв свободного пространства, оставляемый в FFS, можно считать потерянным, так как он не может быть задействован для размещения файлов пользователей. Однако процент потерь в s5fs при использовании блоков размером 1 Кбайт примерно равен аналогичному показателю FFS для варианта 4-килобайтовых блоков и фрагментов длиной 512 байт. Для таких параметров систем объем зарезервированного пространства устанавливается равным 5% общего объема диска.

Структура диска, показанная на рис. 9.7, не является единственно возможной. Современные диски SCSI (Small Computer System Interface, интерфейс малых компьютерных систем) [1] имеют цилиндры неодинакового размера. Такие диски используют преимущества того факта, что их внешние дорожки способны хранить информации больше, чем внутренние, поэтому весь диск разбит на несколько зон. Внутри одной зоны каждая дорожка имеет одинаковое число секторов.

Система FFS не умеет распознавать эти диски, что заставляет пользователей применять фиктивные размеры цилиндров. Для получения дорожек равной величины, принятой в FFS, производители дисков делят все 512-байтовые секторы диска на общее количество дорожек и секторов на дорожку. Такая операция производится из соображений удобства и не изменяет физических характеристик диска. В результате оптимизация вращения в системе FFS не имеет для таких дисков никакого значения и может вообще повлиять на производительность отрицательно. Группировка цилиндров является попрежнему уместной, поскольку блоки соседних цилиндров обычно расположены в близлежащих дорожках диска.

Если рассматривать в целом, файловая система FFS имеет большое количество преимуществ, позволивших ей получить широкое распространение. Разработчики SVR4 включили поддержку FFS в свой продукт. Более того, многие из нововведений файловой системы FFS были добавлены ими в реализацию s5fs для SVR3. Таким образом, система s5fs стала поддерживать символические ссылки, разделяемую и эксклюзивную блокировку файлов, а также вызов rename.

Система FFS стала значительным шагом вперед по сравнению с оригинальной s5fs, но она еще далека от совершенства. Осталось много нереализованных возможностей увеличения производительности файловых систем. Одним из способов оптимизации является связывание нескольких буферов ядра, цепочки которых затем можно считывать или записывать на диск при проведении одной операции. Однако это потребует внесения изменений во всех дисковые драйверы. Увеличить производительность работы также можно, если быстро растущим файлам выделять сразу несколько блоков предварительно, освобождая неиспользованные блоки после закрытия файла. Еще одно важное средство — структурированные и основанные на расширениях файловые системы, будут описаны в главе 11.

Система FFS уже изначально обладала некоторыми дополнительными качествами, не имевшимися в s5fs. В операционной системе 4.3BSD стали использоваться два новых метода кэширования, позволившие ускорить получение имен [14]. Изначально файловая система FFS использовала кэш подстановки имен файлов, основанный на рекомендациях. Эта методика около 70% имен извлекала из кэша. При переносе FFS на SVR4 разработчики сделали коды реализации кэша независимыми от файловой системы, что позволило использовать его как глобальный ресурс, доступный всем файловым системам. Рекомендации были заменены ссылками на кэшируемые файлы. Более подробно о методиках кэширования имен читайте в разделе 8.10.2.

Вторым способом является кэширование каждым процессом смещения последнего компонента каталога, являющегося частью недавно преобразованного полного имени. Если следующее преобразование будет совершено над файлом, находящимся в том же каталоге, то поиск начнется именно с этого компонента каталога. Такой подход удобен при последовательном просмотре содержимого каталога (которое производится в 10–15% случаев поиска). В реализации файловой системы для SVR4 кэшируемое смещение стало храниться в памяти. Это позволило кэшировать смещение всех каталогов вместо хранения данных только об одном каталоге для каждого процесса. С другой стороны, если несколько процессов используют один и тот же каталог одновременно, кэшируемое смещение лучше оставить в покое.

9.10. Временные файловые системы

Многие утилиты и приложения, в особенности компиляторы и оконные системы, используют для хранения промежуточных результатов работы временные файлы. Такие файлы удаляются сразу после выхода из приложения. Следовательно, временные файлы существуют в течение лишь небольшого промежутка времени и не требуют поддержки их сохранности (после краха системы). Ядро использует для отложенной записи буферный кэш, перед сбросом данных на диск временные файлы обычно удаляются. В результате процедура ввода-вывода для таких файлов является весьма быстродействующей и не требует проведения дисковых операций. Однако создание и удаление временных файлов происходит медленно, так как нуждается в выполнении нескольких асинхронных операций доступа к диску для изменения блоков метаданных и информации о каталоге. Синхронное изменение не требуется для обработки временных файлов, поскольку они не хранятся в системе долгосрочно. Однако, в любом случае, для быстрого создания и обеспечения доступа к временным файлам предпочтительно использовать специализированную файловую систему, предназначенную для этих целей.

Проблема временных файлов долгое время решалась посредством RAM-дисков, позволяющих размещать файловые системы полностью в оператив-

ной памяти компьютера. Такие диски поддерживаются при помощи драйверов устройств, эмулирующих обычный диск. При этом реально данные располагаются в физической памяти и могут быть доступны со скоростью, превышающей скорость обращения к обычным дискам. Технология RAM-дисков не требует создания специализированного типа файловой системы. После установки RAM-диска (путем запроса непрерывного участка физической памяти) можно использовать его для добавления обычных файловых систем, таких как s5fs или FFS, при помощи обычных утилит, например newfs. Различия между RAM-диском и обычными дисками системы видны только на уровне драйверов устройств.

Одним из главных недостатком технологии является необходимость выделения больших объемов оперативной памяти компьютера для поддержки RAM-дисков, что представляется не самым лучшим способом использования системных ресурсов. Текущий объем памяти, необходимый для поддержки временных файлов, неодинаков и зависит от утилизации ресурсов системы. Память, выделяемая для RAM-диска, управляется отдельно от памяти ядра, что требует дополнительных операций копирования при изменении метаданных. Становится очевидной потребность создания специализированной файловой системы, эффективно использующей память для поддержки временных файлов. Следующие подразделы главы посвящены описанию двух реализаций таких файловых систем.

9.10.1. Memory File System

Memory File System (*mfs*) разработана в Калифорнийском университете в Беркли [15]. Файловая система полностью построена в виртуальном адресном пространстве процесса, обрабатывающего операцию mount. Такой процесс не завершает работу по возвращению вызова mount, а продолжает оставаться в ядре системы в режиме ожидания запросов ввода-вывода к файловой системе. Каждый объект mfsnode, являющийся зависимой от файловой системы частью vnode, содержит идентификатор PID *процесса монтирования*, который функционирует как сервер ввода-вывода. Для того чтобы произвести ввод-вывод, вызывающему процессу необходимо поместить запрос в очередь, поддерживаемую структурой mfsdata (содержащей закрытые данные файловой системы). После этого будится процесс монтирования, а процесс, желающий произвести ввод-вывод, наоборот переходит в режим сна. Процесс монтирования обрабатывает запрос путем копирования данных из соответствующей области своего адресного пространства и затем возобновляет выполнение первого процесса.

Так как файловая система полностью располагается в виртуальной памяти процесса монтирования, управление ее страницами может осуществляться аналогично механизмам обработки обычной памяти. Страницы файлов mfs находятся, как и другие процессы, в физической памяти. Страницы, не

имеющие активных ссылок, переносятся в область свопинга и могут быть использованы другими процессами при необходимости. Такой подход позволяет поддерживать временные файловые системы, имеющие больший объем по сравнению с количеством физической памяти компьютера.

Хотя система mfs работает намного быстрее, чем обычные дисковые файловые системы, она обладает некоторыми недостатками, большинство из которых является следствием ограничений архитектуры памяти BSD. Использование отдельного процесса для обработки запросов ввода-вывода требует каждый раз двух переключений контекста. Файловая система располагается в отдельном (виртуальном) адресном пространстве, что подразумевает дополнительные операции копирования участков памяти из одного местоположения в другое. Формат файловой системы совпадает с FFS, хотя для нее не имеют смысла такие методики, как группировка цилиндров.

9.10.2. Файловая система tmpfs

Файловая система *tmpfs* разработана корпорацией Sun Microsystems и совмещает в себе возможности интерфейса vnode/vfs и архитектуры VM (виртуальной памяти, Virtual Memory) [7], предлагая эффективный механизм работы с временными файлами. Средства VM, упомянутые в этом разделе, будут более подробно описаны в главе 14.

Файловая система tmpfs полностью реализована на уровне ядра и не требует выделения для ее обслуживания отдельного серверного процесса. Все метаданные файлов хранятся в памяти, не разбиваемой на страницы, а выделяемой динамически из кучи ядра. Блоки данных располагаются постранично и представлены в ядре при помощи средства *анонимных страниц* (anonymous page) подсистемы VM. Данные о размещении каждой страницы хранятся в *анонимном объекте* (структуре `anon`), описывающей расположение страницы в физической памяти или области свопинга. Зависящий от файловой системы объект `tmpnode` содержит указатель на *анонимную карту* файла (структуру `anon_map`). Карта является массивом указателей на анонимные объекты каждой страницы файла. Связь между структурами показана на рис. 9.9. Так как страницы сами по себе размещаются в страничной памяти, они могут быть перенесены в область свопинга страничной подсистемой и использованы как физическая память другими процессами, так же как и в файловой системе mfs.

Анонимные объекты и карты используются также подсистемой VM для описания страниц в адресном пространстве процесса. Это позволяет процессам отображать файл tmpfs в свое адресное пространство напрямую, востребуя для этой цели интерфейс системного вызова `mmap`. Такое отображение реализовано при помощи совместного использования структуры `anon_map` файлом и процессом. Соответственно процесс обладает прямым доступом к такому файлу без необходимости копирования его в собственное адресное пространство.

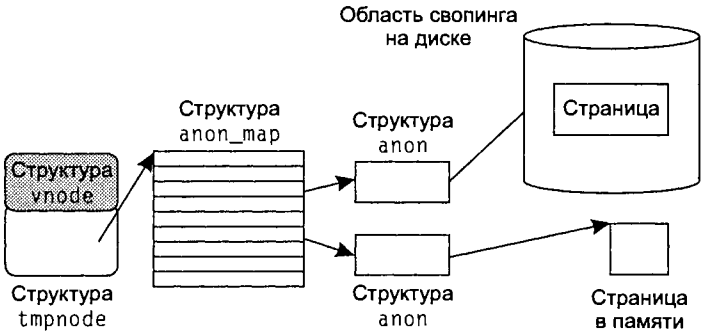


Рис. 9.9. Размещение страниц в файловой системе tmpfs

Файловая система tmpfs решила некоторые проблемы, имеющиеся в mfs. Система не использует выделенный сервер ввода-вывода, что помогает избавиться от необходимости лишних контекстных переключений. Хранение метаданных в куче дает возможность не производить операции копирования из одного участка памяти в другой, а также защищает от некоторого избытка дискового ввода-вывода. Поддержка карт памяти позволяет осуществлять быстрый прямой доступ к данным файла.

Еще одна разновидность временных файловых систем описана в [17]. Система поддерживает дополнительную опцию задержки delay вызова mount, устанавливающую соответствующий флаг в закрытом объекте vfs_data для ufs. Большинство процедур ufs, которые производят синхронные изменения на диске (обычно после изменения метаданных), были усовершенствованы. Теперь перед началом действий они проверяют наличие флага. Если флаг установлен, процедуры откладывают запись, маркируя при этом буфер как dirty («грязный»). Такой подход имеет несколько преимуществ. Система не использует отдельный RAM-диск, что экономит адресное пространство и не требует копирования блоков в память. Показатели производительности этого решения просто впечатляют. Его основным недостатком является необходимость переписывания процедур ufs, а не просто внесения добавлений в ядро без правки исходного кода. В системах, использующих несколько типов файловых систем, каждая реализация должна быть соответственно изменена для поддержки опции задержки.

9.11. Файловые системы для специальных целей

Изначальной целью интерфейса vnode/vfs была поддержка на одной машине сразу нескольких файловых систем, как локальных (например, s5fs или FFS), так и удаленных (например, RFS или NFS, о них см. подробнее в главе 10). Через некоторое время после появления этого интерфейса было разработано несколько специализированных файловых систем, использующих его мощ-

ность и универсальность. Некоторые из этих систем стали частью стандартной версии SVR4, другие поставлялись отдельно как компоненты, добавляемые производителями. В следующих подразделах будут рассмотрены некоторые интересные примеры специализированных файловых систем.

9.11.1. Файловая система specfs

Файловая система specfs предлагает стандартный интерфейс взаимодействия с файлами устройств. Она невидима на прикладном уровне и не может быть смонтирована как обычная система, а также используется как общий интерфейс для любой файловой системы, поддерживающей специальные файлы. Основной целью specfs является перехват вызовов ввода-вывода и трансляция их в вызовы процедур драйвера соответствующего устройства. Эта задача кажется простой, так как идентификация файла устройства происходит по полю v_type, а минимальный и максимальный номера устройства указываются в поле v_rdev. Независимый от файловой системы код должен обладать возможностью вызывать драйверы устройств напрямую, используя переключения блочных и символьных устройств.

Проблемы возникают в том случае, когда несколько файлов относятся к одному и тому же устройству. Если несколько пользователей пытаются одновременно получить доступ к устройству при помощи отличающихся друг от друга имен, ядру системы необходимо синхронизировать такой доступ. Для блочных устройств оно должно отследить существование копий блоков в буферном кэше. Становится очевидным, что ядру системы требуется обладать информацией, какие объекты vnode в текущий момент времени представляют одно и то же устройство.

На уровне specfs для каждого файла устройства создается *теневой объект vnode* (shadow vnode), представляющий собой зависимую от файловой системы структуру данных snode. Теперь все операции поиска файла устройства будут возвращать ссылку не на *действительный*, а на теневой объект vnode. Если это необходимо, действительный vnode может быть получен при помощи операции vop_realvp. Структура snode содержит поле s_commonvp, указывающее на *общий объект vnode* (ассоциированный с другой snode) для этого устройства. Для каждого устройства существует только один общий объект vnode. Он идентифицируется номером и типом устройства. На такой объект могут ссылаться сразу несколько объектов snode. Все операции, требующие синхронизации, а также ввод-вывод блочных устройств производятся через общий объект vnode. Более подробно о файловой системе specfs можно прочесть в разделе 16.4.

9.11.2. Файловая система /proc

Файловая система /proc [5] предлагает элегантный и мощный интерфейс доступа к адресному пространству любого процесса. Система изначально раз-

рабатывалась как средство отладки, которое должно было заменить собой ptrace, но переродилось в общий интерфейс к модели процессов. Файловая система /proc позволяет приложениям читать и изменять данные в адресном пространстве других процессов, а также управлять им, используя стандартный интерфейс файловой системы и системных вызовов. Из этого следует, что управление доступом к адресному пространству осуществляется при помощи обычных привилегий на чтение, запись и выполнение. По умолчанию запись и чтение файлов /proc разрешены только для их владельцев.

В ранних реализациях системы каждый процесс представлялся в виде файла, расположенного в каталоге /proc. Имя файла определялось десятичным значением идентификатора процесса, а его размер соответствовал его пользовательскому адресному пространству. Доступ к любому адресу осуществлялся путем открытия соответствующего файла /proc и применения системных вызовов lseek, read и write. Различные операции управления процессом выполнялись при помощи набора команд ioctl в отношении файла.

Позже реализация /proc претерпела большие изменения. Здесь описывается вариант, представленный в SVR4.2. В этой операционной системе каждый процесс представлен в виде отдельного каталога, расположенного внутри /proc. Имя каталога является десятичным значением идентификатора процесса. В каждом таком каталоге имеются следующие файлы и подкаталоги:

status	Файл (доступный только для чтения) содержит информацию о статусе процесса. Его формат определен структурой pstatus, которая хранит идентификатор процесса, идентификаторы группы и сеанса, размеры и местонахождение стека и кучи, а также некоторые другие данные
psinfo	Файл (доступный только для чтения) содержит информацию, необходимую для команды ps(1). Его формат определен структурой psinfo, включающей в себя некоторые поля файла статуса, а также дополнительную информацию, в том числе размер образа и идентификатор устройства контролируемого терминала
ctl	Файл (доступный только для записи) позволяет пользователям производить операции управления процессом посредством записи форматированных сообщений в этот файл. Некоторые операции управления будут подробнее описаны ниже
map	Файл (доступный только для чтения) представляет собой карту виртуальных адресов процесса. Он содержит массив структур prmap, каждый элемент которого описывает один непрерывный диапазон адресов процесса. Карты виртуальных адресов будут подробно рассмотрены в разделе 14.4 3
as	Файл (доступный как для чтения, так и для записи) является картой виртуального адресного пространства процесса. Любой адрес может быть доступен при помощи операции lseek, осуществляющей позиционирование в файле, и последующего вызова read или write
sigact	Файл (доступный только для чтения) содержит информацию об обработке сигналов. В нем находится массив структур sigaction (см. раздел 4.5), по одной для каждого сигнала

cred	Файл (доступный только для чтения) содержит пользовательские полномочия процесса. Его формат определяется структурой prcred
object	Этот каталог соотносит по одному файлу на каждый объект, отображенный в адресном пространстве процесса (см. раздел 14.2). Пользователь может получить файловый дескриптор для такого объекта путем открытия соответствующего файла
lwp	Каталог содержит по одному подкаталогу для каждого LWP (см. главу 3) процесса. В каждом подкаталоге находится по три файла: lwpstatus, lwpsinfo и lwpctl, предлагающих набор статусных и управляющих операций, сходных описываемым в файлах status, psinfo и ctl соответственно

Важно упомянуть о том, что перечисленные файлы не являются обычными файлами, физически расположенными на дисках. Они используются только в качестве интерфейса доступа к процессу. Операции над такими файлами преобразуются системой /proc в соответствующие действия над процессом либо его адресным пространством. Один и тот же файл /proc могут открывать одновременно сразу несколько пользователей. Рекомендательная блокировка при открытии файлов as, ctl или lwpctl устанавливается флагом O_EXCL. Файлы ctl и lwpctl позволяют производить некоторые статусные и управляющие операции, в том числе:

PCSTOP	Остановку всех LWP процесса
PCWSTOP	Ожидание остановки всех LWP процесса
PCRUN	Возобновление выполнения приостановленных LWP. При помощи необязательных флагов могут быть заданы некоторые дополнительные действия, как, например, PRCSIG для сброса текущего сигнала или PRSTEP для пошагового выполнения процесса
PCKILL	Отправку процессу указанного сигнала
PCSENTRY	Указание LWP на необходимость остановки перед началом выполнения определенных системных вызовов
PCSEXIT	Указание LWP на необходимость остановки после выхода из определенных системных вызовов

В системе не предусмотрена поддержка контрольных точек. Но они могут быть реализованы при помощи вызова write, копирующего инструкцию контрольной точки в любое место текстового сегмента. Большинство систем обладают определенными инструкциями контрольных точек. С другой стороны, можно применять инструкции, вызывающие прерывания ядра.

Интерфейс файловой системы /proc обладает механизмами отслеживания потомков процесса. Отладчик может установить процессу флаг inherit-on-fork, после чего отслеживать выходы из вызовов fork и vfork. Флаг заставляет родительский и дочерний процессы приостановить работу после возврата fork. Приостановка предка дает отладчику возможность проверить величину,

возвращаемую fork, для определения идентификатора PID потомка и последующего открытия его файлов /proc. Отладчик обладает полным контролем над процессом-потомком, так как тот приостанавливает выполнение до выхода из fork.

Появление интерфейса /proc позволило разработать несколько сложных отладчиков и профилировщиков. Например, файловая система /proc дала возможность dbx присоединяться (и отключаться) прямо к выполняющимся программам. Реализация работает корректно с файлами /proc, расположенными на удаленных машинах и доступными через RFS [14]. Система разрешает приложениям производить отладку и управлять процессами одинаково, независимо от того, являются ли они локальными или удаленными. С появлением /proc системный вызов ptrace стал устаревшим и ненужным. Позже некоторые команды, например ps, были переписаны заново с использованием возможностей /proc. Было разработано генерализированное средство *точек наблюдения* (watchpoints), основанное на возможности подсистемы VM управлять защитой страниц памяти динамически.

9.11.3. Процессорная файловая система

Процессорная файловая система предлагает интерфейс доступа к отдельным процессорам на многопроцессорных машинах. Система монтируется в каталоге /system/processor, в котором для каждого процессора создается по одному файлу. Имя файла является десятичным представлением номера процессора. Каждый файл имеет фиксированный размер и доступен только для чтения. Поля данных, хранящихся в нем, содержат следующую информацию:

- статус процессора, работающий или отключенный;
- тип процессора;
- скорость процессора в мегагерцах;
- размер кэша в килобайтах;
- наличие блока операций с плавающей точкой (сопроцессора);
- драйверы, относящиеся к данному процессору;
- время последнего изменения статуса процессора.

Кроме этого файловая система содержит файл ctl, являющийся доступным только суперпользователю (для записи). Запись в этот файл позволяет управлять конкретными процессорами, например, изменять статус с работающего на отключенный.

Процессорная файловая система является частью многопроцессорной реализации SVR4.2. В будущем она может быть расширена до поддержки таких понятий, как наборы процессоров и легковесные процессы путем их ассоциации с дополнительными файлами в пространстве имен.

9.11.4. Translucent File System

Translucent File System, TFS [8] была разработана корпорацией Sun Microsystems для удовлетворения потребностей по оптимальному построению информационной структуры программных приложений. Система предлагает механизмы усовершенствованных версий и управления сборкой, а также поддерживает инструмент управления настройками Network Software Environment (NSE). Система TFS является стандартным компонентом SunOS.

Разработчики крупных программных сред сталкиваются со стандартным кругом потребностей. Часто пользователи поддерживают собственную иерархию файлов, так как им необходимо вносить изменения в определенные файлы. При этом нет нужды в поддержке закрытых копий неизменяемых файлов, однако их необходимо защитить от возможности внесения изменений другими разработчиками. Более того, программная среда должна содержать средства контроля версий, позволяющие пользователю выбрать, к какой версии системы ему необходимо получить доступ.

Такие возможности предоставляет файловая система TFS, задействующая технологию копирования при записи. Файлы из разделяемой иерархии копируются в закрытую пользовательскую иерархию только в том случае, если они были модифицированы. Реализация этой возможности базируется на нескольких уровнях построения каталога TFS, каждый из которых представляет собой физический каталог. Уровни соединены между собой при помощи скрытых файлов, называемых *ссылками поиска* (searchlinks). Они содержат имя каталога следующего уровня. Каждый уровень является как бы одной редакцией каталога, а верхний уровень содержит последнюю модификацию.

Видимые файлы каталога TFS являются объединением файлов всех уровней. По умолчанию доступна последняя модификация файла (поиск по уровням производится с верхнего уровня к нижнему). Если необходима более ранняя версия, то для обращения к ней нужно следовать по цепочке ссылок поиска, что можно осуществить и на прикладном уровне, так как каждый уровень физически представлен отдельным каталогом. Реализация копирования при записи сделана при помощи установки всех уровней, кроме верхнего, в режим «только для чтения». Для изменения файлов нижних уровней необходимо произвести их копирование в верхний уровень.

Производительность файловой системы FFS зависит от того, как много уровней нужно просмотреть операции поиска (общее количество уровней типовой среды может быть очень большим). Разработчики TFS решили эту проблему при помощи агрессивного использования кэша подстановки имен файлов. Система также предлагает средства различения уровней в зависимости от используемых аппаратных архитектур, так как объекты файлов неодинаковы для каждого варианта. Для доступа к файлам TFS нет нужды вносить изменения в прикладные приложения. Чтобы воспользоваться преимуще-

ствами TFS, достаточно произвести ее начальную настройку системным администратором. Изначально файловая система TFS разрабатывалась для работы на сервере NFS, но впоследствии была изменена для непосредственного использования интерфейса vnode/vfs.

В разделе 11.12.1 будет описана файловая система union mount 4.4BSD, имеющая сходные возможности, но построенная на стековом интерфейсе vnode/vfs операционной системы 4.4BSD.

9.12. Буферный кэш в ранних версиях UNIX

Дисковый ввод-вывод является узким местом любой системы. На прочтение блока размером 512 байт необходимо затратить порядка нескольких миллисекунд. При этом для копирования такого же объема данных из одного участка памяти в другой достаточно всего лишь нескольких микросекунд. Скорость двух операций разнится в тысячу раз. Если каждая операция ввода-вывода будет требовать доступа к диску, то система станет работать неудовлетворительно медленно. Становится очевидной необходимость минимизации объема операций ввода-вывода. В системах UNIX это делается при помощи кэширования недавно использованных дисковых блоков в оперативной памяти.

В традиционных реализациях UNIX для этой цели отводится определенная область памяти, называемая буферным кэшем, применяемая для кэширования блоков, доступ к которым был произведен файловой системой. Система виртуальной памяти кэширует тексты процесса и страницы данных отдельно. В современных вариантах UNIX, таких как SVR4 или SunOS (версии 4 и выше), буферный кэш встроен в страничную подсистему. В этом разделе будет описан традиционный вариант кэша. О современной реализации можно прочесть в разделе 14.8.

Буферный кэш состоит из буферов данных, каждый из которых имеет размер, достаточный для хранения одного дискового блока. Системы, основанные на BSD, поддерживают буферы изменяемых размеров, так как различные файловые системы на одной и той же машине могут иметь неодинаковые размеры блоков и их фрагментов. С каждым буфером кэш ассоциирует заголовок, в котором хранится информация об именовании, синхронизации и управлении кэшем. Размер кэша составляет обычно 10% от физического объема памяти.

Постоянным местонахождением данных кэша является вспомогательная память (backing store). Кэш может обрабатывать данных из нескольких различных областей вспомогательной памяти. Для буферного кэша такая память представляет собой файловые системы на диске. Если машина подключена к сети, вспомогательная память способна включать в себя файлы на удаленных узлах.

Кэш может быть со сквозной или отложенной записью. Кэш *сквозной записи* производит сброс данных во вспомогательную память незамедлительно при их модификации. Такой подход имеет несколько преимуществ. Данные во вспомогательной памяти всегда соответствуют текущим (разве что кроме последней операции записи), следовательно, при использовании такого метода кэширования не существует проблемы потери информации или повреждения файловой системы в результате краха системы. Управление таким кэшем является простым, что делает его приемлемым средством аппаратного кэширования, например буферизации на жестких дисках.

Сквозная запись не подходит для буферного кэша, поскольку она противоречит основной цели увеличения производительности. Запись составляет примерно треть всех операций файлового ввода-вывода, которые являются весьма длительными: производится перезапись данных или файл удаляется после «лишних» минут, потраченных на запись ранее. Такой подход приводит к большому количеству избыточных операций записи, что качественно снижает быстродействие системы.

Именно по этим причинам буферный кэш в системах UNIX использует преимущественно метод *отложенной записи*[1]. Измененные блоки помечаются как «грязные» (dirty), а запись их на диск осуществляется позднее. Данный подход позволяет избавиться от большого количества операций записи, а также упорядочить их в определенном порядке, позволяющем увеличить производительность дисков. Однако задержка записи может стать причиной повреждений файловой системы в результате краха. Более подробно эта проблема будет рассмотрена в разделе 9.12.5.

9.12.1. Основные операции

Перед началом дисковой операции чтения или записи процесс осуществляет поиск блока в буферном кэше. Чтобы увеличить эффективность такой процедуры, подсистема кэша поддерживает набор таблиц хэширования, основанных на имеющихся устройствах и номерах блоков. Если блок не окажется в кэше, он будет прочтен с диска (кроме операции перезаписи блока целиком). Ядро запросит для него буфер в кэше, ассоциирует буфер с этим блоком и затем инициализирует чтение (при необходимости). При изменении блока ядро производит все модификации в копии блока, находящейся в буфере, после чего помечает такой блок как «грязный», устанавливая соответствующий флаг в заголовке буфера. При необходимости очистки блока при повторном использовании ядро произведет запись данных, содержащихся в нем, на диск.

При использовании буфера необходимо предварительно блокировать его. Блокировка происходит либо перед инициализацией дискового ввода-выво-

[1] Конечно, метаданные сбрасываются на диск синхронно (см. раздел 9.12.5).

да, либо при необходимости чтения из буфера или записи в него. Если буфер уже окажется заблокированным, пытающийся осуществить доступ к нему процесс будет переведен в режим сна до отмены блокировки. Обработчик дисковых прерываний также может пытаться получить доступ к буферу, поэтому на период получения буфера ядро отключает все дисковые прерывания.

Незаблокированный буфер хранится в списке свободных буферов. Список формируется по принципу последнего недавно использовавшегося элемента (LRU). Если ядру системы необходим свободный буфер, будет выбран тот, который не использовался дольше остальных. Это правило основано на том факте, что типовая система работает преимущественно по принципу *локальности ссылок*: наиболее часто используются данные, к которым был доступ осуществлен совсем недавно в отличие от «старых» данных, хранящихся неиспользуемыми долгий период времени. При освобождении буфера он помещается в конец списка, после чего он становится последним недавно использованным. С течением времени буфер постепенно перемещается в направлении начала списка. При достижении начала буфер становится наиболее давно использовавшимся и будет предоставлен любому процессу, запросившему свободный буфер.

При использовании описанной методики могут возникать некоторые исключительные ситуации. Первая из них приводит к повреждению буферов, что происходит в результате ошибок ввода-вывода либо при удалении или обрезании части файла. Такие буферы будут помещены в начало очереди незамедлительно, так как они гарантированно не будут больше запрошены тем же процессом.

Вторая проблема имеет место, если «грязный» буфер успевает достичь начала списка до того, как буфер будет удален из него и переведен в очередь записи дискового драйвера. После завершения операции записи такой буфер будет помечен как чистый и возвращен в список свободных буферов. Так как очищенный буфер уже достиг начала списка до момента доступа к нему, он будет возвращен не в конец списка (как обычно), а в его начало.

9.12.2. Заголовки буфера

Каждый буфер представлен своим заголовком. Ядро использует такие заголовки для идентификации и отслеживания буфера, синхронизации доступа к нему, а также для управления кэшем. Заголовок также является интерфейсом к дисковому драйверу. Если необходимо прочесть данные в буфер с диска или осуществить запись, ядро загружает параметры операции ввода-вывода в заголовок и передает этот заголовок дисковому драйверу. Некоторые важные поля структуры buf, представляющей собой заголовок буфера, показаны в табл. 9.2.

Таблица 9.2. Поля структуры buf

Поле	Описание
int b_flags	Статусные флаги
struct buf *b_forw, *b_back	Указатели хранения buf в таблице хэширования
struct buf *av_forw, *av_back	Указатели хранения buf в очереди свободных буферов
caddr_t b_addr	Указатели на данные
dev_t b_edev	Номер устройства
daddr_t b_blkno	Номер блока в устройстве
int b_error	Статус ошибки ввода-вывода
unsigned b_resid	Количество оставшихся передаваемых байтов

Поле b_flags является битовой маской, задающей несколько флагов. Ядро использует флаги B_BUSY и B_WANTED для синхронизации доступа к буферу. Флаг B_DELWRI помечает буфер как «грязный». Флаги B_READ, B_WRITE, B_ASYNC, B_DONE и B_ERROR предназначены для дискового драйвера. Флаг B_AGE указывает на «старый» буфер, являющийся идеальным кандидатом на повторное использование.

9.12.3. Преимущества

Основной целью использования буферного кэша является уменьшение дискового обмена и предупреждение лишних операций ввода-вывода, а также увеличение эффективности работы дисковой подсистемы. Правильно настроенное кэширование позволяет увеличить производительность примерно на 90% [18]. Технология обладает и другими весомыми преимуществами. Буферный кэш синхронизирует доступ к дисковым блокам путем установки флагов locked и wanted. Если два процесса попытаются одновременно запросить один и тот же блок, заблокировать его сможет только один из процессов. Буферный кэш является модульным интерфейсом взаимодействия между дисковым драйвером и другими частями ядра. Никакая другая подсистема ядра не имеет доступа к драйверу. Интерфейс взаимодействия основан на полях заголовка буфера. Более того, буферный кэш изолирует остальную часть ядра от необходимости приведения ее к требованиям подсистемы ввода-вывода, так как буферы сами по себе организованы постранично. По этому при запросах дискового ввода-вывода по отношению к неупорядоченным адресам у ядра не возникает проблем.

9.12.4. Недостатки

Несмотря на очевидные преимущества, буферный кэш не свободен и от некоторых недостатков. Во-первых, отложенная запись, используемая подсистемой кэширования, может стать причиной потери данных в случае краха ОС.

При этом целостность дисков может оказаться нарушенной. Более подробно читайте об этой проблеме в разделе 9.12.5. Во-вторых, несмотря на уменьшение общего числа операций доступа к диску, что увеличивает производительность системы, данные приходится копировать дважды: сначала с диска в буфер и только потом в пользовательское адресное пространство. Второе действие занимает значительно меньше времени, чем первое. Уменьшение общего количества операций за счет кэширования обычно компенсирует затраты на копирование данных из одной области данных в другую, но становится весьма весомым фактором задержки при последовательном чтении или записи в файл большого объема. Такая операция приводит к возникновению ситуации, получившей название *«затирания» кэша* (cache wiping). При необходимости прочтения целиком большого файла, к которому в дальнейшем не потребуется доступ, возникает эффект изменения содержания кэша. Так как все блоки файла нужно прочесть за короткий период времени, они постепенно займут все буферы кэша, сбрасывая тем самым его предыдущее содержимое. Это приводит к некоторому временному промежутку неэффективности кэша до тех пор, пока он снова не заполнится более полезными данными, что уменьшает производительность системы. Защититься от проблемы затирания кэша можно на прикладном уровне. Например, в файловой системе Veritas (VxFS) позволяется пользователю указать, каким именно образом следует производить доступ к файлам. При помощи этого средства можно запретить кэширование больших файлов и запросить систему передавать данные с диска напрямую в пользовательское адресное пространство, минуя промежуточный буфер.

9.12.5. Целостность файловой системы

Основной проблемой, связанной с применением кэша, является несовпадение данных на диске с их текущим состоянием. Если система работает нормально, это не является критичным, так как ядро использует блоки, содержащие «свежие» копии дисковых данных. Проблема возникает в случае краха системы, при котором могут быть потеряны внесенные изменения. Такими данными могут оказаться как блоки файлов, так и их метаданные. В системах UNIX принято обрабатывать потерю этих двух типов информации по-разному.

С точки зрения операционной системы потеря данных при записи на диск является не столь катастрофичной, как для пользователя, поскольку при этом не нарушается целостность файловой системы. Если производить все операции записи на диск синхронно, снижение производительности системы станет значительным, поэтому по умолчанию используется технология отложенной записи. Существуют несколько способов, заставляющих ядро системы сбросить данные на диск. Системный вызов sync выполняет запись на диск содержимого всех «грязных» буферов. При этом, однако, ядро не ждет завершения процедуры записи, поэтому не существует гарантии того, что конкретный блок будет являться актуальным после завершения работы sync.

Пользователь также может открывать файл в синхронном режиме, после чего все операции записи в этот файл будут производиться синхронно. Некоторые реализации системы используют демон обновления (в SVR4 это процесс под названием fsflush), который периодически (по умолчанию каждые 30 секунд) вызывает sync для очистки кэша.

При потере метаданных файловая система теряет свою целостность. Изменения в метаданных производят многие файловые операции; если при этом какие-либо из модификаций не будут сохранены на диск при крахе ОС, файловая система может оказаться поврежденной. Например, при добавлении ссылки к файлу происходит запись элемента нового имени в соответствующий каталог и увеличение *счетчика ссылок* в индексном дескрипторе. Представьте, что крах системы произошел как раз в тот момент, когда были записаны изменения в каталоге, но внесенные в индексный дескриптор данные не сохранены на диск. После перезагрузки системы мы увидим два элемента каталогов, указывающих на файл, имеющий всего одну ссылку. Если этот файл будет удален, то его индексный дескриптор и занимаемые блоки будут освобождены, так как значение счетчика ссылок уменьшилось теперь до значения «ноль». Второе вхождение будет указывать на незанятый (или указывающий уже на другой файл) индексный дескриптор. Возникновение подобных ситуаций в системе необходимо уметь предупреждать.

Существуют две методики защиты от повреждения файловой системы, применяемые в UNIX. Ядро выбирает определенный порядок записи метаданных, уменьшающий влияние крахов системы. Представьте, что получится, если в предыдущем примере поменять порядок операций записи: после краха системы дескриптор стал уже обновленным, но не произошла запись в каталог. После перезагрузки такой файл будет иметь дополнительную ссылку, но при этом оригинальное вхождение каталога окажется правильным и доступ к файлу может быть продолжен без каких-либо проблем. Если файл будет удален, пропадет элемент каталога, но не освободятся блоки и индексный дескриптор, так как на этот файл останется еще одна ссылка. Такой подход, конечно, не защищает от повреждений, но проводит к меньшей степени вреда, чем в исходном примере.

Порядок записи метаданных должен определяться очень внимательно. Проблема порядка выполнения остается по-прежнему, потому что дисковый драйвер обслуживает запросы не в порядке их получения. Чтобы указать порядок выполнения запросов, у ядра есть единственное средство — сделать операции синхронными. Следовательно, в описанном примере ядро произведет запись индексного дескриптора на диск, подождет завершения этой операции и затем вызовет запись в каталог. Ядро использует синхронную запись метаданных во многих операциях, требующих изменения нескольких взаимосвязанных объектов [2].

Вторым методом борьбы с повреждением файловой системы является применение утилиты fsck [4], [10]. Эта программа проводит диагностику файло-

вой системы, находит нарушения целостности и восстанавливает ее, если это возможно. Если метод исправления не очевиден, программа попросит пользователя выбрать наиболее приемлемый. По умолчанию системный администратор выполняет fsck каждый раз при перезагрузке системы. Программа также может быть запущена в любой момент времени по требованию. Утилита fsck использует прямой доступ к дисковому драйверу для доступа к файловой системе (см. более подробно в разделе 11.2.4).

8.13. Заключение

Интерфейс vnode/vfs позволяет нескольким различным файловым системам сосуществовать на одной машине. В этой главе были описаны реализации нескольких файловых систем. Сначала были рассмотрены две наиболее популярные локальные системы, s5fs и FFS, после чего вы увидели примеры специализированных файловых систем, использующих преимущества некоторых свойств vnode/vfs для предоставления пользователю полезных средств. В конце главы приведено описание буферного кэша, являющегося общим ресурсом, используемым всеми файловыми системами.

В следующих главах будут описаны другие файловые системы. Мы поговорим о распределенных системах, таких как NFS, RFS, AFS и DFS. Глава 11 будет посвящена специфичным и экспериментальным файловым системам, использующим для увеличения функциональности и производительности такие технологии, как поддержка журналов.

9.14. Упражнения

1. Почему системы s5fs и FFS обладают фиксированным значением дисковых индексных дескрипторов на каждую файловую систему?
2. Почему индексный дескриптор отделен от элемента каталога файла?
3. Какие преимущества и недостатки вы видите при размещении каждого файла на диске последовательно? Для каких типов приложений наиболее подходит файловая система, основанная на этом подходе?
4. Что произойдет, если в результате дисковой ошибки уничтожится суперблок s5fs?
5. Какие существуют преимущества динамического запроса и освобождения индексных дескрипторов?
6. Система, использующая кэш подстановки имен, основанный на ссылках, может оказаться без свободных индексных дескрипторов просто потому, что кэш ссылается на дескрипторы, которые уже являются свободными. Какие действия производит файловая система в случае возникновения такой ситуации?

7. Поиск имен в каталоге большого размера может быть совершенно неэффективным в обычных файловых системах, таких как s5fs или FFS. Исследуйте возможность организации каталога в виде таблицы хэширования. Должна ли такая таблица находиться только в памяти или частично храниться на постоянных носителях? Будет ли обсуждаемая технология простой копией функциональности кэша подстановки имен?
8. В 4.4BSD кэш подстановки имен поддерживает вхождения неудачных операций поиска. Какими преимуществами обладает кэширование такой информации? С какими проблемами столкнется система, использующая описанный подход?
9. Почему системному вызову write иногда необходимо сначала прочесть данные блока с диска?
10. Почему в системе FFS принято размещать каждый новый каталог в группе цилиндров, отличной от занимаемой его родительским каталогом?
11. В каких ситуациях фактор чередования секторов ухудшает производительность системы?
12. Почему алгоритмы размещения данных на дисках системы FFS могут отрицательно повлиять на производительность при использовании современных SCSI-дисков?
13. С какой целью в FFS предусмотрен резерв свободного пространства?
14. Представьте некую файловую систему, в которой данные небольших файлов хранятся прямо в индексном дескрипторе вместо использования отдельного дискового блока. Какими преимуществами и недостатками обладает такой подход?
15. Опишите преимущества применения специализированной файловой системы для обработки временных файлов.
16. Какие действия производятся файловой системой с целью уменьшения затирания кэша?
17. Какие преимущества дает отделение подсистемы кэширования буферов от подсистемы памяти? Какие при этом возникают недостатки?

9.15. Дополнительная литература

1. American National Standard for Information Systems, «Small Computer Systems Interface-2 (SCSI-2)», X3.131–199X, Feb. 1992.
2. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
3. Barkley, R. E., and Lee, T. P., «A Dynamic File System Inode Allocation and Reclaim Policy», Proceedings of the Winter 1990 USENIX Technical Conference, Jan. 1990, pp. 1–9.

4. Bina, E. J., and Emrath, P. A, «A Faster fsck for BSD UNIX», Proceedings of the Winter 1989 USENIX Technical Conference, Jan. 1989, pp. 173–185.
5. Faulkner, R. and Gomes, R., «The Process File System and Process Model in UNIX System V», Proceedings of the 1991 Winter USENIX Conference, Jan. 1991, pp. 243–252.
6. Gaede, S., «A Scaling Technique for Comparing Interactive System Capacities», Conference Proceedings of CMGXIII, Dec. 1982, pp. 62–67.
7. Gingell, R. A., Moran, J. P., and Shannon, W. A., «Virtual Memory Architecture in SunOS», Proceedings of the Summer 1987 USENIX Technical Conference, Jun. 1987, pp. 81–94.
8. Hendricks, D., «A FileSystem for Software Development», Proceedings of the Summer 1990 USENIX Technical Conference, Jun. 1990, pp. 333–340.
9. Kleiman, S. R., «Vnodes: An Architecture for Multiple File System Types in Sun UNIX», Proceedings of the Summer 1986 USENIX Technical Conference, Jun. 1986, pp. 238–247.
10. Kowalski, T., «FSCK — The UNIX System Check Program», Bell Laboratory, Murray Hill, N.J. 07974, Mar. 1978.
11. Kridle, R., and McKusick, M., «Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX», Technical Report No. 8, Computer Systems Research Group, Dept. of EECS, University of California at Berkeley, CA, 1983.
12. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3BSD UNIX Operating System», Addison-Wesley, MA, 1989.
13. McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S., «A Fast File System for UNIX», ACM Transactions on Computer Systems, Vol. 2, (Aug. 1984), pp. 181-197.
14. McKusick, M. K., Karels, M., and Leffler, S. J., «Performance Improvements and Functional Enhancements in 4.3BSD», Proceedings of the Summer 1985 Conference, Jun. 1985, pp. 519–531.
15. McKusick, M. K., Karels, M. K., and Bostic, K., «A Pageable Memory Based Filesystem», Proceedings of the Summer 1990 USENIX Technical Conference, Jun. 1990.
16. Nadkarni, A. V., «The Processor File System in UNIX SVR4.2», Proceedings of the 1992 USENIX Workshop on File Systems, May 1992, pp. 131–132.
17. Ohta, M. and Tezuka, H., «A Fast /tmp File System by Delay Mount», Proceedings of the Summer 1990 USENIX Conference, Jun. 1990, pp. 145–149.
18. Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G., «A Trace-Driven Analysis of the UNIX 4.2 BSD File

System», Proceedings of the Tenth Symposium on Operating System Principles, Dec. 1985, pp. 15–24.

19. Rifkin, A. P., Forbes, M. P., Hamilton, R. L., Sabrio, M., Shah, S., and Yueh, K. «RFS Architectural Overview», Proceedings of the Summer 1986 USENIX Conference, Jun. 1986, pp. 248–259.
20. Salus, P. H., «A Quarter Century of UNIX», Addison-Wesley, Reading, MA, 1994.
21. Satyanarayan, M., «A Study of File Sizes and Functional Lifetimes», Proceedings of the Eighth Symposium on Operating Systems Principles, 1981, pp. 96–108.
22. Snyder, P., «tmpfs: A Virtual Memory File System», Proceedings of the 1990 European UNIX Users' Group Conference, Oct. 1990, pp. 241–248.
23. Thompson, K., «UNIX Implementation», The Bell System Technical Journal, Jul.–Aug. 1978, Vol. 57, No. 6, Part 2, pp. 1931–1946.

Глава 10
Распределенные файловые системы

10.1. Введение

Появление в 70-х годах возможности соединения компьютеров между собой стало поистине революционным для компьютерной индустрии. Именно это и послужило причиной создания средств совместного использования файлов в компьютерных сетях. Поначалу производители сконцентрировали свое внимание на средствах копирования файлов с одной машины на другую целиком, таких как программа `uucp` (UNIX-to-UNIX copy Protocol) [20] и протокол передачи файлов (FTP, File Transfer Protocol) [24]. Однако подобные решения еще только представляли иллюзию доступа к файлам на удаленных машинах, ничем не отличающегося для пользователя от работы с локальной файловой системой.

В середине 80-х годов появились первые распределенные файловые системы, предоставляющие прозрачный доступ к удаленным файлам по сети. Это системы NFS (Network File System) корпорации Sun Microsystems [27], RFS (Remote File Sharing) компании AT&T [26] и AFS (Andrew File System), созданная в университете Карнеги—Меллона [29][1]. Перечисленные три файловые системы отличаются между собой как по целям, стоявшими перед их разработчиками, так и по их архитектуре и семантике. Вместе с тем их создатели пытались решить одну и ту же общую проблему. На сегодняшний день RFS поддерживается многими операционными системами, основанными на System V. Файловая система NFS получила более широкое распространение и используется многими вариантами как UNIX-, так и не-UNIX-систем. Развитие AFS было продолжено компанией Transarc Corporation, где она была доведена до системы DFS (Distributed File System), являющейся компонентом Distributed Computing Environment (распределенная среда вычислений), продвигаемой Open Software Foundation.

---

[1] AFS получила название в честь основателей университета и его спонсоров Эндрю Карнеги (Andrew Carnegie) и Эндрю Меллона (Andrew Mallon). — *Прим. ред.*

Эта глава начинается с обсуждения общих характеристик распределенных файловых систем. Затем вы получите представление об архитектуре и реализации каждой из упомянутых выше систем, а также вас ждет анализ их сильных и слабых сторон.

10.2. Общие характеристики распределенных файловых систем

Обычная централизованная файловая система позволяет нескольким пользователям одной системы совместно использовать файлы, размещенные на одной машине локально. Распределенные системы расширяют возможности разделения файлов между пользователями, позволяя им иметь доступ к удаленным файлам, расположенным на компьютерах, объединенных в сеть. Распределенные файловые системы основаны на модели «клиент-сервер». Клиентом является машина, запрашивающая доступ к файлу, в то время как сервер представляет собой место хранения этих файлов и обеспечивает доступ к ним клиентов. В некоторых системах клиенты и серверы должны дислоцироваться на отдельных машинах, в других один и тот же компьютер может одновременно являться как клиентом, так и сервером.

Важно понять различие между распределенными файловыми системами и распределенными операционными системами [38]. Распределенная файловая система, такая как *V* [5] или *Amoeba* [39], видится на прикладном уровне как централизованная система, но на самом деле выполняется на нескольких компьютерах. Она управляет на прикладном уровне соединениями между операционными и файловыми системами. Она также предоставляет службу распределенного доступа к файлам систем, имеющих централизованное ядро, и используется совместно всеми хостами.

Существует несколько важнейших свойств, присущих распределенным файловым системам [16]. Каждая конкретная реализация может обладать всеми или только некоторыми из них. Эти свойства помогут нам оценить и сравнить различные архитектуры.

- **Прозрачность в сети.** Клиенты должны иметь возможность получения доступа к удаленным файлам, применяя для этой цели те же операции, которые они используют для работы с локальными файлами.
- **Прозрачность расположения.** Имя файла не должно зависеть от конкретного месторасположения в сети.
- **Независимость расположения.** Имя файла не должно изменяться при изменении физического местонахождения.
- **Мобильность пользователя.** Пользователи должны иметь возможность доступа к совместно используемым файлам из любой точки сети.

- **Отказоустойчивость.** Система обязана продолжать работать после возникновения сбоя в каком-либо одном компоненте (сервере или сегменте сети). При этом, однако, может понизиться ее производительность или часть файловой системы станет недоступной.
- **Масштабируемость.** Система должна уметь масштабироваться по мере увеличения загрузки. Также важна способность к постепенному увеличению системы путем добавления компонентов.
- **Мобильность файлов.** Должна поддерживаться возможность перемещения файлов из одного физического расположения в другое при выполнении системы.

10.2.1. Некоторые соглашения

При разработке распределенной файловой системы необходимо выбрать несколько различных факторов, которые влияют на ее функциональность, семантику и производительность. Мы будем проводить сравнение различных распределенных файловых систем, исходя из нижеперечисленных соображений.

- **Пространство имен.** Некоторые распределенные файловые системы предлагают унифицированное пространство имен, например такие, в которых для доступа к конкретному файлу все клиенты используют одно и тоже полное имя. В иных реализациях каждый клиент может настроить пространство имен индивидуально путем монтирования поддеревьев в определенных каталогах файловой иерархии. В распределенных системах применимы оба метода.
- **Работа в режиме запоминания состояния и без него.** Сервер, поддерживающий запоминание состояния, сохраняет информацию об операциях клиентов между запросами и использует ее для корректного обслуживания последующих запросов. Некоторым запросам, например `open` или `seek`, необходимо помнить, какие файлы были открыты клиентом, а также смещение для каждого открытого файла. В системах, не запоминающих состояние, каждый запрос является самодостаточным, а сервер не хранит информацию о клиентах. Последние серверы работают быстрее, поскольку используют преимущества знаний о состоянии клиента, что экономит значительный объем сетевого трафика. Однако такие серверы обычно используют более сложные механизмы поддержки целостности и восстановления после сбоев. Серверы, не сохранящие информацию о состоянии, более просты в реализации, но не обладают высокой производительностью.
- **Семантика разделения.** В распределенных файловых системах должна быть определена семантика, регламентирующая действия при одновременном доступе к файлу нескольких клиентов. Семантика UNIX требует того, чтобы изменения, сделанные одним клиентом, были видимы

для остальных при последующем вызове read или write. Некоторые файловые системы поддерживают семантику сеансов, где изменения показываются остальным клиентам только после вызовов open или close. Другие системы предоставляют иные виды гарантий, например некий определенный интервал времени, после которого клиент должен увидеть изменения в файле.

- **Методы удаленного доступа.** В идеальной клиент-серверной модели для доступа к файлам применяется метод удаленных служб, где каждое действие инициализируется клиентом, а сервер является простым агентом, обслуживающим клиентские запросы. Во многих распределенных файловых системах (в частности, поддерживающих запоминание состояния) сервер играет более заметную роль. Он не только обслуживает клиентские запросы, но и участвует в механизме согласования кэша, уведомляя клиентов, если их кэшированные данные стали неверными.

Рассмотрим несколько распределенных файловых систем, используемых в UNIX, и то, как они поддерживают перечисленные средства.

10.3. Network File System (NFS)

Корпорация Sun Microsystems представила NFS в 1985 году как средство прозрачного доступа к удаленным файловым системам.[1] Кроме публикации протоколов компания лицензировала рекомендации, которые были использованы некоторыми производителями для переноса NFS на свои операционные системы. После этого файловая система NFS стала индустриальным стандартом де-факто. Она поддерживается всеми вариантами UNIX, а также некоторыми другими операционными системами, такими как VMS или MS-DOS.[2]

Архитектура NFS основана на модели «клиент-сервер». Файл-сервер представляет собой машину, экспортирующую набор файлов.[3] Клиентами являются машины, запрашивающие доступ к этим файлам. Один компьютер может быть

---

[1] Имеется в виду NFSv2 для SunOS 2.0. Сама NFS была разработана в 1980 г. В 1985 Sun разрешила IETF публикацию протокола в качестве открытого стандарта [RFC 1094]. — *Прим. ред.*

[2] На основе спецификаций и исходного кода Sun ныне NFS выпускается более чем 200 производителями. Клиентская версия PC-NFS 5.1 (Sun Microsystems) работает с VMS, MS-DOS, Windows 3.x/95/NT через TCP/IP, поддерживает автоматическое конфигурирование клиентов (DHCP), длинные имена файлов, доступ к Интернету (DNS, NIS), консоль Telnet для Windows, включая печать на сетевых принтерах. Есть версии PC-NFS для Digital Alpha, MIPS и PowerPC, работающих под управлением Windows NT. — *Прим. ред.*

[3] Экспортированием называется объявление сервером доступности своей файловой системы или ее части для клиентов, фактически это синоним понятия выделения в совместное пользование (sharing). Клиенты получают доступ к удаленной файловой системе через ее монтирование (mounting), после чего удаленные файлы и каталоги выглядят как часть локальной файловой системы. — *Прим. ред.*

одновременно сервером и клиентом для нескольких файловых систем. При этом код NFS разделен на две отдельные части — клиентскую и серверную, что позволяет использовать только клиентские или только серверные составляющие.

Клиенты и серверы взаимодействуют между собой через *вызовы удаленных процедур*, функционирующих как синхронные запросы. Когда приложение на клиентской машине пытается получить доступ к удаленному файлу, ядро посылает запрос на сервер, а клиент блокируется до получения ответа. Сервер ожидает входящие клиентские запросы и отправляет ответы на них.

10.3.1. NFS с точки зрения пользователя

Сервер NFS экспортирует одну или несколько файловых систем. Каждая экспортируемая система может являться как целым разделом диска, так и его частью[1]. Серверу можно указать (обычно через вхождения файла /etc/exports), какие из клиентов обладают доступом к каждой экспортируемой файловой системе, а также какими из привилегий они обладают («только чтение» или «чтение-запись»).

Клиентские машины умеют монтировать такие файловые системы (или отдельные поддеревья каталогов) в любой каталог в существующей файловой иерархии, точно так же, как производится монтирование локальных файловых систем. Клиент может подключать каталог в режиме «только для чтения» даже в том случае, если сервер экспортирует его в режиме чтения-записи. Система NFS поддерживает два типа монтирования: *жесткий* (hard) и *мягкий* (soft). Тип влияет на поведение клиента в случае отсутствия ответа на запрос от сервера. Если система смонтирована жестко, клиент будет посылать повторные запросы до тех пор, пока сервер не ответит. Для систем, смонтированных мягко, клиент через некоторый промежуток времени получает ошибку. После удачного выполнения mount клиент может получать доступ к удаленным файлам, используя те же операции, что и для локальных файлов. Некоторые системы поддерживают *смешанное* (spongy) монтирование, при котором используется жесткий подход при монтировании системы и мягкий для последующих операций ввода-вывода.

Монтирование NFS не требует такого уровня привилегий, как для локальных файловых систем. Протокол не обязывает, чтобы пользователь, монтирующий систему, являлся привилегированным, однако большинство клиентов навязывают это требование[2]. Клиент волен монтировать одну и ту же файло-

[1] Различные реализации UNIX обладают собственными правилами степени детализации при экспорте. Некоторые из них поддерживают экспорт только всей файловой системы целиком, в то время как другие позволяют иметь доступ всего к одному поддереву каждой файловой системы.

[2] К примеру, система ULTRIX корпорации Digital разрешает любому пользователю монтировать файловую систему NSF до тех пор, пока тот будет обладать правами на запись экспортируемой файловой системы целиком, в то время как другие могут монтировать только одно поддерево в каждой файловой системе.

вую систему в нескольких местах дерева каталогов, даже в подкаталогах ее же самой. Сервер может экспортировать только собственные локальные системы и не имеет права пересекать свои точки монтирования при просмотре каталогов. Следовательно, для того чтобы клиент видел все файлы на сервере, необходимо монтировать все файловые системы, находящиеся на нем.

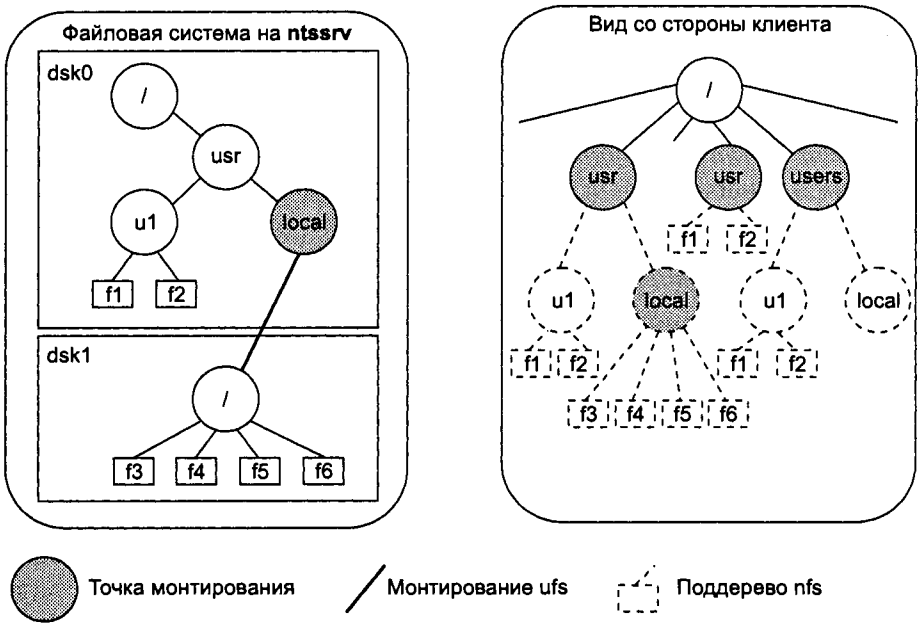


Рис. 10.1. Монтирование файловых систем NFS

Пример монтирования NFS показан на рис. 10.1. Серверная система nfssrv имеет два диска. Первый диск dsk1 монтирован в каталоге /usr/local диска dsk0 и экспортирует два каталога — /usr и /usr/local. Клиент может выполнить следующие варианты операций монтирования:

```
mount -t nfs nfssrv:/usr          /usr
mount -t nfs nfssrv:/usr/u1       /u1
mount -t nfs nfssrv:/usr          /users
mount -t nfs nfssrv:/usr/local    /usr/local
```

Все приведенные варианты команд являются верными. На клиентской машине поддерево /usr отражает поддерево /usr системы nsfsrv полностью, так как клиент смонтировал и каталог /usr/local. Поддерево /u1 клиента указывает на поддерево /usr1/u1 сервера. Пример показывает возможность монтирования подкаталога экспортируемой файловой системы[1]. Поддерево /users клиента указывает только на часть поддерева /usr системы nsfsrv, рас-

[1] Такую функциональность поддерживают не все реализации.

положенной на диске dsk0. Файловая система на диске dsk1 невидима в каталоге /users/local клиента.

10.3.2. Цели, стоявшие перед разработчиками

При создании оригинального варианта NFS перед разработчиками стояли следующие задачи:

- система NFS не должна ограничиться UNIX. Реализация сервера или клиента NFS обязана быть доступна любой операционной системе;
- обеспечение независимости протокола от конкретно используемого оборудования;
- реализация простых механизмов восстановления после отказов сервера или клиента;
- приложения должны иметь возможность прозрачного доступа к файлам, не применяя для этого специализированные полные имена, а также не использовать дополнительные библиотеки и повторную компиляцию кодов;
- для UNIX-клиентов поддержка семантики, принятой в файловых системах UNIX;
- производительность NFS должна быть сравнима со скоростью работы систем, размещенных на локальных дисках;
- реализация должна быть независима от средств передачи данных.

10.3.3. Компоненты NFS

Система NFS состоит из нескольких компонентов. Часть из них расположена либо на клиенте, либо только на сервере, в то время как другие имеются и там и там. Некоторые из компонентов не требуются для поддержки стандартных средств и формируют собой часть расширенного интерфейса NFS:

- *протокол NFS* определяет набор запросов, направляемых клиентами серверу, а также аргументы и возвращаемые величины для каждого из них. *Версия 1* этого протокола использовалась только внутри корпорации Sun Microsystems и никогда не выходила «на свет». Все реализации NFS поддерживают протокол *NFS версии 2* (*NFSv2*)[1], реализованный в системе SunOS 2.0 в 1985 году [28]. В данной главе книги мы будем рассказывать преимущественно об этой версии системы. *Версия 3* будет обсуждаться в разделе 10.10. Она была опубликована в 1993 году и в дальнейшем реализована большим количеством поставщиков ОС. Полный набор запросов NFSv2 перечислен в табл. 10.1;

[1] Эта версия содержит функции, поддерживаемые NFSv3.

- *протокол вызова удаленных процедур* (Remote Procedure Call, RPC) определяет формат всех взаимодействий между клиентом и сервером. Каждый запрос NFS посылается в пакете RPC;
- *внешнее представление данных* (Extended Data Representations, XDR) является машинно-независимой методикой кодирования данных для пересылки по сети. Все запросы RPC используют кодирование XDR для передачи данных. Следует упомянуть о том, что технологии XDR и RPC применяются не только в системе NFS, но и в других службах;
- *код сервера NFS* отвечает за обработку всех клиентских запросов и предоставляет доступ к экспортируемым файловым системам;
- *код клиента NFS* реализует все клиентские системные вызовы, производимые по отношению к удаленным файлам посредством отправки одного или нескольких запросов RPC серверу;
- *протокол монтирования* определяет семантику монтирования и размонтирования файловых систем NFS. Краткое описание протокола приведено в табл. 10.2;
- система NFS использует несколько *процессов-демонов*. На сервере выполняется набор демонов nfsd, ожидающих клиентские запросы NFS и отвечающих на них. Демон mountd обрабатывает запросы на монтирование. На клиентской машине набор демонов biod отвечает за асинхронный ввод-вывод блоков файлов NFS;
- *сетевой администратор блокировки* (Network Lock Manager, NLM) и *монитор состояния сети* (Network Status Monitor, NSM) совместно являются средствами блокировки файлов по сети. Они не привязаны конкретно к NFS, но обычно имеются в большинстве реализаций этой файловой системы, предлагая возможности, не поддерживаемые базовым протоколом. Средства NLM и NSM реализованы на сервере в виде демонов lockd и statd соответственно.

Таблица 10.1. Операции, определенные в NFSv2

Процедура	Входные аргументы	Результат
NULL	void	void
GETATTR	fhandle	status, fattr
SETATTR	fhandle, sattr	status, fattr
LOOKUP	dirfh, name	status, fhandle, fattr
READLINK	fhandle	status, link_value
READ	fhandle, offset, count, totcount	status, fattr, data
WRITE	fhandle, offset, count, totcount, data	status, fattr
CREATE	dirfh, name, sattr	status, fhandle, fattr
REMOVE	dirfh, name	status

Процедура	Входные аргументы	Результат
RENAME	dirfh1, name1, dirfh2, name2	status
LINK	fhandle, dirfh, name	status
SYMLINK	dirfh, name, linkname, sattr	status
MKDIR	dirfh, name, sattr	status, fhandle, fattr
RMDIR	dirfh, name	status
READDIR	fhandle, cookie, count	status, dir_entries
STATFS	fhandle	status, file_stats

Обозначения: fattr — файловые атрибуты, sattr — устанавливаемые атрибуты, cookie — объект закрытого типа, возвращенный предыдущим вызовом READDIR, fhandle — дескриптор файла, dirfh — дескриптор файла каталога.

Таблица 10.2. Протокол mount (версии 1)

Процедура	Входные аргументы	Результат
NULL	void	void
MNT	pathname	status, fhandle
DUMP	void	mount list
UMNT	pathname	void
UMNTALL	void	void
EXPORT	void	export list

Обозначения: fhandle — дескриптор каталога верхнего уровня монтируемого поддерева.

10.3.4. Сохранение состояний

Одной из важнейших характеристик протокола NFS является отсутствие запоминания состояний сервером[1]. Он не хранит никакой информации о корректности операций, производимых клиентами. Каждый запрос абсолютно независим от других и содержит всю информацию, необходимую для его обработки. Сервер не ведет каких-либо записей предыдущих запросов клиентов, кроме тех, что производятся в целях сбора статистики или кэширования.

К примеру, протокол NFS не поддерживает запросы на открытие или закрытие файла, так как они должны нести некую информацию о состоянии, которую сервер должен помнить. Запросы READ и WRITE передают смещение в качестве входного аргумента, в то время как стандартные операции

[1] В русском языке более корректного перевода термина stateless, чем «запоминание состояния», нет. Наиболее близкое по смыслу значение — безразличность, независимость от состояния. NFS изначально задумывался с целью снять максимум нагрузки с серверной части, чтобы ее функционирование ни в коей мере не зависело от информации о состоянии клиентов, и наоборот — при «падении» сервера его можно перезагрузить, не волнуясь о клиентах. — *Прим. ред.*

read и write по отношению к локальным файлам не требуют передачи смещения, поскольку получают его значение из *объекта открытого файла* (см. раздел 8.2.3)[1].

При использовании протоколов, не запоминающих состояния, реализация механизмов защиты весьма проста. При отказе клиента нет необходимости проводить восстановление, так как сервер не поддерживает какой-либо постоянно хранящейся информации о клиенте. После перезагрузки клиентской машины система может заново смонтировать и загрузить приложения, осуществляющие доступ к удаленным файлам. Серверу не нужно знать или заботиться о выходе из строя системы клиента.

Если происходит отказ сервера, клиенты обнаруживают отсутствие ответов на их запросы. Они продолжают посылать повторные запросы до тех пор, пока сервер не будет перезагружен[2]. После этого сервер получит запросы и сможет обработать их, так как отдельный запрос не зависит от какой-либо информации о состоянии. После отправки ответа клиент заканчивает посылку одного и того же запроса. В системе не существует какого-либо средства, позволяющего клиенту определить, что случилось с сервером: претерпел ли он крах (или перезагрузку) или просто работает слишком медленно.

Протоколы, не запоминающие состояния, требуют сложных механизмов восстановления после краха. Серверу необходимо обнаруживать сбои клиентов и запрещать любые действия по отношению к ним. Если сервер выходит из строя, то после перезагрузки он должен оповестить клиентов о необходимости перенастроить их доступ к серверу.

Основной проблемой для таких протоколов является потребность сохранять все изменения на постоянном носителе перед отправкой ответа на запрос. Это означает, что не только данные файла, но и его метаданные (например, индексные дескрипторы или блоки косвенной адресации) должны быть сброшены на диск до возврата результатов клиенту. С другой стороны, при выходе из строя сервера могут быть потеряны данные, которые клиент считает успешно сохранившимися на дисках (при крахе системы есть вероятность их исчезновения даже на локальных файловых системах, но в этом случае пользователи предупреждаются о факте сбоя и возможной потере информации). Имеются и некоторые другие недостатки. В этом случае требуется отдельный протокол (NLM) для поддержки блокировки файлов. Также для решения проблем, связанных с синхронностью записи, большинство клиентов кэшируют данные и метаданные локально. Эта уступка является гарантией корректности протокола (см. подробнее в разделе 10.7.2).

---

[1] Некоторые системы поддерживают вызовы pread и pwrite, где смещение используется в качестве входного аргумента. Такие команды особенно полезны в многонитевых системах (см. раздел 3.3.2).

[2] Запросы будут повторяться до получения ответа только при использовании *жесткого монтирования* (устанавливаемого по умолчанию). При *мягком монтировании клиент* через определенный промежуток перестает упорствовать и возвращает вызвавшему его приложению ошибку.

10.4. Набор протоколов

В основной набор протоколов NFS входят RPC, NFS и Mount. Все они используют для кодирования данных метод XDR. К файловой системе относятся еще несколько протоколов: NLM, NSM и portmapper. В этом разделе будет описана методика XDR и протокол RPC.

10.4.1. Представление внешних данных (XDR)

Программы, использующие сетевые межкомпьютерные коммуникации, должны заботиться о некоторых проблемах, связанных с интерпретацией данных, передаваемых по сети. Компьютеры на каждом конце соединения могут иметь различную аппаратную архитектуру или операционную систему, а также неодинаковые соглашения о внутреннем представлении элементов данных. Среди таких различий — порядок следования байтов, размеры типов данных (например, целых чисел — integer), а также формат строк и массивов. Все перечисленные расхождения не играют заметной роли в коммуникациях внутри одной машины или при взаимодействии двух компьютеров, однако требуют внимания в не ,однородных средах.

Данные, передаваемые по компьютерным сетям, можно разделить на две категории: закрытого типа и типизированные. Использование закрытых (opaque) данных или потоков байтов осуществляется, к примеру, при передаче файлов или модемных соединениях. Получатель воспринимает информацию как последовательность байтов и не пытается интерпретировать их каким-либо образом. Типизированные данные, наоборот, интерпретируются получателем, что требует определенного соглашения между ним и отправителем о формате. К примеру, машина, использующая *обратный порядок байтов*, отправляет двухбайтовое целое число 0x0103 (или 259 в десятичной системе). Получателем числа является компьютер, основанный на *прямом следовании байтов*, который может истолковать его как 0x0301 (десятичное 769), если между машинами не существует договоренности о форматах. Очевидно, что в данном случае отправитель и получатель не смогут понять друг друга.

Стандарт XDR [34] определяет машинно-независимое представление данных, передаваемых по сети. В стандарте описаны несколько основных типов данных, а также правил, по которым осуществляется построение более сложных типов данных. Методика XDR была разработана корпорацией Sun Microsystems и реализована для архитектуры Motorola 680x0 (рабочие станции Sun-2 и Sun-3 имели процессор 680x0) в отношении таких проблем, как порядок следования байтов. Ниже перечислены некоторые базовые определения стандарта XDR.

- **Целые числа** (integer) являются 32-разрядными, где байт 0 является старшим значащим байтом (нумерация байтов осуществляется слева направо). Целые числа со знаком представлены двумя дополнительными определениями.

- **Закрытые данные переменной длины** (variable-length opaque data) описываются полем length (четырехбайтовым целым числом) и следующими за ним данными. Данные заполняются нулями до достижения четырехбайтовой границы. Поле length не добавляется для закрытых данных фиксированного размера.
- **Строки** (string) представлены полем length и следующими за ним ASCII-кодами строки, заполненной нулями до достижения четырехбайтовой границы. Если длина строки кратна четырем, она не заканчивается байтом NULL (как это принято в UNIX).
- **Массивы** (array) однородных элементов описаны полем size, после чего следуют элементы в их обычном порядке. Поле размера является четырехбайтовым целым числом и не используется для массивов фиксированных размеров. Размер каждого элемента должен быть кратен четырем. Хотя элементы массива обязаны быть одного типа, они могут иметь неодинаковую длину, например в массиве строк.
- **Структуры** (structure), для которых кодирование их компонентов производится в обычном порядке. Каждый компонент должен дополняться до четырехбайтовой границы.

Нескорые примеры кодирования методом XDR представлены на рис. 10.2. Кроме указанного набора определений стандарт предлагает спецификацию формального языка, применяемого для описания данных. Спецификации RPC, описываемые в следующем разделе, являются простыми дополнениями языка XDR. Компилятор rpcgen понимает спецификации XDR и умеет создавать процедуры, которые шифруют и расшифровывают данные, представленные в форме XDR.

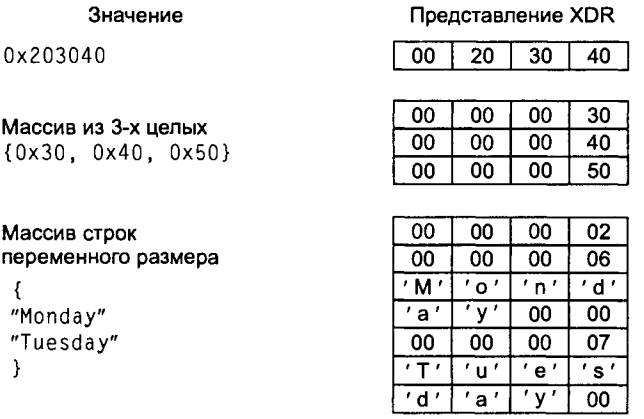


Рис. 10.2. Пример XDR-преобразования

Стандарт XDR формирует универсальный язык, применяемый для осуществления коммуникаций между самыми различными компьютерами. Его

основным недостатком является снижение производительности при использовании на машинах, семантика представления данных которых отличается от принятой в XDR. Таким компьютерам приходится выполнять операции преобразования для каждого передаваемого элемента данных. Если две взаимодействующие стороны поддерживают одинаковое внутреннее представление данных и не требуют их преобразования для передачи друг другу, избыточность таких операций становится очевидной.

В качестве примера можно привести две машины VAX-11, работающие с протоколом, основанным на кодировании XDR. Так как VAX использует обратный порядок байтов (нулевой байт является старшим), отправителю необходимо произвести преобразование данных в формат с прямым порядком байтов (принятый в XDR), после передачи которых получатель снова перекодирует их в форму с обратным порядком следования байтов. Эти избыточные действия можно предотвратить, если представление данных будет зависеть от аппаратных характеристик, тогда преобразование нужно выполнять только в случае несовместимых архитектур. Реализация DCE RPC [21] использует эту схему вместо оригинальной методики, принятой в XDR.

10.4.2. Вызов удаленных процедур (RPC)

Протокол вызова удаленных процедур (Remote Procedure Call, RPC) определяет формат взаимодействий между клиентом и сервером. Клиент отправляет *запрос RPC* серверу, который, в свою очередь, обрабатывает его и возвращает результат в *ответном сообщении RPC*. Протокол устанавливает формат сообщений, передачу и аутентификацию и т. д., то есть решает вопросы, не зависящие от определенной спецификации или сервиса. На основе RPC построено несколько служб, таких как NFS, Mount, NLM, NSM, portmapper и NIS (информационная сетевая служба).

ПРИМЕЧАНИЕ

Существуют несколько различных реализаций протокола RPC. Файловая система NFS использует версию протокола RPC, представленную Sun Microsystems [35], известную под названиями Sun RPC и ONC-RPC (ONS является аббревиатурой Open Network Computing). В этой книге термин RPC означает реализацию Sun RPC (если это не оговорено иначе). Единственным альтернативным вариантом RPC, упоминаемым на страницах книги, является DCE RPC — версия протокола Distributed Computing Environment, созданная для OSF.

В отличие от DCE RPC, поддерживающего как асинхронные, так и синхронные операции, протокол Sun RPC прибегает только к синхронным запросам. После того как клиент производит запрос RPC, вызывающий процесс блокируется до тех пор, пока не будет получен ответ. Такой подход делает RPC похожим на технологию работы локальных процедурных вызовов.

Протокол RPC отвечает за надежность передачи запросов. Это означает, что он должен удостовериться в достижении запросом адресата и отправке ответа на него. Хотя протокол RPC фундаментально независим от используемого транспорта, он часто применяется поверх протоколов UDP/IP (User Datagram Protocol/Internet Protocol), по природе ненадежных. На уровне RPC реализуется надежная служба дейтаграмм путем отслеживания запросов, не получивших ответа, и периодической повторной посылки их, осуществляемой до тех пор, пока ответ не будет получен.

На рис. 10.3 показан типичный формат запроса и обратного (удачного) сообщения RPC. Переменная xid — идентификатор передачи, являющийся меткой запроса. Клиент генерирует для каждого запроса уникальный номер xid, сервер в ответном сообщении возвращает то же значение переменной. Это позволяет клиенту идентифицировать, на какой из запросов пришел ответ, а серверу обнаруживать дубликаты запросов (появляющиеся после повторов запросов клиентов). Поле direction указывает, является ли сообщение запросом или ответом. Поле rpc_vers содержит номер версии используемого протокола RPC (текущая версия — 2). Переменные prog и vers указывают на номер программы, обеспечивающей определенную службу RPC, а также ее версию. Служба RPC может поддерживать многопротокольные версии. Например, протокол NFS имеет программный номер 100 003 и различается версиями 2 и 3. Поле proc конкретизирует процедуру, которую необходимо вызвать служебной программой. В ответном сообщении поля reply_stat и accept_stat содержат информацию о статусе.

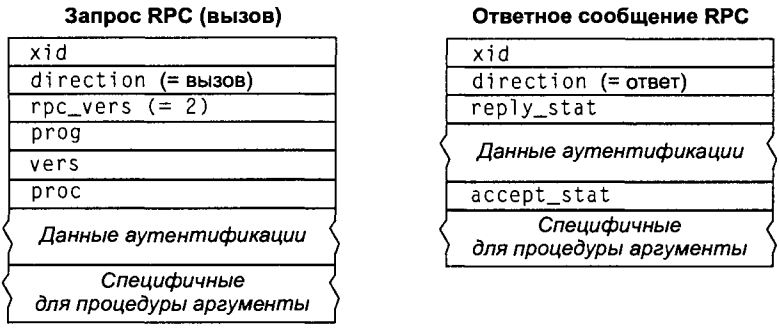


Рис. 10.3. Формат RPC-сообщений

Протокол RPC применяет пять механизмов опознания обращающегося к серверу — AUTH_NULL, AUTH_UNIX, AUTH_SHORT, AUTH_DES, AUTH_KERB. При задании AUTH_NULL какая-либо проверка отсутствует, метод AUTH_UNIX основан на привилегиях, принятых в UNIX, и включает в себя имя клиентской машины, идентификатор пользователя и один или несколько групповых идентификаторов. Сервер может сгенерировать AUTH_SHORT после получения привилегий AUTH_UNIX и возвратить их клиенту для использования в после-

дующих запросах. Такой подход применяется потому, что сервер способен очень быстро идентифицировать клиентов по привилегиям AUTH_SHORT, тем самым ускоряя процесс аутентификации. Описанное средство является дополнительным и поддерживается далеко не всеми службами. Средство проверки AUTH_DES использует механизм *закрытых ключей* [36], AUTH_KERB основано на методике *Kerberos* [32]. Выбор инструмента контроля осуществляется каждой службой самостоятельно. Файловая система NFS поддерживает все пять перечисленных механизмов, ограничивая применение AUTH_NULL для процедуры NULL[1]. Большинство реализаций NFS, тем не менее, используют преимущественно AUTH_UNIX.

Корпорация Sun представила также язык программирования RPC и компилятор языка, называемый rpcgen. Служба, основанная на RPC, может быть полностью описана при помощи этого языка. При переработке описания программа rpcgen создает набор исходных файлов на C, которые содержат процедуры преобразования XDR и фиктивные версии клиентских и серверных процедур, а также заголовочный файл, включающий в себя определения, используемые совместно сервером и клиентом.

10.5. Реализация NFS

В этом разделе будут рассмотрены реализации протокола NFS в стандартных системах UNIX. Протокол NFS был перенесен и на не-UNIX-системы, такие как MS-DOS и VMS. Некоторые решения являются чисто клиентскими или серверными, в то время как другие версии предлагают оба варианта. Более того, существуют серверные реализации под конкретные системы, созданные компаниями Auspex, Network Alliance Corporation и Novell, не работающие под управлением систем, предназначенных для общих целей. Созданы и реализации NFS на прикладном уровне для различных операционных систем, многие из которых доступны бесплатно или условно-бесплатно. Конечно, эти варианты системы в деталях устроены по разному. Некоторые интересные варианты будут описаны в разделе 10.8. В этом разделе мы расскажем только о реализации NFS на уровне ядра для традиционных систем UNIX, поддерживающих интерфейс vnode/vfs.

10.5.1. Управление потоком

На рис. 10.4 сервер экспортировал файловую систему ufs, монтированную клиентом. Если процесс на клиентской машине производит системный вызов, относящийся к файлу NFS, независимая от файловой системы часть кода

[1] Процедура NULL используется только для тестирования и хронометража ответа сервера. — *Прим. ред.*

идентифицирует vnode файла и выполняет соответствующую этому объекту операцию. В случае NFS зависимая от файловой системы структура данных, ассоциированная с vnode, называется rnode («для удаленного узла»). Поле v_op объекта vnode указывает на вектор клиентских процедур NFS (структуру vnodeops), реализующих различные операции над vnode. Процедуры производят создание запросов RPC и отправку их серверу. Вызывающий клиентский процесс блокируется до получения ответа от сервера. Сервер обслуживает запрос путем идентификации vnode соответствующего локального файла и активизации его структуры vnodeops, реализованной для конкретной локальной файловой системы (в рассматриваемом примере — ufs).

Сервер завершает обработку запроса, упаковывает результаты в ответное сообщение RPC и пересылает его обратно клиенту. Получателем ответа является процесс, работающий на уровне RPC, который разбудит ожидающий процесс. Процесс, в свою очередь, завершит выполнение операции над клиентским объектом vnode и оставшейся части системного вызова и возвратит управление системой пользователю.

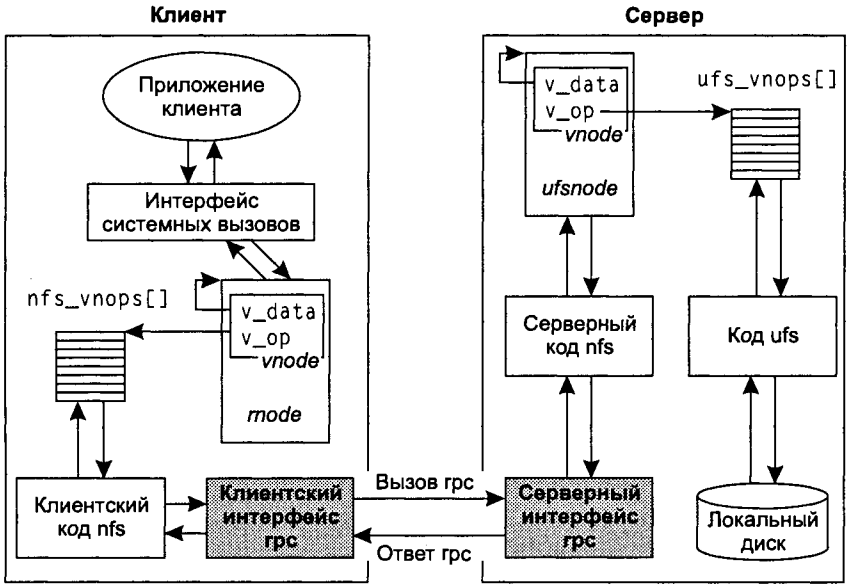


Рис. 10.4. Управление потоком в системе NFS

Пример показывает необходимость поддержки клиентом локального объекта vnode каждого активного файла NFS. Такой подход позволяет осуществлять более точные связи между операциями над vnode и клиентскими функциями NFS. Он также позволяет клиенту кэшировать атрибуты удаленных файлов, производя некоторые операции без доступа к серверу. Вопросы, связанные с кэшированием в клиентской системе, обсуждаются в разделе 10.7.2.

10.5.2. Дескрипторы файлов

При создании запроса NFS клиентом ему необходимо указать серверу, какой из файлов ему необходим для доступа. Передача полного имени при каждом доступе может оказаться слишком медленной. Вместо этого протокол NFS ассоциирует с каждым файлом или каталогом объект, называемый *дескриптором файла* (file handle). Сервер создает такой дескриптор при первом доступе к файлу или при его создании путем запросов LOOKUP, CREATE или MKDIR. Сервер возвращает дескриптор клиенту в ответном сообщении, после чего тот может использовать его в последующих операциях над файлом.

Клиент воспринимает дескриптор файла как закрытый 32-разрядный объект и не интерпретирует его содержимое. Сервер создает дескриптор на свое усмотрение, а также осуществляет уникальную связь между файлами и их дескрипторами. Обычно дескриптор содержит *идентификатор файловой системы* (однозначно идентифицирующий локальную файловую систему), номер *индексного дескриптора* файла и *генерируемый номер* для указанного индексного дескриптора. Он может хранить также индексный дескриптор и его генерируемый номер для экспортируемого каталога, через который осуществляется доступ к файлу.

Генерируемый номер был добавлен в индексный дескриптор для решения некоторых проблем, относящихся к системе NFS. Существует вероятность удаления файла и повторного использования его индексного дескриптора между первоначальным клиентским доступом к файлу (обычно происходящем через вызов LOOKUP, возвращающим дескриптор файла) и запросом ввода-вывода по отношению к этому файлу. Следовательно, сервер должен обладать возможностями определения факта устаревания дескриптора, присланного клиентом. Это реализовано путем автоувеличения генерируемого номера индексного дескриптора, производимого при каждом освобождении соответствующего индексного дескриптора (при удалении ассоциированного с ним файла). Сервер таким образом может распознать запросы, относящиеся к уже не существующим файлам, и ответить на них отправкой ошибки ESTALE (*устаревший дескриптор файла*).

10.5.3. Операция монтирования

При монтировании файловой системы NFS клиентом ядро ассоциирует с ней новую структуру vfs и вызывает функцию nfs_mount(). Функция отправляет запрос RPC серверу, используя протокол Mount. Аргументом такого запроса является имя монтируемого каталога. Демон mountd, выполняющийся на сервере, получает запрос и преобразует полное имя. Он проверяет, является ли указанное имя действительно каталогом, а также может ли оно быть экспортировано клиенту. Если это возможно, демон отправляет успешное ответное сообщение, в котором указывает дескриптор файла этого каталога.

После получения успешного ответа клиент завершает инициализацию структуры vfs. Он вносит в объект закрытых данных vfs имя и сетевой адрес сервера. Затем он размещает rnode и vnode для корневого каталога. Если сервер даст сбой и перезагрузится, система на клиентской машине останется монтированной, но эта информация будет потеряна сервером. Так как клиент посылает верные дескрипторы файлов в запросах к NFS, сервер сделает вывод об успешном проведении операции монтирования ранее и перестроит свои внутренние записи.

Серверу необходимо проверять права доступа к файловой системе NFS при *каждом* запросе к ней. Ему требуется убедиться в том, что файлы экспортируются только определенному клиенту (если запрос изменяет файл, последний должен быть экспортирован на чтение-запись). Чтобы сделать такое испытание эффективным, дескриптор файла содержит пару <inode, generation number> (<индексный дескриптор, генерируемый номер>) экпортируемого каталога. Сервер поддерживает в памяти список всех таких каталогов, что позволяет быстро осуществлять нужные проверки.

10.5.4. Просмотр полных имен

Как результат операции mount, клиент получает дескриптор каталога верхнего уровня. Он получает и другие дескрипторы при просмотре полных имен или после выполнения операций CREATE и MKDIR. Клиент инициализирует операцию просмотра при применении таких системных вызовов, как open, creat или stat.

На клиентской системе операция просмотра начинает свою работу в текущем либо корневом каталоге (в зависимости от того, является ли полное имя относительным или абсолютным) и просматривает по одному компоненту в один момент времени. Если текущим каталогом является каталог системы NFS или происходит пересечение границы монтирования в сторону каталога NFS, операция просмотра вызовет специфическую для NFS функцию VOP_LOOKUP. Функция отправит серверу запрос LOOKUP, в котором передаст дескриптор родительского каталога (сохраненного клиентом в объекте rnode), а также имя компонента для поиска.

Сервер извлекает идентификатор файловой системы из дескриптора и использует его для обнаружения структуры vfs, относящейся к данной системе. После он загружает операцию VFS_VGET, преобразующую дескриптор файла и возвращающую указатель на vnode родительского каталога (размещая в памяти новый объект, если это необходимо). Затем сервер выполняет операцию VNODE_LOOKUP применительно к этому объекту vnode, которая и вызовет функцию для локальной файловой системы. Функция производит поиск в каталоге искомого файла и при обнаружении размещает его объект vnode в памяти (если последнего там нет, разумеется) и возвращает на него указатель.

Далее сервер вызывает операцию VOP_GETATTR по отношению к vnode найденного файла, а также функцию VOP_FID, создающую для него дескриптор. Завершающим действием сервера является создание ответного сообщения, содержащего статус, дескриптор файла компонента и атрибуты этого файла.

После того как клиент получит ответное сообщение, он размещает новые rnode и vnode для найденного файла (конечно, если файл искался и ранее, клиент может уже обладать его vnode). Затем он копирует дескриптор и атрибуты файла в rnode и начинает поиск следующего компонента.

В системе NFS используется технология поиска одного компонента за один прием. Это происходит весьма медленно и требует отправки нескольких сообщений RPC. Клиент может частично уменьшить количество запросов при помощи кэширования информации о каталоге (см. раздел 10.7.2). Хотя на первый взгляд кажется, что отправка имени в одном запросе LOOKUP будет более эффективной, такой подход обладает несколькими важными недостатками. Во-первых, клиент может смонтировать файловую систему в одном из каталогов, являющемся частью этого имени, следовательно, серверу необходимо знать обо всех точках монтирования на клиентской машине, чтобы корректно произвести поиск. Во-вторых, просмотр имени целиком требует, чтобы сервер понимал семантику UNIX. Однако это противоречит цели достижения независимости от операционной системы и политике запоминания состояний. Серверы NFS перенесены на огромное число систем, в том числе на VMS и Novell NetWare, в которых приняты совершенно иные соглашения об именах. Полные имена используются только в операции монтирования, привлекающей для своей работы другой протокол.

10.6. Семантика UNIX

В системах UNIX проверка прав доступа происходит при первом открытии файла процессом, но не при каждой операции чтения или записи. Представьте, что после того как пользователь открыл некий файл для записи, его владелец изменил права доступа к этому файлу на «только для чтения». В системах UNIX пользователь может продолжать писать в файл до тех пор, пока не закроет его. В файловой системе NFS отсутствует поддержка концепции открытых файлов, и права доступа проверяются при каждом чтении или записи. В описанном случае система вернет ошибку, которую клиент никак не ожидает увидеть.

Хотя и не существует способа предупреждения подобной ситуации, в NFS ее можно обойти. Сервер всегда разрешает владельцу файла осуществлять чтение или запись в него независимо от прав. Это противоречит семантике UNIX, так как позволяет владельцам модифицировать свои файлы, защищенные от записи. После открытия файла клиентом операция LOOKUP возвращает наряду с дескриптором и атрибуты файла. Атрибут содержит права

доступа на момент вызова open. Если файл защищен от записи, вызов open вернет клиенту код ошибки EACCESS («доступ запрещен»). Важно упомянуть, что механизмы защиты сервера зависят от правильности поведения клиента. В данном случае проблема не является серьезной, поскольку относится только к праву доступа на запись владельца файла. Основные проблемы защиты NFS будут обсуждаться более подробно в разделе 10.9.

10.6.2. Удаление открытых файлов

Если процесс UNIX удаляет файл, открытый другим процессом (или тем же процессом повторно), ядро просто помечает такой файл как удаленный и убирает его вхождение в родительском каталоге. Хотя другие процессы и не могут больше открыть такой файл, процессы, открывшие его, могут продолжать им пользоваться. Только после закрытия файла последним из процессов ядро системы физически удаляет такой файл. Описанное средство применяется некоторыми утилитами для временных файлов.

Для системы NFS это представляет определенную проблему, так как сервер не обладает информацией об открытых файлах. Для того чтобы обойти создавшуюся трудность, необходимо изменить код клиента NFS, потому что именно клиент *не* знает об открытых файлах. Когда клиент обнаруживает попытку удаления открытого файла, он автоматически подменяет вызванную операцию на RENAME, давая файлу новое имя и место на диске. Обычно клиент использует необычные и длинные имена, которые скорее всего не будут конфликтовать с уже существующими. После закрытия файла последним процессом клиент создает запрос REMOVE для его удаления.

Описанная методика хороша, если процесс, открывающий файл, выполняется на той же машине, что и удаляющий его. Однако не существует защиты против удаления файла процессом другого клиента (или сервера). Если удаление произойдет, пользователь получит неожиданное сообщение об ошибке («устаревший дескриптор файла») при последующих попытках чтения или записи в этот файл. Еще одна неувязка может возникнуть в случае сбоя в работе клиента, произошедшего после операции RENAME и до REMOVE: на сервере останется лишний файл («мусор»).

10.6.3. Чтение и запись

В системах UNIX системные вызовы read и write блокируют vnode файла перед началом ввода-вывода. Такой подход делает операции чтения и записи неделимыми. Если по отношению к одному и тому же файлу одновременно будет вызвана функция write, ядро выполнит их последовательно. Вторая операция чтения начнет работу только после завершения первой. Точно так же система гарантирует отсутствие изменений в файле при его чтении. Обра-

батывает блокировку локальный зависимый от файловой системы код в контексте одной операции vop_rdwr.

В случае файла NFS клиентский код при одновременной попытке доступа к файлу двух процессов на локальной машине делает эти операции последовательными. Однако если это окажутся процессы, запущенные на разных машинах, они будут пытаться получить доступ к серверу независимо друг от друга. Операция read или write может потребовать нескольких запросов RPC (максимальный размер сообщения RPC равен 8 192 байт), при том что сервер, не хранящий информацию о состояниях, не поддерживает блокировку между этими запросами. Система NFS не предлагает защиты в случае совмещения запросов ввода-вывода.

Взаимодействующие процессы могут использовать для защиты файлов или их частей протокол сетевого администратора блокировки (NLM). Этот протокол предлагает только рекомендательную блокировку. Отсюда следует, что процесс всегда может обойти ее и получить доступ к файлу, если он этого захочет.

10.7. Производительность NFS

Одной из задач, поставленной при разработке файловой системы, было достижение производительности, сравнимое с локальными системами. Существуют несколько методик тестирования, пытающихся имитировать обычную загрузку файловой системы NFS, наиболее популярными из них стали LADDIS [40] и nhfsstone [18]. В этом разделе главы описываются важнейшие проблемы, связанные с производительностью работы NFS, а также меры по их устранению.

10.7.1. Узкие места

Серверы NFS обычно представляют собой мощные машины с кэшами большого объема и быстрыми дисками, что компенсирует время, затраченное на отправку и получение запросов RPC. Однако в системе NFS существуют некоторые участки, напрямую приводящие к ухудшению производительности.

Являясь протоколом, не запоминающим состояния, NFS требует записи всех данных до отправки ответного сообщения. Эта информация включает не только изменения метаданных файла (индексных дескрипторов и блоков косвенной адресации), но также и содержимого самого файла. В итоге любой запрос NFS, производящий произвольные изменения в файловой системе (например, WRITE, SETATTR или CREATE), работает очень медленно.

Отправка атрибутов файла требует одного вызова RPC на каждый файл. В результате такие команды, как ls -l применительно к каталогу, приводят к большому количеству запросов RPC. В случае локальной файловой системы

такая операция выполняется очень быстро, так как индексные дескрипторы хранятся в кэше, поэтому вызовам stat достаточно ссылки в памяти.

Если сервер недостаточно скоро произведет ответ на запрос, клиент пошлет его заново, считая, что произошел отказ сервера или запрос был утерян в сети. Обработка повторного сообщения также увеличивает загрузку сервера и может привести к возникновению проблем. Возникает каскадный эффект, влекущий в результате то, что сервер «падает» под натиском входящего трафика.

Рассмотрим некоторые способы решения проблем производительности работы NFS.

10.7.2. Кэширование на стороне клиента

Если каждая операция над удаленным файлом будет требовать доступа к нему по сети, производительность NFS окажется очень низкой. Именно поэтому большинство клиентов NFS кэшируют блоки и атрибуты файлов. Блоки кэшируются в буферном кэше, а файловые атрибуты — в объектах rnode. Эта операция несет в себе определенную опасность, так как клиент не знает, является ли содержимое его кэша действительным, когда просто обращается к серверу при каждом его использовании.

Клиенты выполняют те же предупредительные действия для уменьшения вероятности попадания на устаревшие данные. Ядро поддерживает срок устаревания в rnode, при помощи которого отслеживается давность кэширования атрибутов. Обычно клиент кэширует атрибуты после получения их от сервера на 60 или менее секунд. Если к ним произойдет обращение после истечения срока, клиент запросит новые сведения от сервера. Примерно таким же образом обрабатываются блоки данных: клиент удостоверяет целостность кэша путем проверки *времени изменения*. Если значение, установленное после чтения данных с сервера, не изменилось, данные считаются действительными. Клиент может продолжать использовать величину, хранимую в кэше, или вызвать GETATTR, если срок хранения истек.

Кэширование на стороне клиента положительно влияет на производительность системы. Описанные упреждающие меры благотворны, но, к сожалению, не защищают от проблем целостности. К примеру, они могут привести к состязательности, что подробно описано в работах [17] и [12].

10.7.3. Отложенная запись

Требования синхронности записи в системе NFS относятся только к серверной части. Клиент может отложить операцию записи, следовательно, если данные будут утеряны в результате его выхода из строя, пользователь будет знать об этом. В любом случае правила предписывают использовать асинхронную запись для полных блоков (передавать запрос WRITE, но не ждать

ответа) и отложенную запись для фрагментов блоков (производя WRITE позже). Большинство реализаций UNIX сбрасывают изменения на сервер каждые 30 секунд, а также при закрытии файла. Обработку таких запросов ведут клиентские демоны biod.

Хотя серверу необходимо выполнить операцию записи на надежный носитель перед отправкой ответного сообщения, он не обязан сохранять данные именно на диске. Сервер может использовать специализированную аппаратуру, гарантирующую отсутствие потерянных данных при своем отказе. Например, некоторые серверы имеют специальную энергонезависимую память, с подпиткой от батарейки, называемую NVRAM. Сервер сбрасывает буферы NVRAM на диске через определенные промежутки времени. Это позволяет ему быстро реагировать на запросы записи, так как перенос данных на NVRAM происходит намного быстрее, чем на диск. Дисковый драйвер может оптимизировать порядок записи с NVRAM на диск, добиваясь минимизации перемещения головок. Более того, несколько изменений одного и того же буфера может быть записано на диск одной операцией. Реализации системы NFS с применением NVRAM описаны в работах [9] и [18].

В книге [13] демонстрируется методика, называемая *сбором записей* (write-gathering), уменьшающая проблему появления узкого места при записи без использования дополнительных аппаратных устройств. Она построена на том факте, что типичные клиенты NFS загружают определенное количество демонов biod для обработки операций записи. После того как клиент откроет файл и произведет в него запись, ядро сбросит изменения в кэш и пометит его для отложенной записи. Когда клиент закроет файл, ядро передаст блоки на сервер. Если на клиентской машине загружено достаточное количество демонов biod, они могут вести запись параллельно. В результате серверы часто получают сразу несколько запросов на запись одного и того же файла.

При использовании технологии сбора записей сервер не производит обработку запросов WRITE немедленно. Вместо этого он обычно задерживает операцию записи на небольшой промежуток времени в надежде дождаться остальных запросов WRITE, относящихся к тому же файлу. Затем сервер собирает запросы на запись одного файла воедино и обрабатывает их все вместе. Ответные сообщения сервер генерирует по завершении обработки группы запросов. Описанная методика эффективна при использовании клиентами демонов biod и наиболее оптимальна в случае большого количества запущенных копий этой программы. На первый взгляд при этом увеличивается задержка обработки отдельных запросов, однако рост производительности происходит за счет меньшего количества дисковых операций на сервере. Например, если сервер собрал n запросов на запись файла, он может сбросить изменения на диск, используя только одну операцию записи данных и одну операцию записи индексного дескриптора (в противовес n вызовов каждой из перечисленных операций). Сбор записей увеличивает пропускную способность сервера и уменьшает загруженность дисков. Технология

наращивает задержку обработки отдельного запроса, сокращая при этом среднее время записи.

Некоторые серверы оборудованы устройством бесперебойного питания (uninterruptable power supply, UPS), позволяющим сбросить дисковые блоки, размещенные в кэше, в случае возникновения проблем с электропитанием. Другие серверы просто игнорируют требование синхронной записи, принятое в NFS, считая, что крах системы происходит весьма редко. Разнообразие решений и обходных путей описанной проблемы указывает на ее степень серьезности. Протокол NFSv3, описываемый в разделе 10.10, разрешает клиентам и серверам безопасно использовать асинхронную запись.

10.7.4. Кэш повторных посылок

Клиенты RPC с целью сохранения надежности передачи отправляют одни и те же запросы повторно до тех пор, пока не получат ответное сообщение. Обычно период ожидания, после которого последует повтор запроса, не слишком велик (примерно 1–3 секунды). Он увеличивается экспоненциально при каждом последующей посылке сообщения. Если по достижении определенного количества повторов клиент не получит ответ, он может отправить новый запрос (в некоторых реализациях системы), который будет идентичен предыдущему, но будет обладать другим идентификатором xid.

Повторная передача запросов возникает в результате потерь пакетов (составляющих как сам запрос, так и ответное сообщение) или при отсутствии возможности сервера ответить на запрос. Ответное сообщение на первый запрос часто приходит уже после того, как клиент отправляет копию. Многократных вторичных передач следует ожидать при отказе сервера или большой перегрузке сети.

Сервер должен обладать средствами отслеживания повторных одинаковых сообщений и правильно обрабатывать их. В файловой системе NFS запросы можно разделить на две категории, идемпотентные и неидемпотентные [12][1]. Запросы первого типа, такие как READ или GETATTR, могут быть выполнены дважды без каких-либо отрицательных последствий. Повтор неидемпотентных запросов способен привести к некорректному поведению. Все запросы, изменяющие файловую системы любым способом, являются неидемпотентными.

В качестве примера приведена последовательность событий, могущих возникнуть в результате дублирования операции REMOVE:

1. Клиент отправляет запрос на удаление файла (REMOVE).
2. Сервер успешно удаляет файл.
3. Сервер отправляет ответное сообщение об удачном завершении операции удаления, однако это сообщение теряется в сети.

[1] Идемпотентность означает, что повышенная активность одной задачи не оказывает негативного воздействия на производительность системы в целом. — *Прим. ред.*

4. Клиент посылает повторный запрос на удаление файла.
5. Сервер обрабатывает запрос REMOVE, результатом выполнения которого станет ошибка (так как файл уже был удален ранее).
6. Сервер посылает клиенту сообщение об ошибке, которое успешно доходит до клиента.

В результате клиент получает сообщение об ошибке, хотя реально операция REMOVE была успешно завершена.

Повторная отправка запросов отрицательно влияет на производительность сервера, потому что ему приходится тратить большое количество времени на выполнение излишней работы. Ситуация усугубляется тем, что обычно пересылка запросов происходит при перегрузке сервера, который в таком случае и так работает медленнее обычного.

Однако в любом случае повторная пересылка запросов должна отслеживаться и корректно обрабатываться. Для этого на сервере поддерживается кэш недавних запросов. Запросы могут быть идентифицированы как повторные по совпадению xid, номеров процедур и идентификаторов клиентов с аналогичными параметрами запросов, «лежащих» в кэше (сравнение одного лишь xid не всегда является достаточным, так как некоторые клиенты способны генерировать одинаковые xid в запросах, исходящих от разных пользователей). Такой кэш называется *кэшем пересылок* или *кэшем идентификаторов xid*.

В оригинальной реализации Sun кэш поддерживается только для запросов CREATE, REMOVE, LINK, MKDIR и RMDIR. Проверка кэша происходит при неудачном завершении обработки запроса с целью определения, не является ли причиной ошибки повторный запрос. Если это так, клиенту посылается ответное сообщение об удачном завершении операции. Такой подход не является достаточно эффективным в силу того что позволяет преодолеть лишь некоторые трудности и даже становится причиной возникновения новых проблем целостности. Более того, описанная методика не влияет на производительность, поскольку сервер проверяет кэш уже после обработки запроса.

В [12] проведен подробный анализ проблем отслеживания повторных запросов. На ее основе компания Digital разработала вариант кэша идентификаторов xid, используемый в системе ULTRIX. В этой реализации кэшируются *все* запросы, и их проверка происходит до обработки новых посылок. Каждое вхождение кэша содержит данные, используемые для их идентификации (идентификатор клиента, xid и номер процедуры), а также поле состояния (state) и времени в формате timestamp[1]. Если сервер обнаружит в кэше запрос, имеющий состояние in progress (в обработке), то он просто удалит дубликат. Если запрос имеет состояние done (обработка завершена), сервер отвергнет запрос в том случае, если значение timestamp укажет на недавно завершившуюся обработку (с окном временного хранения (throwaway window), установленным в значение 3–6 секунд). Если время окажется большим, чем

[1] Число секунд, прошедших с 1 января 1970 года до момента модификации файла. — *Прим. ред.*

установленное значение, сервер произведет обработку идемпотентного запроса. В случае неидемпотентного запроса сервер проверит, не изменялся ли файл уже после оригинального значения timestamp. Если изменений не происходило, клиенту будет отправлено сообщение об удачной обработке запроса, в противном случае запрос будет обработан повторно. Элементы кэша располагаются в порядке «последнего, недавно использованного элемента». Это дает возможность обрабатывать повторные сообщения сразу же, если клиент продолжает отправлять серверу дубликаты.

Кэш идентификаторов xid помогает предотвратить выполнение некоторых дополнительных операций, что положительно сказывается не только производительности системы, но и на корректности работы сервера. Можно развить предложенную технологию. Если сервер будет кэшировать ответное сообщение вместе с информацией о его xid, то он сумеет обрабатывать повторные запросы путем отправки ответа, взятого из кэша. Дубликаты, накопленные окном временного хранения, могут быть удалены единовременно. Это уменьшит количество лишних операций даже в случае обработки идемпотентных запросов. Описанная технология требует применения кэша большого объема, позволяющего хранить сообщения целиком. Некоторые ответные сообщения, например создаваемые в результате запросов READ или READDIR, могут иметь значительный размер. Такие запросы целесообразно исключать из кэша и обрабатывать их только при необходимости.

10.8. Специализированные серверы NFS

Большинство поставщиков систем UNIX предлагают в качестве файловых серверов обычные рабочие станции, оборудованные дополнительными дисками, оперативной памятью, а также сетевыми картами. Такие системы работают под управлением варианта UNIX, разработанного поставщиками и предназначенного для применения в мультизадачных средах. Далеко не всегда подобные решения подходят для высокопроизводительной службы NFS. Однако существуют производители, предлагающие специализированные серверы NFS, в которых либо добавлены в стандартную UNIX некоторые дополнительные возможности, либо используется собственная операционная система. В этой главе описываются две подобные архитектуры.

10.8.1. Архитектура функциональной многопроцессорной системы Auspex

Компания Auspex Systems пришла на рынок высокопроизводительных серверов NFS со своей разработкой, называемой функциональной многопроцессорной системой (Functional multiprocessing (FMP) system) NS5000. Архитектура

FMP подразумевает, что служба NFS состоит из двух важнейших подсистем — сетевой и подсистемы памяти.[1] Машина построена на процессорах Motorola 680x0, каждый из которых выделен одной из подсистем и использует общую магистраль. В таких процессорах выполняется небольшое *функциональное многопроцессорное ядро* (functional multiprocessing kernel, FMK). Взаимодействие процессоров друг с другом осуществляется путем высокоскоростного обмена сообщениями. На одном процессоре (например, на 68020) выполняется модифицированная версия системы Sun OS 4.1 (с добавлением поддержки FMK), и он выступает в роли управляющего. Архитектура системы показана на рис. 10.5.

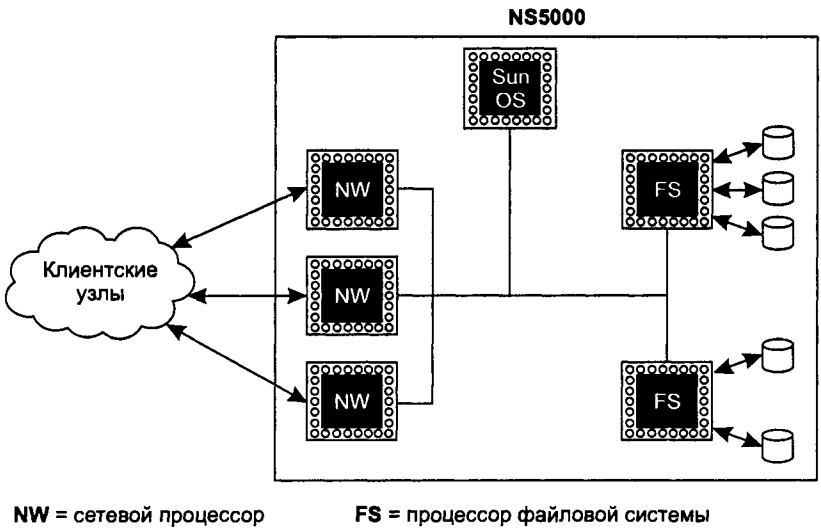


Рис. 10.5. Архитектура системы Auspex NS5000

Специальная реализация UNIX может обращаться к каждому из процессоров напрямую. Такое взаимодействие происходит через стандартный сетевой драйвер if. Специализированная локальная файловая система, реализующая интерфейс vfs, служит для обращения к процессорам файловой системы. Процессор также обладает возможностью прямого доступа к накопителям через драйвер устройства, связанного с диском Auspex и преобразующего дисковые запросы ввода-вывода в сообщения FMK. Такой подход позволяет использовать различные утилиты, такие как fsck или newfs, не внося в них каких-либо изменений.

Обычный запрос NFS обходит все процессоры UNIX. Он поступает на сетевой процессор, в котором поддержаны уровни IP, UDP, RPC и NFS. Затем запрос попадает к процессору файловой системы, который может направить

[1] В оригинальной версии третья подсистема обеспечивала функции хранения. В последних реализациях для хранения отдельные процессоры уже не выделяются.

запросы ввода-вывода процессору устройств хранения. Сетевой процессор периодически отправляет клиентам ответные сообщения.

Ядро FMK поддерживает небольшой набор примитивов, в том числе легковесные процессы, передачу сообщений и динамическое выделение памяти. Избавленное от тяжести «груза», составляющего традиционное ядро UNIX, ядро FMK весьма быстро производит переключение контекста и передачу сообщений. К примеру, FMK не поддерживает управление памятью и процессы системы никогда не подвергаются завершению.

Разработанная архитектура является основой для высокоскоростных серверов NFS. Она позволила компании Auspex Systems стать поставщиком передовых реализаций систем NFS. Позже компания была смещена с лидирующих позиций кластерными серверами NFS, предлагаемыми такими поставщиками, как Sun Microsystems и Digital Equipment Corporation.

10.8.2. Сервер HA-NFS фирмы IBM

В работе [2] описывается прототип сервера HA-NFS, разработанного компанией IBM. В этой системе проблема доступности службы NFS поделена на три отдельные категории: надежности в сети, надежности дисков и надежности сервера. Система использует зеркалирование[1] и репликацию в сети (опционально), а также два взаимодействующих сервера с целью увеличения отказоустойчивости.

Архитектура HA-NFS показана на рис. 10.6. Каждый сервер имеет два сетевых интерфейса и, соответственно, два IP-адреса. Один из интерфейсов выбирается сервером основным и используется при нормальной работе. Второй интерфейс применяется только в случае сбоя второго сервера.

Система HA-NFS использует оснащенные двумя портами диски, соединенные с обоими серверами посредством общей шины SCSI. Для каждого диска задан основной сервер, производящий доступ к нему при нормальном функционировании. Таким образом, все диски разделены на две группы, по одной для каждого сервера.

Серверы взаимодействуют друг с другом при помощи механизма HeartBeat, вырабатывающего сообщения через определенный период времени. Если один сервер не получает такое сообщение от другого, запускается серия тестов, призванных удостовериться в действительности поломки сервера. Если сбой доказан, инициализируется процедура восстановления. Исправный сервер получает управление дисками второго сервера и устанавливает IP-адрес второго сетевого интерфейса равным адресу основного интерфейса неработающего сервера. Это позволяет ему получать и обрабатывать сообщения, адресованные второму серверу.

---

[1] Удвоение, поскольку используется аппаратный контроллер. — *Прим. ред.*

Описанные действия производятся незаметно для клиентов, которые могут ощутить только лишь снижение производительности. «Упавший» сервер не отвечает лишь в течение периода работы процедуры восстановления. После ее завершения рабочий сервер может работать медленнее, так как ему приходится справляться с нагрузкой, рассчитанной на два сервера. Однако при этом не происходит отказа службы.

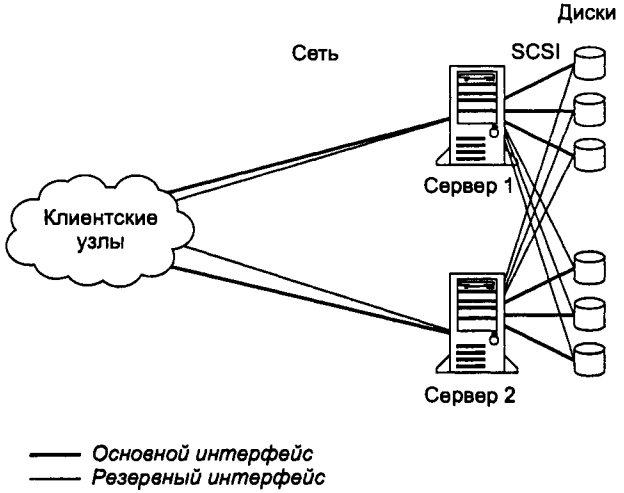


Рис. 10.6. Конфигурация HA-NFS

Оба сервера работают под управлением операционной системы IBM AIX с файловой системой, ведущей журналы метаданных. Для записи операций NFS в HA-NFS добавлена информация о запросах RPC. После сбоя сервер повторяет операции, внесенные в журнал с целью «воскрешения» файловой системы и приведения ее в целостное состояние, а также восстанавливает кэш повторных запросов из информации о RPC, хранящейся в журнале. Такой подход позволяет предотвратить нарушения целостности файловой системы. Серверы также обмениваются информацией о клиентах, производящих блокировку файлов путем отправки запросов, использующих NSM и NLM [3]. Эти действия обеспечивают восстановление состояния менеджера блокировки после отказа одного из серверов.

Существуют два метода, позволяющие сделать замену IP-адреса прозрачной для клиентов. Один из них основан на использовании специальных сетевых интерфейсных карт, позволяющих изменять аппаратный адрес. После сбоя сервер изменяет одновременно IP-адрес и аппаратный адрес второй карты на адреса, принадлежавшие основному интерфейсу «упавшего» сервера. В случае отсутствия описанного оборудования сервер может использовать для переназначения пары <аппаратный адрес, IP-адрес> клиентов преимущества некоторых свойств *протокола разрешения адресов* (address resolution protocol, ARP).

10.9. Защита NFS

Сетевое приложение обычно не может предложить уровень безопасности, характерный для локальной файловой системы. Несмотря на то, что система NFS поддерживает семантику UNIX, ее механизмы защиты являются не совсем приемлемыми. В этом разделе вы увидите описание основных «брешей» в защите NFS, а также некоторых возможных решений возникающих проблем.

10.9.1. Контроль доступа в NFS

В системе NFS проверка доступа осуществляется при монтировании системы и обработке каждого запроса к ней. Серверы поддерживают список экспорта (export list), в котором содержится информация о клиентских машинах, обладающих правом доступа к экспортируемым файловым системам, а также о типе предоставляемого доступа (только чтение или чтение-запись). При каждом запросе клиента на монтирование файловой системы серверный демон `mountd` проверяет этот список и запрещает доступ не включенным в него клиентам. Система не поддерживает ограничений доступа для определенных пользователей. Это означает, что любой пользователь зарегистрированной клиентской машины имеет право монтировать файловую систему.[1]

Клиент отправляет в каждом запросе NFS данные о полномочиях преимущественно в форме `AUTH_UNIX`. Такая информация содержит идентификаторы пользователя и группы, являющихся владельцами процесса, сделавшего запрос. Сервер NFS использует полученную информацию для инициализации структуры полномочий, предоставляемой локальным файловым системам для управления доступом.

Для того чтобы описанная схема работала, сервер и все его клиенты должны использовать пространство пар <UID, GID> сообща. Это означает, что каждый полученный идентификатор UID должен относиться к одному и тому же пользователю на всех машинах, разделяющих между собой файловые системы NFS. Если пользователь П1, работающий на машине К1, имеет идентификатор, совпадающий с UID пользователя П2 на компьютере К2, сервер не сможет различить таких пользователей и предоставит им обоим полный доступ к файлам друг друга.

Озвученная проблема является на сегодняшней день основной трудностью, с которой сталкиваются рабочие группы. Обычно в таких группах каждый пользователь имеет собственную рабочую станцию, а общие файлы хранятся на сервере NFS. Во многих случаях на нем размещаются и каталоги входа в систему. Каждый пользователь, скорее всего, обладает привилегиями

[1] Во многих реализациях клиентской части NFS право монтирования удаленной файловой системы предоставлено только привилегированным пользователям.

суперпользователя на собственной рабочей станции, следовательно может создавать учетные записи с одинаковыми значениями <GID, UID>. При этом вероятна ситуация, когда злоумышленник, обладающий совпадающей последовательностью, получает доступ к чужим файлам на сервере, причем их владелец даже не подозревает об этом. Злоумышленник может завладеть доступом без применения каких-либо сложных программ или внесения изменений в ядро или сеть. Единственным методом защиты от возникновения подобных ситуаций является ограничение доступа к системе NFS только для известных, доверенных клиентов, действия которых можно отслеживать. Приведенный пример ясно показывает недостаточность средств защиты в файловой системе NFS.

Существуют и другие методы проникновения в систему NFS. Так как NFS сильно зависит от данных, пересылаемых по незащищенным сетям, не так сложно написать программу, имитирующую клиентов NFS. Такая программа будет создавать поддельные идентификационные данные, возможно даже отсылаемые с «различных» машин, что может привести к получению доступа пользователями, не имеющими привилегий суперпользователя или работающими на компьютерах, доступ с которых к серверу NFS запрещен.

10.9.2. Переназначение групповых идентификаторов

Существует несколько методик защиты от нападений, описанных в предыдущем подразделе. Первой линией обороны является *переназначение групповых идентификаторов* (GID). При этом вместо использования общего пространства имен <GID, UID> на сервере и всех его клиентах сервер поддерживает для каждого клиента отдельную карту преобразования. Карта определяет преобразования удостоверений, полученных по сети при идентификации, на привилегии, используемые сервером. Они также описываются набором <GID, UID>, который, однако, может быть отличным от набора, присланного клиентом.

Приведем пример. В карте замещений может быть указано на необходимость изменения UID 17 клиента K1 на идентификатор 33 на сервере, притом что UID 17 клиента K2 замещается на сервере на идентификатор 17 и т. п. Сервер также может определять преобразования и для групповых идентификаторов. Карта поддерживает некоторое количество замещений, устанавливаемых по умолчанию (*шаблонов*, wildcards), наподобие «*для заданного набора надежных клиентов не требуется никакого преобразования*». Обычно карта содержит данные о действиях при несовпадении поступивших на сервер удостоверений. Правило по умолчанию предоставляет таким неопознанным мандатам права специального пользователя nobody, имеющего доступ только к файлам, доступным для всех.

Преобразование UID может быть реализовано на уровне RPC. Это означает, что такие преобразования будут действительны для любой службы, основанной на протоколе RPC, и будут применены до интерпретации запроса файловой системой NFS. Так как подобные преобразования требуют дополнительных действий, производительность служб RPC уменьшится. Причем это произойдет даже в том случае, если для определенной службы описанный уровень защиты является излишним.

Альтернативным решением может являться реализация преобразования UID на уровне NFS, например путем описания карт прямо в файле /etc/exports. Это даст серверу возможность использовать для различных файловых систем различные карты преобразования. Например, если сервер экспортирует каталог /bin в режиме «только для чтения», а каталог /usr — на чтение-запись, он может создавать карты UID исключительно для /bin, так как только этот каталог является потенциальным хранителем важных файлов, защищаемых от несанкционированного доступа. Недостаток такого варианта — возможность реализации собственных карт преобразования (или других механизмов защиты) каждой отдельной службой RPC, с дублированием кода и производимых операций.

Лишь некоторые реализации NFS предлагают какую-либо форму преобразований идентификаторов пользователей. Защищенные версии NFS основаны обычно на проверке методами AUTH_DES или AUTH_KERB, описанными в разделе 10.4.2.

10.9.3. Переназначение режима доступа root

Похожая проблема возникает при использовании клиентскими машинами режима доступа root. Очевидно, что суперпользователи любой клиентской системы не должны обладать доступом root к файлам сервера. Традиционным решением проблемы является причисление таких пользователей в nobody. Альтернативный вариант предполагает указание в файле /etc/exports другого идентификатора пользователя, на который будут заменены все клиенты с правами root.

Такой подход защищает от очевидных конфликтов с доступом клиентов в режиме суперпользователя, но приводит к неожиданному эффекту. Пользователи, вошедшие в свои системы в качестве root (или же выполняющие привилегированную программу, установленную в режиме setuid), обладают правами на доступ к файлам NFS меньшими, чем у обычных пользователей. К примеру, они могут не иметь доступа к собственным файлам на сервере.

Многие из описанных проблем относятся не только к одной лишь файловой системе NFS. Средства безопасности традиционной ОС UNIX разработаны для изолированной многопользовательской среды и являются достаточными только для автономных систем (не подсоединенных к сети). Появление сетей, в которых обычно одни узлы доверяют другим, показало недостаток

средств защиты и существование в ней «дыр», что стало причиной разработки служб безопасности и аутентификации. Наиболее прижилась в системах UNIX служба Kerberos [32]. Более подробное изложение проблем защиты компьютерных сетей выходит за рамки книги.

10.10. NFS версии 3

Система NFSv2 приобрела большую популярность. Она была перенесена на самые различные аппаратные платформы и операционные системы. Это помогло выявить ее недостатки. Некоторые из найденных проблем были решены впоследствии, но, к сожалению, часть из них являются следствием особенностей самого протокола и не могут быть устранены. В 1992 году разработчики из нескольких компаний собрались в Бостоне для создания новой, третьей версии протокола NFS [23], [37]. Появившаяся в результате система постепенно стала поддерживаться коммерческими вариантами UNIX. Корпорация DEC встроила поддержку NFSv3 в свою операционную систему DEC OSF версии 3[1], позже к ним присоединилась компания Silicon Graphics и другие поставщики ОС. Рик Маклем (Rick Macklem) из университета города Гуэльф сделал доступной реализацию системы для 4.4BSD, которую можно получить через *анонимный FTP-сервер* по адресу snowhite.cis.uoguelph.ca/pub/nfs[2].

В файловой системе NFSv3 были сняты многие ограничения, характерные для ее предыдущей версии. Основной проблемой, влияющей на производительность работы NFSv2, была необходимость сбрасывать все изменения на постоянный носитель перед отправкой ответного сообщения. Причиной этому является использование протокола без запоминания состояний, в котором не существует иных возможностей подтверждения удачного завершения передачи данных. Система NFSv3 разрешает асинхронную запись при помощи нового запроса COMMIT. Если клиенту необходимо произвести синхронную запись в файл NFS, ядро системы пошлет запросы WRITE на асинхронную запись. Сервер может сохранить данные в локальном кэше и сразу же после этого передать ответное сообщение клиенту. Ядро клиентской системы сохраняет копию данных до тех пор, пока процесс не закроет обрабатываемый файл. После этого ядро посылает серверу запрос COMMIT, в ответ на который сервер сбросит данные на диск и передаст сообщение об успешном завершении операции. После получения ответа на запрос COMMIT клиент может удалить локальную копию обрабатываемых ранее данных.

Асинхронная запись не является обязательным средством NFSv3, поэтому некоторые специфические клиенты или серверы могут не поддерживать

---

[1] На сегодняшний день система DEC OSF/1 известна под названием Digital UNIX.

[2] Конечно, по прошествии нескольких лет ссылка устарела, теперь с сайта канадского университета можно загрузить только следующую, четвертую, версию среды (NFSv4 — ftp://snowhite.cis.uoguelph.ca/pub/nfsv4). — *Прим. ред.*

такую возможность. Например, если клиенту не нужны средства кэширования данных, то ему позволено продолжать использовать устаревший метод синхронной записи, при котором сервер должен сразу же сохранять все данные на носитель. Серверы могут самостоятельно выбрать синхронную запись даже при обработке асинхронных запросов. В этом случае в ответном сообщении делается соответствующая пометка.

Еще одной немаловажной проблемой NFSv2 является использование 32-разрядных полей для указания размера файла, а также смещений при чтении и записи. Это число позволяет поддерживать файлы размером не более 4 Гбайт, что ограничивает возможности системы. Некоторым приложениям необходимо работать с файлами большей величины, поэтому для таких программ NSFv2 абсолютно не подходит. В NFSv3 эти поля расширены до 64 разрядов, что позволяет поддерживать файлы размером до $1{,}6\times10^{19}$ байт (или миллиарда терабайтов).

Протокол NFSv2 может стать причиной возникновения множества запросов LOOKUP и GETATTR. К примеру, при чтении каталога командой ls -l происходит передача запроса серверу, в ответ на который возвращается список файлов, содержащихся в каталоге. Затем клиент передает запросы LOOKUP и GETATTR по отношению к файлам из полученного списка. Для больших каталогов подобные операции способны существенно увеличить объем сетевого трафика.

В системе NSFv3 добавлена операция READDIRPLUS, возвращающая имена, дескрипторы и атрибуты всех файлов заданного каталога. Она позволяет заменить одним вызовом целую серию запросов системы NFSv2. Однако READDIRPLUS необходимо применять с осторожностью, как так в ответ обычно передается большой объем данных. Если клиенту необходима информация всего об одном файле, целесообразнее использовать последовательность вызовов NSFv2.

Реализации системы, поддерживающие протокол NFSv3, всегда совместимы и с предыдущей его реализацией, NSFv2. При этом клиент и сервер обычно используют наиболее свежую из поддерживаемых версий. Клиент отправляет первый запрос серверу, созданный на основе самой высокой версии протокола. Если сервер не распознает запрос, клиент отправит ему еще один, задействуя другую, более старую версию протокола NFS. Клиент будет «понижать» версию до тех пор, пока не дойдет до поддерживаемой обеими системами.

Понять, насколько эффективен протокол NFSv3, нам поможет только время. Пользователи NFS ожидали описанных нововведений. Они помогли увеличить производительность системы. В третьей версии были решены некоторые незначительные проблемы, присущие NSFv2. Некоторые из изменений отрицательно повлияли на производительность NFSv3, однако преимущества асинхронной записи и нового запроса READDIRPLUS полностью ее компенсируют.

10.11. Remote File Sharing (RFS)

Корпорация AT&T представила файловую систему Remote File Sharing (RFS) в SVR3 UNIX. Система позволила получать доступ к удаленным файлам по сети. Несмотря на то, что основная задача файловой системы RFS сходна с NFS, их архитектура и практическая реализация сильно отличаются друг от друга.

Основной целью разработки RFS было предоставление полного прозрачного доступа к удаленным файлам и устройствам при соблюдении семантики, принятой в UNIX. Это означает поддержку всех типов файлов, в том числе файлов устройств и именованных каналов, а также блокировку записей и файлов. Система также должна оставаться совместимой с UNIX на двоичном уровне таким образом, чтобы разработчикам прикладных программ не требовалось переписывать их заново для использования в RFS. Файловая система не должна зависеть от конкретно используемой сети, следовательно, файлы RFS должны быть одинаково доступны как в локальных (LAN), так и территориально-распределенных сетях (WAN).

Первая реализация RFS могла быть перенесена на другие аппаратные платформы только в рамках SVR3 UNIX. Причиной этого ограничения является использование в RFS механизма *переключения файловых систем*, относящегося к SVR3. В следующей версии операционной системы, SVR4, разработчики интегрировали поддержку интерфейса vnode/vfs и RFS. Это дало возможность распространить файловую систему RFS на многие варианты UNIX. В этой главе мы затронем преимущественно реализацию RFS, основанную на vnode.

10.12. Архитектура RFS

Файловая система RFS построена на архитектуре «клиент-сервер». Сервер предоставляет (или экспортирует) каталоги на своих дисках, а клиенты производят их монтирование. Любая машина в сети может выступать в роли как клиента, так и сервера (или выполнять сразу обе функции). Перечисленные свойства характерны и для системы NFS, однако далее будут показаны отличительные особенности RFS. *Файловая система RFS полностью поддерживает запоминание состояний, которое необходимы для снятия противоречий с принятой в UNIX семантикой открытых файлов.* Этот факт значительно влияет как на особенности реализации, так и на функциональность системы.

Разработчики встроили в систему надежную транспортную службу виртуальных соединений, такую как TCP/IP. Каждая пара «клиент-сервер» использует одно виртуальное соединение, выделяемое при операции монтирования. Если в дальнейшем клиент произведет монтирование других каталогов сервера, все подключаемые каталоги будут разделять то же соединение. Канал

связи удерживается открытым на период монтирования. При выходе из строя клиента или сервера соединение разрывается, а другая сторона получает соответствующую информацию и предпринимает необходимые действия.

Реализация RFS на основе STREAMS (см. главу 17) и использовании интерфейса TPI (transport provider interface) корпорации AT&T позволило файловой системе быть независимой от конкретных архитектур сетей. Система может вести взаимодействие через множество потоков, что позволяет ей использовать различные транспортные средства на одной машине. Принцип соединения между клиентом и сервером показан на рис. 10.7.

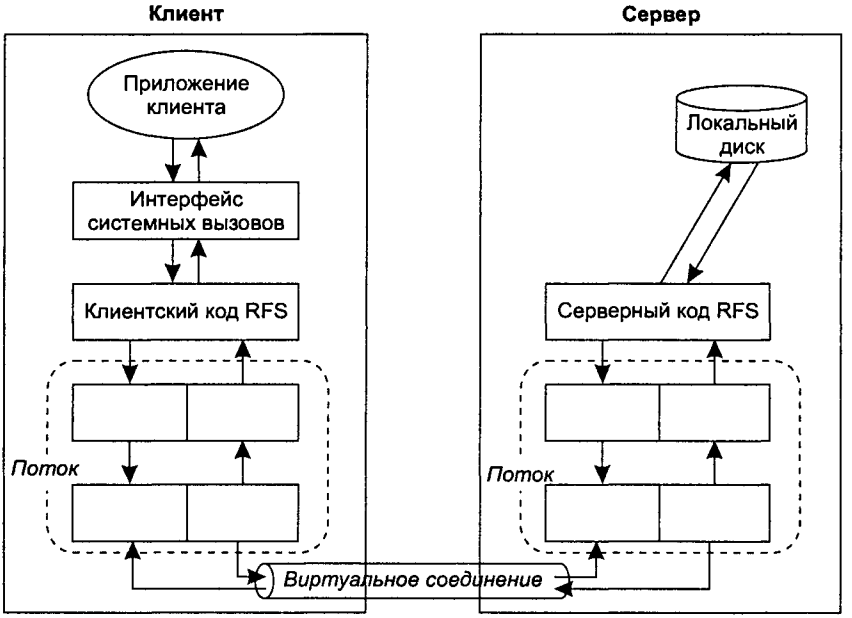


Рис. 10.7. Коммуникации в RFS

В системе RFS с каждым экспортируемым каталогом любого сервера ассоциируется символическое *имя ресурса*. Централизованный *сервер имен* (name server) производит преобразование имен ресурсов соответственно их размещению в сети. Это позволяет использовать ресурсы (экспортируемые файловые деревья) в структуре сети так, что клиенты имеют доступ к таким ресурсам, не зная их текущего месторасположения[1].

Так как система RFS разрабатывалась для применения в крупных компьютерных сетях, в ней предусмотрено комплексное управление ресурсами. В системе представлена концепция *домена*, являющегося логическим объединением набора машин в сети. Ресурсы отличаются друг от друга по именам доменов и именами самих ресурсов. Имена должны быть уникальны только в рамках

[1] Конечно, ресурс нельзя перенести в сеть, пока он монтируется клиентом.

одного домена. Если при запросе не указывается домен, предполагается использование текущего домена. Сервер имен может хранить информацию о ресурсах только своего домена и пересылать остальные запросы серверам соответствующих доменов.

10.12.1. Протокол удаленных сообщений

Изначальная версия системы RFS использовала модель удаленных системных вызовов, в которой предлагались операции RFS для каждого системного вызова по отношению к удаленным файлам. Для любой операции клиент складывал входные аргументы системного вызова и информацию о среде клиентского процесса в запрос RFS. Сервер заново создавал клиентскую среду и выполнял системный вызов. При этом процесс клиента блокировался до тех пор, пока обработка запроса не завершалась и не было получено ответное сообщение, содержащее результаты работы системного вызова. Затем клиент производил интерпретацию результатов и возвращал управление прикладной программе. Описанная реализация получила название протокола RFS1.0.

Позже система RFS была объединена с интерфейсом vnode/vfs, что потребовало для каждой операции использования vnode. При переносе системы на SunOS [4] каждая такая операция была реализована как один или несколько запросов RFS1.0. Например, команда `vn_open` может просто использовать запрос `RFS_OPEN`, в то время как для команды `vn_setattr` потребуется не только `RFS_OPEN`, но и один или несколько запросов `RFS_CHMODE`, `RFS_CHOWN` и `RFS_UTIME`.

Вместе с операционной системой SVR4 была представлена новая версия протокола RFS под названием RFS2.0. В нее был включен набор запросов, напрямую указывающих на vnode и vfs, таким образом, RFS2.0 стала полностью интегрированной с интерфейсом vnode/vfs. Однако это привело к определенным проблемам обратной совместимости, так как машины в сети могут работать под управлением разных реализаций UNIX и, следовательно, поддерживать разные версии протокола RFS.

Для решения описанной проблемы клиентов и серверов SVR4 заставили понимать оба варианта протокола. После соединения (при первой операции `mount`) машины обмениваются друг с другом информацией о поддерживаемых протоколах и договариваются о той версии, которую понимает и клиент и сервер. Следовательно, RFS2.0 будет использоваться только в том случае, если протокол поддерживают обе машины. Если на одной из них установлена система SVR3, а на другой — SVR4, то такие машины будут обмениваться данными при помощи протокола RFS1.0.

Данный подход требует реализации в SVR4 двух версий операций для каждого vnode и vfs, одной для коммуникаций с узлами под управлением SVR4 и другой для более старых систем.

10.12.2. Операции сохранения состояния

Файловая система полностью поддерживает информацию о состояниях, что означает хранение сервером данных о клиентах. В частности, сервер ведет протоколирование информации о файлах, открытых клиентами, и увеличивает счетчики ссылок их объектов vnode. Кроме того, сервер отслеживает факт блокировки файла или записи, установленной каждым клиентом, а также счетчики записей для именованных каналов. Он поддерживает таблицу всех клиентов, монтировавших файловые системы. В этой таблице содержатся сетевые адреса клиентов, имена монтируемых файловых систем и параметры виртуальных соединений.

Необходимость сохранения состояний принуждает сервер и клиента информировать друг друга о произошедшей аварии с целью проведения процедур восстановления. Более подробно об этом читайте в разделе 10.13.3.

10.13. Реализация системы RFS

Протокол RFS позволяет четко разграничить функции клиента и сервера. Операции mount обрабатываются отдельно при помощи средства *удаленного монтирования* во взаимодействии с сервером имен. Остановимся подробнее на каждом компоненте системы по отдельности.

10.13.1. Удаленное монтирование

Сервер RFS может представлять («рекламировать») свои каталоги при помощи системного вызова advfs. Входными аргументами этой функции являются имя экспортируемого каталога, ассоциируемое с ним имя ресурса и список клиентских машин, уполномоченных на получение доступа к ресурсу. Кроме этого при создании виртуального соединения сервер может требовать проверки пароля.

Системный вызов advfs работает следующим образом. Сервер вызывает его с целью представления каталога. Затем advfs создает в списке ресурсов ядра новое вхождение для этого каталога (рис. 10.8). Вхождение содержит имя ресурса, указатель на vnode экспортируемого каталога и список авторизированных клиентов. Оно также может содержать заголовок списка монтированных каталогов каждого клиента, использующего данный ресурс. В системе SVR4 вызов advfs заменен на rfsys, осуществляющий экспорт нескольких подфункций, одной из которых является представление файловой системы.

На рис. 10.9 показано взаимодействие между сервером RFS, сервером имен и клиентом. Для регистрации ресурса на сервере имен сервер RFS вызывает adv(1). После этого клиент может смонтировать ресурс RFS при помощи

```
mount -d <RNAME> /mnt
```

где <RNAME> — имя ресурса. Команда mount производит запрос к серверу имен в целью получения информации о местонахождении ресурса в сети и настройке виртуального соединения (если это необходимо). Затем она вызывает системную функцию mount с полным именем локальной точки монтирования и флагом, указывающим на тип подключения RFS, в качестве входных аргументов. Специфические для RFS параметры системного вызова включают указатель виртуального соединения и имя ресурса.

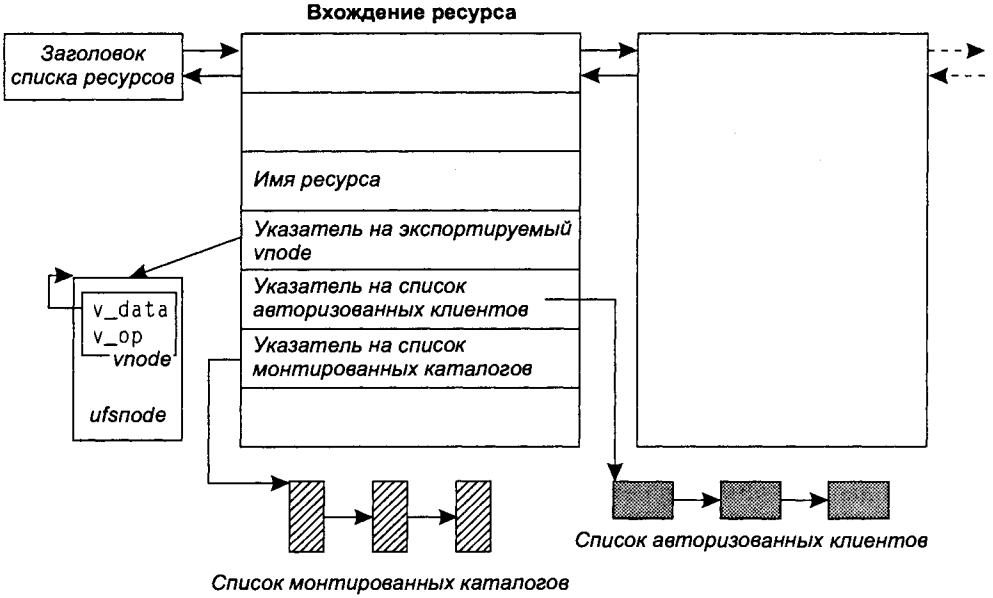


Рис. 10.8. Список ресурсов RFS

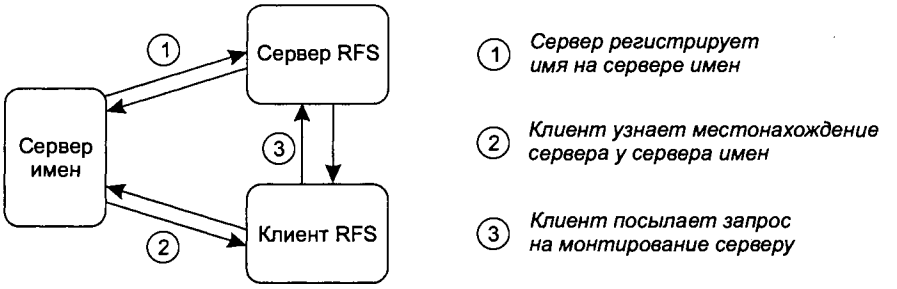


Рис. 10.9. Монтирование файловой системы RFS

Функция mount посылает серверу запрос MOUNT, передавая в нем символическое имя ресурса. Сервер находит соответствующий ему элемент в таблице ресурсов, проверяет права клиента на монтирование ресурса и добавляет в элемент списка монтирования запись о новом клиенте. Затем он увеличивает на единицу счетчик ссылок экспортируемого каталога и отправляет сооб-

щение об удачном завершении операции клиенту. Ответное сообщение содержит *идентификатор монтирования* (mount ID), который будет использоваться клиентом при последующих запросах. Идентификатор позволяет серверу быстро находить необходимый ресурс.

После получения ответа на запрос MOUNT происходит завершение клиентской части обработки. Клиент устанавливает вхождение vfs и сохраняет идентификатор монтирования в специфической для файловой системы структуре данных. Он также устанавливает для корневого каталога ресурса объект vnode. Поле v_data объекта vnode системы RFS указывает на структуру данных под названием *дескриптор отправлений* (send descriptor), содержащий информацию о виртуальном соединении (например, указатель на поток), а также на дескриптор файла, используемый сервером для обнаружения соответствующего локального объекта vnode.

При первой операции монтирования между клиентом и сервером образуется виртуальное соединение. Все последующие монтирования (а также другие операции RFS) будут разделять то же виртуальное соединение, которое будет поддерживаться до тех пор, пока не будет размонтирован последний ресурс. Операция mount инициализирует соединение к процессу-демону на сервере при помощи транспортного интерфейса. После установления соединения клиент и сервер договариваются о некоторых параметрах его работы, в том числе о номере версии используемого протокола и типе аппаратной архитектуры. Если две машины имеют различную архитектуру, то для кодирования данных они будут использовать метод XDR.

Установка виртуального соединения происходит в пользовательском режиме. Для этого задействуется стандартный сетевой программный интерфейс. После завершения установки пользователь вызывает функцию FWRD системного вызова rfsys для передачи созданного соединения под управление ядра.

10.13.2. Серверы и клиенты RFS

Клиент может обращаться к файлам RFS при помощи имени или файлового дескриптора. При просмотре пути ядро может достичь точки монтирования системы RFS. В реализации первой версии протокола остальная часть полного имени будет отослана серверу, который произведет дальнейший поиск одной операцией и возвратит дескриптор файла. В RFS2.0 клиент может монтировать в каталогах RFS и другие файловые системы. Система всегда проверяет каталоги на наличие такого подключения. Если она обнаружит его, клиент будет преобразовывать полное имя по одному компоненту за один проход, что позволит корректно обработать встретившуюся при этом точку монтирования. Если к каталогам RFS не присоединены какие-либо файловые системы, клиент отправит серверу полное имя целиком. В ответ сервер возвратит дескриптор, сохраняемый клиентом в закрытой структуре объекта vnode. Структуры данных клиента и сервера представлены на рис. 10.10.

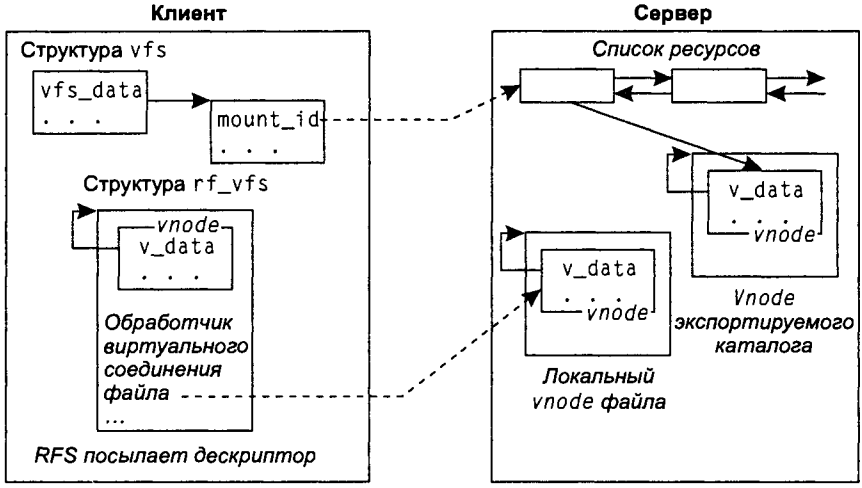


Рис. 10.10. Структуры данных RFS

Все последующие операции с файлами осуществляются через дескрипторы, используемые ядром для обнаружения vnode. Операции, указанные в vnode, вызывают клиентские функции RFS, которые извлекают из объекта vnode дескриптор файла и передают его в запросе RFS, отправляемом серверу. Дескриптор полностью прозрачен для клиента и обычно содержит обыкновенный указатель на объект vnode, расположенный на сервере.

На сервере RFS запущен один или несколько постоянно выполняющихся демонов. Такие процессы работают полностью в ядре. Это сделано с целью избавления от необходимости переключения контекста между пользовательским и привилегированным режимами ядра. Каждый демон «прослушивает» систему на предмет появления входящего запроса, после полной обработки которого принимается за следующий. При обслуживании запроса демон идентифицирует клиентский процесс при помощи удостоверений, ограничений на ресурсы, а также других атрибутов, находящихся в сообщении. При необходимости ожидания освобождения ресурсов демоны RFS могут переходить в режим сна. Планирование выполнения демонов RFS осуществляется в том же порядке, что и для обычных процессов системы.

10.13.3. Восстановление после отказа

В системах, поддерживающих сохранение состояния, необходимы определенные механизмы восстановления после выхода из строя. Фатальный сбой и другие ошибки должны уметь обнаружить и обработать не только серверы, но и клиенты. При отказе любой машины в связке «клиент-сервер» происходит разрыв виртуального соединения. Транспортный механизм предупреждает систему RFS о возникновении такой ситуации. Виртуальное соединение

может быть разорвано и в результате сбоя в сети, однако RFS способна интерпретировать его только как произошедший крах системы. Механизмы восстановления клиента и сервера отличны друг от друга.

При потере клиентом работоспособности серверу необходимо *восстановить* все данные о состоянии, относящиеся к этому клиенту. С этой целью сервер поддерживает информацию о состоянии всех индексных дескрипторов для каждого клиента. Эти данные используются для проведения следующих мероприятий:

- уменьшение (декремент) счетчиков ссылок индексных дескрипторов на количество ссылок, удерживаемых ранее клиентом, претерпевшим отказ в работе;
- освобождение все блоков отправителей-получателей в именованных каналах, используемых клиентом;
- освобождение всех блоков записей и файлов, удерживаемых клиентом.

В случае аварии сервера все клиентские процессы, ожидающие ответного сообщения от него, будут разбужены, системные вызовы завершат работу с ошибкой ENOLINK. Все индексные дескрипторы RFS, указывающие на файлы вышедшего из строя сервера, получат флаг, заставляющий все последующие операции над такими файлами завершать с ошибкой. Для выполнения дополнительных задач, связанных с восстановлением, будет разбужен демон прикладного уровня rfudaemon (1M).

10.13.4. Другие части системы

Пул процессов (process pool). Система RFS предоставляет прозрачный доступ к удаленным устройствам и именованным каналам, поэтому серверные процессы могут быть заблокированы на продолжительный период времени в ожидании завершения ввода-вывода устройства или канала[1]. Таким образом, серверу необходимо поддерживать динамическое резервирование процессов. Если при поступлении клиентского запроса все серверные процессы окажутся занятыми, на сервере будет создан новый процесс. Пул имеет (настраиваемую) верхнюю границу, последний свободный процесс не может быть переведен в режим сна.

Удаленные сигналы (remote signals). UNIX позволяет прервать выполнение системного вызова по сигналу. Это средство ОС можно расширить и до сред RFS. Представьте ситуацию, когда клиентский процесс блокируется в ожидании ответа сервера на запрос. Если такой процесс получит сигнал, ядро системы пошлет серверу запрос signal. Такой запрос идентифицирует

[1] На практике применение RFS для разделения ресурсов является весьма проблематичным за исключением случаев полностью однородных систем. Даже небольшие различия в семантике системного вызова могут сделать невозможным совместное использование ресурсов.

процесс при помощи *системного идентификатора* (system ID) совместно с идентификатором PID. Сервер использует полученную информацию для обнаружения спящего процесса-демона и отправляет ему сигнал.

Передача данных (data transfer). При использовании в локальных системных вызовах read и write процедуры copyin() и copyout() передают данные между буферным кэшем и пользовательским адресным пространством. Операции RFS должны направлять данные в одну или другую сторону через сообщения с запросами. При этом для указания процесса, являющегося сервером RFS, устанавливается специальный флаг в элементе таблицы процесса. Процедуры copyin() и copyout() проверяют этот флаг и корректно обрабатывают передачу данных.

10.14. Кэширование на стороне клиента

Если бы каждая операция ввода-вывода требовала доступа к удаленной системе, производительность системы RFS была бы невелика. Естественным решением проблемы является некая форма кэширования данных. Однако для распределенных подсистем кэша существуют определенные трудности в поддержке корректности данных, находящихся во множестве клиентских кэшей и одном на сервере. В NFSv2 имеется весьма простое решение проблемы, подходящее для систем, не сохраняющих состояния и не требующих точного соблюдения семантики UNIX. Однако для RFS необходимо более продуманное средство наряду с поддержкой работоспособного протокола целостности данных.

Кэширование на стороне клиента RFS впервые появилось в операционной системе SVR3.1 [1]. Активация кэша происходит при монтировании системы. Пользователь имеет право отключить кэш при желании (это требуется для некоторых приложений, имеющих собственные средства кэширования). Кэш RFS разрабатывался с целью строгого соблюдения целостности данных, что означает невозможность возникновения ситуаций получения из него устаревшей информации.

Кэш является полностью сквозным. Клиенты отправляют запросы серверу сразу же после модификации локальной кэшируемой копии. Это не увеличивает производительность записи, но является важным элементом сохранения корректности. Системный вызов read возвращает кэшированные данные тогда и только тогда, когда запрошенные данные имеются в кэше (операция чтения может считывать данные из нескольких буферов). Если какая-либо часть данных в нем не обнаружена, клиент заберет все данные с сервера. Если читать с сервера только те блоки, которые отсутствуют в кэше, не гарантируется неделимость операции read, так как блоки из кэша и с сервера могут соответствовать различным состояниям файла.

Кэш RFS использует ресурсы локального дискового буферного кэша. При этом часть буферов резервируется для RFS, часть — для локальных файлов,

а остальные пригодны для обеих целей. Такой подход защищает локальные файлы от монополизации системой RFS всего набора буферов. Работа с буферами осуществляется в порядке *последнего, недавно использовавшегося* (LRU).

10.14.1. Достоверность кэша

Модель сохранения корректности гарантирует, что операция чтения всегда возвратит данные, идентичные образу файла на сервере на момент времени доступа. Образ файла на сервере соответствует дисковой копии файла, а также более старым блокам буферного кэша сервера.

Любое изменение файла, произведенное пользователем сервера или одним из его клиентов, делает все копии кэшированных данных устаревшими. Один из способов решения проблемы подразумевает оповещение всех клиентов, «пострадавших» от операции записи, однако он приводит к значительному увеличению сетевого трафика и является слишком затратным. В системе RFS корректность данных гарантируется более элегантным способом: путем разделения клиентов на удерживающих данный файл открытым и уже успевших его закрыть.

Протоколы сохранения целостности требуются только для файлов, используемых несколькими клиентами. Если удаленный файл разделяется между некоторым количеством процессов, выполняющихся на одной машине, за корректностью кэша следит клиент без привлечения к этому сервера.

На рис. 10.11 показана схема поддержки непротиворечивости кэша. После получения сервером первого запроса на запись по отношению к файлу, открытому одним или несколькими клиентами, операция write временно приостанавливается, а остальным клиентам, открывшим файл, отправляется сообщение invalidate («объявить недействительными»). Клиенты помечают все данные кэша о таком файле как устаревшие и временно запрещают его кэширование. Последующие операции чтения файла вместо кэша будут запрашивать данные прямо с сервера. Кэширование разрешается после закрытия файла процессом, осуществлявшим запись, либо через определенный (настраиваемый) промежуток времени, прошедший после последнего изменения файла. Обработка запроса на запись будет продолжена после того, как все клиенты пометят данные кэша как недействительные и вышлют подтверждения.

Периодически возникает ситуация, когда некоторые из клиентов успевают закрыть файл, но часть его блоков все еще хранится в кэше. Важно защититься от использования устаревших данных кэша при повторном открытии файла. Такая возможность реализована посредством ассоциирования с каждым файлом *номера версии*, который инкрементируется всякий раз при внесении в файл изменений. Сервер возвращает номер версии в каждом ответном сообщении на вызов open. Клиент сохраняет это число в кэше. Если файл будет модифицирован после того, как клиент уже закрыл его, в ответ

на попытку его повторного открытия сервер пошлет несовпадающий номер версии. При получении другого номера версии клиент может смело очистить все блоки, ассоциируемые с файлом, что станет гарантией невозможности получения устаревших данных.

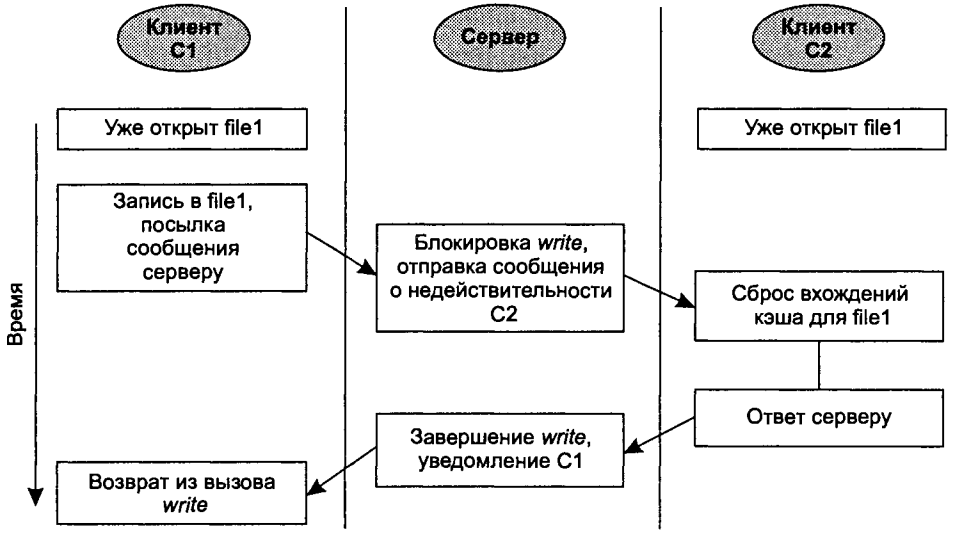


Рис. 10.11. Алгоритм подтверждения корректности системы RFS

В нормальных условиях функционирования описываемые механизмы RFS гарантируют целостность системы при достаточно приемлемых затратах. Проблема может возникнуть в случае выхода из строя или отсутствия реакции клиента. Это чревато длительной запаздыванием ответного сообщения на предмет недействительности данных в кэше, что задерживает выполнение операций на остальных узлах. В данном случае единичная ошибка всего лишь одного клиента может стать причиной возникновения сложностей во всей сети. С другой стороны, преимущества кэширования заключаются не только в поддержке целостности системы, но и во влиянии на ее производительность. Тесты показали увеличение производительности работы RFS более чем в два раза по сравнению с предыдущими реализациями системы (измерения проводились при подключении от одного до пяти клиентов).

10.15. Andrew File System

Файловые системы NFS и RFS предназначены для использования в небольших локальных сетях и с ограниченным числом клиентов. Обе системы не подходят для типичных университетских сетей, объединяющих несколько зданий и сотни, а иногда и тысячи клиентских машин. В 1982 году универси-

тет Карнеги—Меллона и корпорация IBM создали для развития вычислительной инфраструктуры учебных сетей Information Technology Center (ITC). Одним из наиболее важных проектов центра стала разработка распределенной файловой системы Andrew File System (AFS), обладающей возможностью обслуживать тысячи пользователей [19].

Объединение ITC реализовало несколько редакций системы AFS, кульминацией которой стала версия 3.0. Позже работа над AFS стала осуществляться компанией Transarc Corporation, созданной группой разработчиков систем. Здесь AFS была преобразована в компонент распределенной среды вычисления OSF (Distributed Computing Environment, DCE) под названием *распределенной файловой системы* (Distributed File System, DFS). Следующие разделы будут посвящены описанию AFS. О файловой системе DCE DFS будет рассказано в разделе 10.18.

Разработчики AFS стремились добиться не только хорошей масштабируемости системы, но и некоторых других целей [29]. Система должна быть совместима с UNIX, не требуя изменения кодов для работы на клиентах AFS. Система также должна предоставлять универсальное, независимое от фактического месторасположения разделяемых файлов пространство имен. Пользователям системы AFS нужно иметь возможность доступа к своим файлам с любой клиентской машины в сети. Перемещение файлов не должно дополнительно загружать систему. Необходимо предусмотреть устойчивость к ошибкам, так, чтобы сбой сервера или некоего компонента сети не смог привести к невозможности использования системы. Ошибки следует изолировать как можно ближе к точке сбоя. Система также должна обладать средствами защиты, не доверяющими слепо клиентским рабочим станциям или сети. И последним требованием при разработке системы являлась производительность работы, не меньшая, чем у систем разделения времени.

10.15.1. Масштабируемая архитектура

При создании масштабируемой распределенной файловой системы большое значение приобретают три наиболее важные проблемы. Если обслуживание клиентских запросов (поступающих в большом количестве) передать одному серверу, результатом станет не только перегрузка самого сервера, но и сети в целом. Неверно настроенное кэширование на стороне клиента может привести к интенсивному сетевому трафику. И, наконец, если загрузить сервер обработкой всех операций сразу, это приведет через некоторый период времени к его перегрузке. В масштабируемых системах необходимо уделить внимание корректному решению всех трех перечисленных проблем.

В системе AFS контроль перегрузки сети и сервера осуществляется путем разделения сети на некоторое количество независимых кластеров. В отличие от NFS или RFS, в файловой системе AFS используются отдельные специа-

лизированные серверы. Каждая машина может быть либо клиентом, либо сервером, но никогда не выполняет сразу обе функции. Организация сети AFS представлена на рис. 10.12. Каждый кластер содержит определенное число клиентов, а также сервер, хранящий файлы этих клиентов, такие как каталоги владельцев клиентских рабочих станций.

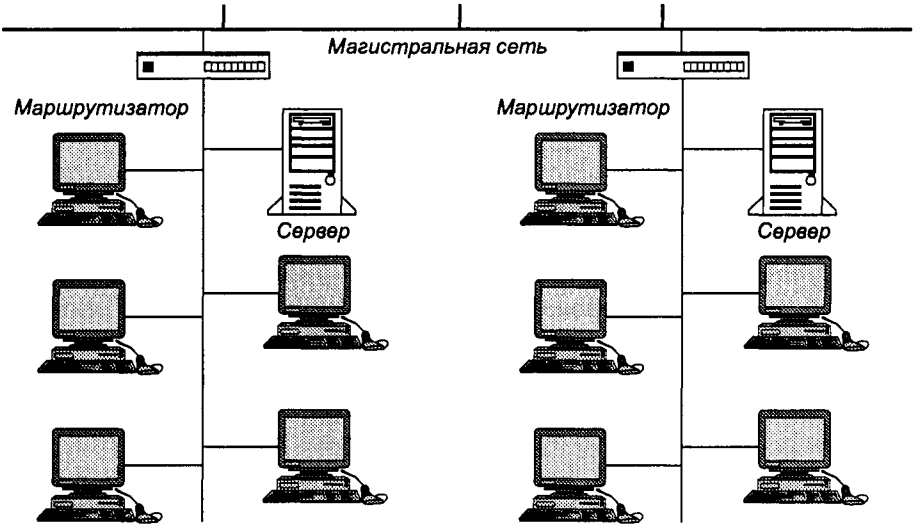


Рис. 10.12. Файловая система AFS: организация сети

Такая архитектура дает возможность наиболее быстрого доступа к файлам, расположенным на сервере, относящемся к одному сегменту сети. Пользователи имеют право доступа к файлам любого сервера, однако при этом производительность операций будет меньше. Сеть можно перенастраивать динамически с целью балансировки загруженности серверов и сегментов сети.

Наряду с протоколом, поддерживающим сохранение состояний для минимизации сетевого трафика, в системе AFS обязательно применяется кэширование файлов. Клиенты кэшируют недавно запрошенные файлы на локальных дисках. В оригинальной реализации AFS файл кэшировался целиком. В AFS3.0 файл разделяется на 64-килобайтовые порции. Каждая такая порция кэшируется отдельно. В управлении кэшированием на клиентских машинах активное участие принимают серверы. Они уведомляют своих клиентов о недействительности данных кэша. Более подробно об используемом протоколе будет рассказано в разделе 10.16.1.

Уменьшение загруженности сервера AFS проистекает также за счет перекладывания бремени просмотра сетевых имен с сервера на клиентов. Клиенты кэшируют каталоги полностью и самостоятельно производят поиск имен файлов. Более подробно об этом читайте в разделе 10.16.2.

10.15.2. Организация хранения и пространство имен

Все совместно используемые файлы хранятся на наборе серверов AFS, названных *Vice* (сокращение от аббревиатуры Vast Integrated Computing Environment, что означает «большая интегрированная среда вычислений»). В системе AFS файлы хранятся в логических единицах, называемых *томами*. Том — это коллекция связанных между собой файлов и каталогов, формирующих поддерево в иерархии разделяемых файлов [30]. Например, том может содержать все файлы одного пользователя. В каждом разделе диска может храниться несколько небольших томов. Однако большие тома способны объединять сразу несколько дисков.

Том является единицей хранения, отличающейся от понятия раздела, которое относится к физическому размещению данных на дисках. Такое разделение сущностей обладает несколькими преимуществами. Тома можно свободно перемещать из одного месторасположения в другое, не влияя на работу активных пользователей. Эти действия могут производиться с целью балансировки загруженности системы или при передислокации пользователей. Если рабочая станция пользователя физически перемещается в другой сегмент сети, администратор также может перенести пользовательский том на сервер, являющийся для нового сегмента локальным. Применение томов позволяет создавать файлы, имеющие больший объем, чем один диск. Тома в режиме «только для чтения» могут быть подвергнуты репликации (тиражированию) на несколько серверов сразу, что способствует увеличению их доступности и производительности работы с ними. И, наконец, существует возможность создания резервной копии и восстановления каждого тома по отдельности.

Система AFS предоставляет универсальное пространство имен, независимое от физического размещения файлов. Каждый файл идентифицируется при помощи fid, состоящего из *идентификатора тома* (volume ID), *номера vnode* (vnode number) и *унификатора vnode* (uniquifier vnode). Исторически сложилось так, что в системе AFS термин vnode используется для обозначения *индексного дескриптора Vice* (Vice inode). Следовательно, номер vnode является индексом к списку индексных дескрипторов тома. Унификатор – это генерируемый номер, увеличивающийся на единицу каждый раз при повторном использовании vnode.

Независимость местонахождения и прозрачность достигается путем применения базы данных размещения томов. База осуществляет связь между идентификатором тома и его физическим расположением. Копия базы данных реплицируется на каждом сервере, поэтому не является ресурсом, приводящим к возникновению узких мест в системе. Если том перемещается, оригинальный сервер хранит информацию об этом, что дает возможность не обновлять базы данных остальных серверов немедленно. При перемещении

тома оригинальный сервер может произвести необходимые модификации данных, которые позже будут перенесены на новый сервер.

Каждая рабочая станция должна быть оборудована локальным диском. На диске содержится небольшой набор локальных файлов, а также каталог, на который производится монтирование совместно используемой иерархии. По соглашению каждая рабочая станция монтирует дерево файлов в один и тот же каталог. Локальными файлами обычно являются системные файлы, необходимые для осуществления минимума операций, а также файлы, хранимые пользователем на локальном диске из соображений безопасности или производительности работы. Из сказанного следует, что клиенты видят одинаковое совместно используемое пространство имен плюс некоторое количество собственных уникальных файлов, расположенных локально. Локальные диски играют роль кэша недавно использованных разделяемых файлов.

10.15.3. Семантика сеансов

В централизованных системах UNIX при модификации файла одним процессом все остальные процессы могут увидеть изменения сразу при последующем вызове read. Перенесение семантики UNIX на распределенные файловые системы приведет к увеличению сетевого трафика и уменьшению производительности. В системе AFS 2.0 используется протокол под названием *семантики сеансов*, где корректность кэша обеспечивается на уровне операций open и close. Клиенты сбрасывают изменения данных на сервер только при закрытии файла. После этого сервер уведомляет клиентов, обладающих копиями файла в кэше, о недействительности их копий. Клиенты не проверяют корректность данных при каждом чтении или записи файла и продолжают использовать устаревшую информацию до тех пор, пока не откроют файл заново. Таким образом, пользователи на разных машинах видят изменения совместно используемого файла только при его открытии-закрытии в отличие от систем UNIX, в которых для этого достаточно вызвать read или write.

В системе AFS гарантируется действительность метаданных, которые обновляются при передаче на сервер (или от сервера клиентам) немедленно. Например, если на одной из клиентских машин завершена обработка системного вызова rename, ни одна другая машина в сети больше не сможет открыть файл под старым именем.

Гарантии корректности при использовании семантики сеансов слабее по сравнению с семантикой UNIX. В AFS 3.0 действительность данных проверяется при каждом чтении или записи, что дает большую степень гарантии их корректности. Такой подход лишь немного приближает систему к семантике UNIX, так как клиенты по-прежнему сбрасывают изменения файла на сервер только при его закрытии. В файловой системе DFS, потомке AFS, семантика UNIX поддержана при помощи механизма передачи маркеров доступа, который будет описан в разделе 10.18.2.

10.16. Реализация AFS

Файловая система разделяет серверы и клиенты. Коллекция серверов именуется Vice, клиентские станции получили название *Virtue*. На серверах и клиентах в качестве операционной системы установлен один из вариантов UNIX. Первые реализации AFS использовали для реализации клиентских и серверных средств простейшие процессы прикладного уровня. Ядро клиента было модифицировано с целью обнаружения ссылок на разделяемые файлы и переноса их клиентским процессам AFS, известным как *Venus*. В AFS 3.0 ядро клиента стало содержать менеджер кэша AFS, осуществляющий операции AFS через интерфейс vnode/vfs. Сервер работает как один многонитевый прикладной процесс (использующий нитевые библиотеки прикладного уровня традиционных систем). В более поздних реализациях большинство средств сервера было встроено в ядро, а сам сервер выполнялся в контексте процессов-демонов.

10.16.1. Кэширование и достоверность данных

Менеджер кэша [31] реализует операции интерфейса vnode для файлов AFS на клиентской машине. При первом открытии файла AFS клиентом менеджер кэша производит его чтение целиком (или 64-килобайтовыми порциями для файлов объемом больше 64 Кбайт) и кэширует его как файл в клиентской файловой системе. Менеджер передает все вызовы read или write кэшированной копии. При закрытии файла менеджер сбрасывает изменения на сервер.

При получении файла менеджером кэша сервер также организует *обратную связь* (callback), ассоциированную с данными. Обратная связь используется как гарантия действительности данных. Если файл изменит другой клиент и запишет эти изменения на сервер, последний проинформирует об этом всех остальных клиентов, обладающих обратной связью по отношению к данному файлу. Такое действие называется *прерыванием обратной связи*. В ответ клиент очищает устаревшие данные и переносит с сервера обновления, если это необходимо[1].

Клиенты кэшируют атрибуты файлов отдельно от их данных. В отличие от содержимого файлов, сбрасываемого на диск, атрибуты кэшируются в памяти. Клиенты и серверы применяют для атрибутов одинаковые механизмы обратной связи. Клиент уведомляет сервер при каждом изменении файловых атрибутов, после чего сервер разрывает все обратные связи, ассоциированные с ними.

Применение механизма разрыва обратных связей может привести к возникновению состязательности [14]. Представьте ситуацию, при которой кли-

[1] До появления AFS 3.0 клиенты удаляли устаревшие данные только при следующем открытии файла. Такой подход (а также большой размер порций) сделал предыдущие реализации системы неприменимыми для систем управления базами данных и обработки транзакций.

ент получает файл с сервера сразу после отправки сообщения о разрыве обратной связи с этим файлом. Если данные будут получены клиентом позже уведомления, то он не сможет распознать, к каким именно данным оно относится, текущим или более старым. В любом случае клиент не будет знать, является ли полученное уведомление действительным. В системе AFS возникающая проблема решена при помощи сброса текущих данных и повторного получения их с сервера. Такой подход может привести к увеличению сетевого трафика и уменьшению скорости операций. Однако подобные ситуации складываются при редком стечении обстоятельств, и на практике происходят не часто.

Еще одной проблемой, которой стоит уделить особое внимание, является невозможность доставки уведомлений о разрыве связей при возникновении временных сбоев сети. В системе AFS клиент может не обращаться к серверу в течение продолжительного периода времени. В течение этого промежутка клиент считает данные, хранящиеся в его кэше, корректными. По истечении установленного периода времени (по умолчанию — каждые 10 минут) клиент проверяет все серверы, с которыми была установлена обратная связь.

Механизм обратных связей дает серверу возможность отслеживать состояния. Сервер хранит информацию обо всех обратных связях, ассоциированных с каждым файлом. При внесении клиентом изменений в файл сервер должен разорвать связи, ассоциированные с этим файлом. Если том, хранящий информацию, становится неуправляемым, сервер вправе разорвать некоторые из существующих связей и исправить ошибки носителя. Информация о корректности кэшируемых файлов должна поддерживаться клиентом.

10.16.2. Просмотр имен

Масштабируемая система должна уметь предупреждать перегрузку сервера. Одним из способов решения проблемы является передача определенных операций от сервера клиентам. В частности, операция *просмотра имен* интенсивно использует ресурсы процессора. В системе AFS ее обработка полностью перенесена на клиентские машины.

Клиент кэширует символические ссылки и каталоги, а также вхождения базы данных размещения томов. Просмотр имен происходит по одному компоненту за один проход. Если каталог не находится в кэше, клиент получает его от сервера. Затем он производит поиск компонента в этом каталоге. Вхождения каталога связывают имя компонента с его fid, содержащим идентификатор тома, номер и унификатор vnode.

Если клиент обладает информацией о расположении тома, то для получения следующего компонента он соединяется с соответствующим сервером (если, конечно, компонент уже не находится в локальном кэше). Иначе клиент запрашивает базу данных расположения томов ближайшего сервера и кэширует полученный ответ. Клиент интерпретирует вхождения кэшированной

базы данных как *рекомендации* (hints). При изменении информации сервер отвергает запрос, клиенту необходимо запрашивать базу данных для определения правильного расположения тома. Иногда возникает ситуация, когда том уже перемещен, а ближайший сервер еще не получил информацию об этом. В таком случае клиент сначала попытается обратиться по адресу предыдущего расположения тома, при этом сервер скорее всего будет обладать данными о перемещении и отреагирует на запрос соответствующим образом (передаст запрос по месту текущего расположения тома).

10.16.3. Безопасность

В файловой системе AFS защита ограничена Vice (коллекцией серверов). Рабочие станции и сеть подразумеваются полностью не защищенными. При таком подходе по сети никогда не передаются нешифрованные пароли, так как в противоположном случае их будет легко перехватить любым компьютером, прослушивающем сеть.

В AFS применяется система аутентификации Kerberos, разработанная Massachusetts Institute of Technology [32]. Клиенты системы Kerberos идентифицируют себя ответом на зашифрованные вызовы от сервера вместо отправки паролей, известных серверу и клиенту. Клиент расшифровывает вызов, затем шифрует ответ при помощи того же ключа и отправляет кодированное сообщение серверу. Так как сервер каждый раз использует различные вызовы, клиент не может применять один и тот же ответ дважды.

В работе [10] описываются несколько «дыр», сопровождающих использование механизма Kerberos в системе AFS 3.0. Клиент хранит некоторые важные структуры данных в своем адресном пространстве в открытом виде, что делает их потенциально уязвимыми. Пользователи, имеющие привилегии доступа root на рабочую станцию, могут просмотреть структуры данных ядра и получить данные об идентификации Kerberos других пользователей. Более того, протокол ответа на вызовы, применяемый в AFS 3.0, подвержен атакам с других узлов сети, отправляющих поддельные вызовы клиентам. Компания Transarc убрала описанные «дыры» позже, выпустив AFS 3.1.

Файловая система AFS поддерживает списки контроля доступа (access control lists, ACL) для каталогов (но не для отдельных файлов). Каждый список ACL представляет собой массив пар. Первым элементом пары является имя пользователя или группы, второй элемент определяет права, установленные для этого пользователя или группы. Система AFS поддерживает четыре типа прав доступа к каталогу: lookup (просмотр), insert (добавление), delete (удаление) и administer (изменение списка ACL для данного каталога). Для файлов каталога предусмотрено три типа полномочий: read (чтение), write (запись) и lock (блокировка). Система AFS также поддерживает стандартные биты привилегий UNIX, для работы с файлом необходимо передать оба типа прав (ACL и UNIX).

10.17. Недостатки файловой системы AFS

Система AFS является очень хорошо масштабируемой архитектурой. В CMU система AFS была введена в строй в середине 1985 года и к январю 1989 года поддерживала 9 000 пользовательских учетных записей, 30 файловых серверов, 1 000 клиентских машин и примерно суммарный объем носителей 45 Гбайт. AFS подходит для разделения файлов между пользователями, находящихся на больших расстояниях друг от друга. К весне 1992 года существовало уже 67 кластеров AFS, доступных для публичного монтирования [7]. Файлы стали доступными в таких отдаленных друг от друга местах, как Институт исследований OSF в Гренобле (Франция), Университет Keio (Япония) и Национальный институт здоровья в Вашингтоне. Для работы с удаленными файлами применялись обычные команды UNIX, такие как cd, ls и cat. В [11] описывается серия проведенных тестов, подтверждающих уменьшение загрузки процессора, сетевого трафика и времени, необходимого для завершения удаленных операций, при применении файловой системы AFS.

Однако производительность работы клиентской части системы далека от совершенства [33]. Для кэширования недавно использованных кусков файлов в AFS используется локальная файловая система клиентской машины. При необходимости доступа к таким данным клиенту приходится выполнять дополнительные операции, отнимающие время. Кроме самого доступа к локальному файлу менеджеру кэширования необходимо проверить корректность кэшированной информации (поискать разорванные обратные связи) и создать связь файла AFS или его части с локальным файлом. Более того, если запрос потребует дробления на порции, менеджер кэша разобьет его на более мелкие, оперирующие с каждой порцией отдельно. В результате даже при наличии файла AFS в кэше доступ к нему требует в два раза больше времени, чем такая же операция по отношению к локальному файлу. При тщательной настройке система AFS работает по *наиболее быстрому пути* (если данные корректны и запрос относится только к одной порции), где время доступа может быть уменьшено на 10–15%.

Модель, сохраняющая состояния, трудна в реализации. Алгоритмы корректности кэша должны уметь предотвращать ситуации состязательности и потенциальные тупиковые ситуации. При реализации модель удаляется от семантики, принятой в UNIX. Действительность данных при этом гарантирована в меньшей степени, так как клиенты записывают изменения данных только при закрытии файла процессом. Такой подход может привести к возникновению неожиданных результатов. Клиент не всегда обладает возможностью сбросить изменения в файл вследствие отказа сервера, сбоев в сети или ошибок (например, переполнения диска). Проблема имеет два важных последствия. Во-первых, системный вызов close в AFS завершает работу

с ошибкой чаще, чем в других файловых системах. Многие приложения не проверяют значение, возвращаемое close, и не производят каких-либо действий по исправлению ошибки. В большинстве случаев приложение закрывает файл принудительно при завершении своей работы. Во-вторых, системный вызов write часто завершается успешно даже в том случае, если реально это не так (например, операция записи должна увеличить размер файла, но для этого нет места вследствие переполнения диска). Обе ситуации ведут к возникновению у клиента неожиданных последствий.

Перенос операции просмотра имен на клиентские машины уменьшает загрузку сервера, но требует от клиента понимания формата каталогов, принятого на сервере. Это отличает систему от NFS, предлагающую информацию о каталогах в формате, не зависящем от конкретной аппаратной архитектуры или установленной операционной системы.

Некоторые из перечисленных недостатков были решены в файловой системе DFS. О ней мы расскажем подробнее в следующем разделе.

10.18. Распределенная файловая система DCE (DCE DFS)

Поддержка и развитие файловой системы AFS в 1989 году были переданы корпорации Transarc. С ее помощью Open Software Foundation приняла технологию AFS как основу распределенной файловой системы для DCE (Distributed Computing Environment). Для новой версии системы AFS чаще всего используется название DCE DFS или просто DFS. Мы также будем называть в этой главе систему кратко — DFS.

Файловая система DFS стала потомком AFS, обладая некоторыми сходными с нею свойствами. Изменения системы велись в следующих направлениях:

- любая машина должна иметь возможность выступать одновременно в роли сервера и клиента;
- более строгое соответствие принятой в UNIX семантике совместно используемых файлов, а также гарантий корректности данных;
- более высокая степень взаимодействия с другими файловыми системами.

Корпорация Transarc разработала в качестве локальной файловой системы серверов DFS файловую систему Episode [6], более подробно описанную в разделе 11.8. Система имеет высокую степень доступности, а также поддерживает логические тома (тома AFS в Episode называются *наборами файлов*, filesets) и совместимые со стандартами POSIX списки контроля доступа. В этой главе будут описаны распределенные компоненты системы DFS.

10.18.1. Архитектура DFS

Архитектура файловой системы DFS во многом сходна с ее предшественницей, AFS [15]. Система использует клиент-серверную модель, поддерживающую запоминание состояний, с активным сервером, инициирующим рассылку сообщений о недействительности кэша. В DFS принято кэшировать файлы полностью (или порциями по 64 Кбайт для файлов больших размеров) в файловой системе клиента. Для поддержки прозрачности имен применяется база данных расположения томов.

Система DFS расширила возможности AFS. В файловой системе DFS интерфейс vnode/vfs используется как на клиенте, так и на сервере, что позволяет взаимодействовать с другими файловыми системами и протоколами доступа. Как правило, к файловой системе DFS могут подключаться и локальные пользователи сервера. Клиенты и серверы взаимодействуют друг с другом через протокол DCE RPC, обладающий рядом полезных возможностей, таких как синхронные и асинхронные режимы, аутентификация Kerberos и поддержка операций в сетях большой протяженности и транспортных средств, ориентированных на соединение.

Архитектура DFS схематично показана на рис. 10.13. Клиент системы схож с клиентом AFS, отличаясь только подходом к обработке каталогов. В обеих системах клиенты кэшируют информацию о каталогах. Однако в DFS сервер может экспортировать файловые системы различных типов (преимущество отдается файловой системе Episode, созданной специально для DFS). Из этого следует, что клиент может не понимать формата каталогов, экспортируемых сервером. Клиенты DFS кэшируют результаты отдельных операций просмотра, а не каталоги целиком.

Устройство серверной части DFS значительно отличается от сервера AFS. В реализации AFS протокол доступа и сама файловая система представляют собой интегрированное средство. В системе DFS они разделены между собой и взаимодействуют при помощи интерфейса vnode/vfs. Такой подход позволяет экспортировать внутреннюю файловую систему сервера. Сервер DFS использует расширенный интерфейс vnode/vfs (называемый VFS+), включающий в себя дополнительные функции для томов и списков управления доступом. Файловая система Episode поддерживает все операции VFS+, предлагая тем самым все функциональные средства DFS. Другие локальные системы могут не обеспечивать всех расширенных возможностей и предоставлять только часть средств DFS.

Обслуживание запросов клиентов DFS осуществляется *экспортерами протоколов* (protocol exporter), которые хранят информацию о состоянии каждого клиента и уведомляют его о недействительности некоторых кэшированных данных. Корректность передачи информации между экспортером протокола и другими методами доступа к файлам (локальным доступом

и иными протоколами, поддерживаемыми сервером) контролируется *связующем уровне* интерфейса vfs. Об этом мы подробнее расскажем в г разделе 10.18.3.

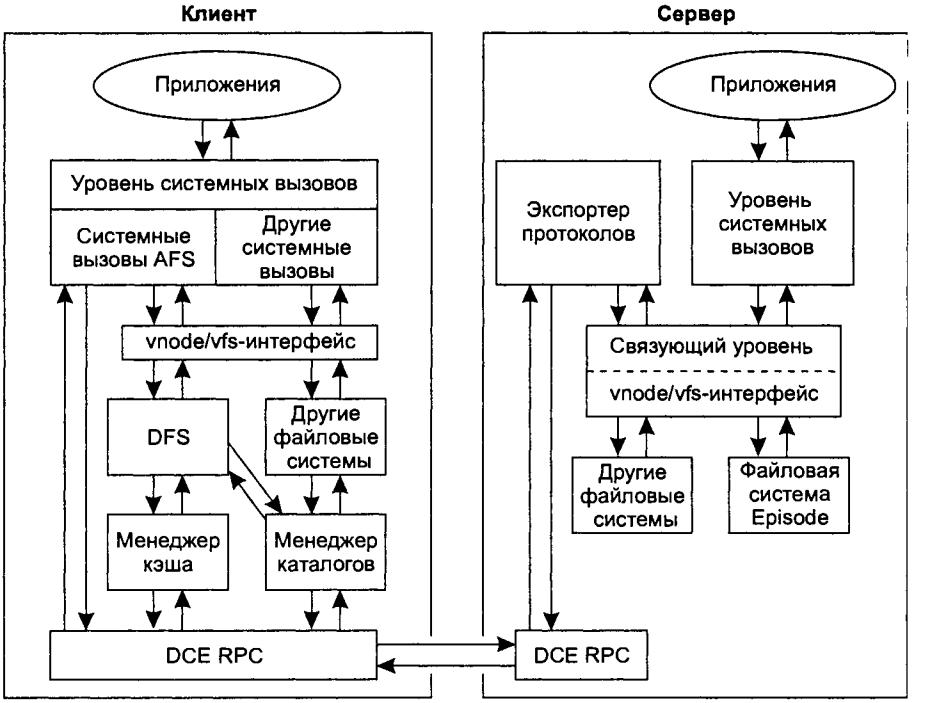


Рис. 10.13. Архитектура файловой системы DFS

10.18.2. Корректность кэша

Для доступа к совместно используемым файлам система DFS экспортир строгую единую семантику, принятую в UNIX. Если один клиент запи данные в файл, другой клиент, осуществляющий чтение, будет видеть обн ленную информацию. Система DFS гарантирует непротиворечивость дан кэша на уровне вызовов read и write (в отличие от AFS, поддерживающей новления только для вызовов open и close).

Сервер DFS реализует описанную семантику при помощи *менедж маркеров доступа* (token manager), отслеживающего ссылки на файлы в активных клиентов. По каждой ссылке сервер передает клиенту один и более *маркер доступа*, гарантирующий действительность данных или ат бутов файла. Сервер может в любое время отменить гарантию путем ан лирования маркера. После этого клиент должен считать соответствующ данные недействительными и запросить их с сервера повторно (если это обходимо).

Система DFS поддерживает четыре типа маркеров доступа, каждый из которых служит для различных наборов файловых операций:

Маркеры данных	Существуют два типа маркеров данных — чтения (read) и записи (write). Каждый из них применяется к байтовой области (записи) внутри файла. Кэшированная копия части файла считается действительной в том случае, если клиент удерживает маркер чтения для нее. При удержании маркера записи клиент может изменять кэшированные данные, не сбрасывая их на сервер. Если сервер аннулирует маркер чтения, клиенту необходимо удалить кэшированные данные. При отзыве сервером маркера записи клиенту следует передать все изменения файла на сервер и только затем удалить данные
Маркеры состояния	Эта группа маркеров дает гарантии корректности атрибутов кэшированных файлов. Маркеры состояния также бывают двух типов — чтения (status read) и записи (status write). Задачи маркеров состояния сходны с семантикой, описанной для маркеров данных. Если клиент удерживает маркер состояния записи, сервер заблокирует всех клиентов, которые попытаются прочесть атрибуты файла
Маркеры блокировки	Позволяют клиенту, обладающему ими, устанавливать различные типы блокировки какой-либо части файла. На период удержания маркера блокировки клиенту не нужно соединяться с сервером для блокировки файла, так как маркер дает гарантию непредоставления блока другому клиенту, способному спровоцировать конфликт
Маркеры открытия	Позволяет обладающему таким маркером открывать файл. Существует несколько типов маркеров открытия (соответствующих различным режимам открытия файла) — чтения (read), записи (write), выполнения (execute) и эксклюзивной записи (exclusive write). Например, клиент, обладающий маркером открытия для выполнения, может быть уверен в том, что никакой другой клиент не сможет изменить файл. Такую частичную гарантию весьма сложно поддержать другими распределенными файловыми системами. Однако она необходима, так как большинство систем UNIX осуществляют доступ к выполняемым файлам постранично (см. раздел 13.2). Если при выполнении файла произойдет его изменение, клиент получит часть старой программы и часть ее обновленной версии, что может привести к непредвиденным результатам

В системе DFS определены правила совместимости на тот случай, если два клиента захотят получить маркеры для одного и того же файла. Маркеры отличающихся типов являются взаимно совместимыми, так как относятся к различным компонентам файла. Для маркеров одного и того же типа правила зависят от конкретного типа маркера. Для маркеров данных и блокировки маркеры чтения и записи являются несовместимыми из-за перекрытия соответствующих им частей файлов. Маркеры состояния чтения или записи являются несовместимыми всегда. Среди маркеров открытия маркер эксклюзивной записи несовместим с остальными маркерами открытия, а маркеры выполнения, как правило, несовместимы с обычной записью. Для прочих комбинаций ограничений не имеется.

Маркеры похожи на ответные сообщения, принятые в файловой системе AFS: оба средства служат для гарантии корректности данных кэша, часть которого может быть объявлена недействительной сервером в любой момент времени. Но, в отличие от ответных сообщений, маркеры являются объектами определенного типа. В системе AFS принят один и тот же тип сообщений для данных и атрибутов файлов. Как это уже недавно упоминалось, система DFS поддерживает несколько типов маркеров. Последний вариант позволяет значительно уменьшить параллелизм файловой системы и предоставляет единообразную семантику UNIX для доступа к совместно используемым файлам.

10.18.3. Менеджер маркеров доступа

В каждом сервере DFS имеется менеджер маркеров доступа, постоянно находящийся на связующем уровне интерфейса vnode. Соединительный уровень содержит процедуру-оболочку для каждой операции vnode. Она запрашивает все маркеры, необходимые для завершения операции, и затем вызывает зависящую от файловой системы процедуру vnode.

В большинстве случаев каждая операция vnode требует сразу несколько маркеров доступа. Некоторые операции нуждаются в предварительном просмотре каталогов для идентификации набора объектов vnode, чьи маркеры должны быть получены для проведения действий. Например, операция rename требует маркеров состояния записи для каталога-источника и каталога-приемника, если они не одинаковы. Операции также необходимо произвести поиск в приемнике для проверки существования файла с именем, совпадающим с указанным как новое. В последнем случае операция переименования получает дополнительные маркеры доступа для удаления файла. Менеджер маркеров должен предупреждать возникновение клинча, применяя для этой цели механизмы, схожие с используемыми для физических файловых систем.

В системе часто возникает ситуация, при которой клиент запрашивает маркер, уже выданный другому клиенту. В таком случае менеджер блокирует запрос и аннулирует конфликтующие маркеры путем уведомления второго клиента (текущего владельца маркера). Если это маркер чтения, владелец просто возвратит его серверу и пометит данные в кэше как недействительные. Если это маркер записи, то перед его возвращением владелец сбросит все модифицированные данные на сервер. При конфликте маркеров открытия или блокировки владелец может не возвращать маркер до тех пор, пока не разблокирует или не закроет файл. После возвращения маркера менеджер завершает запрос и передает маркер ожидающему его клиенту.

Менеджер маркеров доступа должен функционировать на уровне vnode, так как система DFS может сосуществовать и с другими методами доступа, например локальным доступом через системные вызовы или другие распре-

деленные протоколы, такие как NFS. Менеджер запрашивает маркеры независимо от метода доступа, что позволяет предоставлять гарантии системы DFS во время смешанного доступа. Для примера представьте ситуацию, в которой два клиента одновременно осуществляют доступ к файлу, один при помощи DFS, а другой — используя NFS. Если при доступе через NFS не будет проведен запрос или проверка маркеров, будут нарушены гарантии корректности, установленные для клиента DFS. Перенеся менеджер на связующий уровень, сервер получил возможность синхронизации всех операций над файлом.

10.18.4. Другие службы DFS

Файловая система DFS требует не только экспортер протоколов и менеджера кэша на клиентской машине, но и другие средства. Система предлагает большое количество различных служб, работающих во взаимодействии с сервисными файловыми операциями. Перечислим некоторые из этих служб.

- **База данных размещения наборов файлов** (fldb). Глобальная реплицируемая база данных, содержащая данные о размещении каждого тома. Здесь хранится информация о местонахождении наборов файлов и их копий. Схожа с базой данных размещения томов системы AFS.
- **Сервер наборов файлов** (ftserver). Реализует операции над наборами файлов, например осуществление перемещения (миграции) набора.
- **Сервер аутентификации.** Служит для аутентификации на основе Kerberos.
- **Сервер репликации** (rpserver). Файловая система DFS поддерживает репликацию наборов файлов с целью увеличения доступности важной информации. Репликация помогает защититься от проблем, связанных с выходом из строя части сети или серверов, а также уменьшает количество узких мест, равномерно распределяя наиболее часто используемые наборы файлов между несколькими машинами. Несмотря на то, что наборы файлов могут быть доступны для чтения-записи, их копии открыты только для чтения. Система DFS поддерживает два типа репликации, свободную и планируемую. Для осуществления синхронизации копий с оригиналом при свободной репликации клиентам необходимо принудительно вызывать команды fts. При планируемой репликации изменение осуществляется автоматически через определенные промежутки времени.

10.18.5. Анализ

Система DFS предлагает исчерпывающий набор средств распределенного доступа к данным. Файловая система Episode использует для уменьшения

времени восстановления после сбоев ведение журналов, что значительно увеличивает доступность данных. Она использует разделение на две абстракции для организации файловой системы — агрегаты и наборы файлов. Агрегаты являются единицами физического хранения данных, в то время как наборы файлов представляют собой логические части файловой системы. С помощью этих средств в системе осуществляется разделение физического или логического размещения данных.

Файловая система Episode использует списки контроля доступа, совместимые со стандартом POSIX, что улучшает защищенность файлов. Списки контроля доступа с одной стороны позволяют применять более гибкую и надежную схему безопасности по сравнению с принятой в UNIX, однако они не привычны для системных администраторов и пользователей. Методика Kerberos дает возможность увеличения защищенности базовых средств DFS, но требует внесения изменений в некоторые программы, такие как login, ftp, а также в пакетные файлы и почтовые утилиты.

В системе DFS применяется репликация наборов файлов с целью увеличения доступности данных и уменьшения времени доступа путем распределения общей нагрузки между несколькими серверами. Она также позволяет проводить резервное копирование индивидуальных наборов файлов. Копия представляет собой «снимок» реплицируемого набора файлов. Независимость и прозрачность доступа реализована при помощи базы данных размещения наборов файлов.

Архитектура системы DFS основана на кэшировании на стороне клиента. Уведомление о недействительности данных в кэше инициализируется сервером. Такой подход идеален для крупных сетей, так как он уменьшает перегрузки в сети. Реализуя клиентскую и серверную часть на основе интерфейса vnode/vfs, система DFS имеет возможность взаимодействовать с другими физическими файловыми системами, а также протоколами локального и удаленного доступа к файлам.

Однако несмотря на большее количество преимуществ, в системе DFS имеются и некоторые недостатки. Архитектура DCE DFS весьма сложна для реализации. Она требует не только DCE RPC, но и набора других служб, таких как глобальная служба каталогов X.500 [22]. В частности, поддержка системы DFS на слабых машинах и простых операционных системах (например, в MD-DOS) может оказаться очень трудно реализуемой. Это является серьезным барьером для применения DFS в реально гетерогенных средах.

Механизмы корректности кэша и предупреждения тупиковых ситуаций, как правило, достаточно сложны. Алгоритмы должны также уметь восстанавливать работу после сбоев отдельных клиентов, серверов или сегментов сети. Эта проблема характерна для любой распределенной файловой системы, предлагающей семантику параллельного доступа к данным высокой детализации.

10.19. Заключение

В этой главе описывались архитектура и реализация четырех наиболее важных распределенных файловых систем — NFS, RFS, AFS и DFS. Система NFS является наиболее простой для реализации и характеризуется наиболее удобной для переноса архитектурой. NFS была портирована на многие аппаратные платформы и операционные системы, тем самым сделав протокол доступным для по-настоящему разнородных сред. Однако система NFS не обладает хорошей масштабируемостью, далека от семантики, принятой в UNIX, и имеет низкую производительность операции записи[1]. Система RFS предлагает совместное использование устройств и семантику UNIX, но хорошо работает только в System V UNIX и других ОС, основанных на ней. Файловые системы AFS и DFS основаны на весьма масштабируемых архитектурах.

---

[1] В 1998 г. Sun Microsystems представила спецификацию NFS в IETF для придания ей статуса стандарта. IETF взяла на себя обязательство создать новую версию протокола, в результате чего была создана рабочая группа, разработавшая предварительную версию NFSv4. Февраль 2000 г. — начало второй стадии. Sun сформулировала окончательные предложения и передала материалы в IETF для разработки окончательной эталонной реализации. Все документы этого периода и собственно спецификацию можно прочитать на родном сайте NFSv4 — http://www.nfsv4.org. Sun финансирует работы в Университете штата Мичиган, проводимые с целью создания образца и усовершенствования версии NFS для Linux. В настоящее время с NFSv4 связываются следующие надежды: легкость доступа к файлам Интернета и высокая производительность, сравнимые с работой в ЛВС за счет поддержки глобального пространства имен (распределенной по всему Интернету файловой системы), кроссплатформенно-совместимого, как реального противовеса пространству URL (WebNFS [RFC2054, RFC2055]). Основной инструмент NFSv4 — Transport Independent Remote Procedure Call, TI-RPC — удаленный вызов процедур, независимый от транспорта. Высокую производительность работы в Сети и в ЛВС должны обеспечить механизм агрессивного кэширования (делегирования прав) файлов на клиенте (сервер NFSv4, в отличие от версий 2 и 3, обращается к файлам, открытым пользователями, только при их изменении, а не поддерживает открытое соединение постоянно, он перекладывает ответственность за файл до момента закрытия последнего клиентом), сокращение количества соединений при выполнении сложных задач (поддержка составных операций — одновременная отправка на сервер сразу нескольких команд), динамическая реакция клиента на изменения, происходящие на серверах NFS, расширенная интернационализация имен, поддержка атрибутов файлов, улучшенная масштабируемость протокола, совместимость со всеми более ранними вариантами NFS. Естественно, работа в Интернете требует совершенно иного подхода к безопасности. Здесь строгая секретность и безопасность обеспечиваются переведенным из разряда поддерживаемых в основные средства API RPCSEC_GSS [RFC2203], позволяющим задействовать такие технологии, как аутентификация Kerberos v. 5 [RFC1510] или инфраструктура открытых ключей (Low Infrastructure Public Key, LIPKEY). Модель безопасности исключила поддержку SSL и предполагает сочетать алгоритмы делегирования и блокировки. Конечно, сложность задачи сравнима с переводом Интернета «на рельсы» IPv6, поэтому некоторое время еще придется подождать. Сегодня же с сайта университета г. Гуэльф можно загрузить только альтернативную экспериментальную среду NFSv4 — ftp://snowhite.cis.uoguelph.ca/pub/nfsv4, обладающую рядом ограничений. Это — отсутствие делегирования и аутентификации RPCSEC_GSS, клиента версий 4 (из-за частых его сбоев используются версии 2 и 3) и некоторые другие изменения. — *Прим. ред.*

Система DFS предлагает семантику UNIX, а также может взаимодействовать и с другими протоколами доступа. Однако она является весьма сложной для практической реализации. Технология DFS находится в стадии развития, и только время покажет, насколько она оказалась удачной.

На сегодняшний день существует несколько публикаций, посвященных измерению производительности описанных файловых систем. В [11] производится сравнение производительности систем NFS и AFS на идентичных аппаратных конфигурациях. Результаты измерений показали, что при использовании одного сервера и небольшой загрузке (менее 15 клиентов) система NFS работает быстрее, однако при увеличении загруженности эти показатели резко падают.

10.20. Упражнения

1. Почему для распределенных систем важна прозрачность сети?
2. Чем отличается прозрачность расположения от независимости расположения?
3. Назовите преимущества систем, не поддерживающих запоминание состояния. Каковы их недостатки?
4. Какие распределенные файловые системы предлагают для совместного доступа к файлам семантику UNIX? Какие из них поддерживают семантику сеансов?
5. Почему протокол mount отделен от протокола NFS?
6. Как работают асинхронные запросы PRC? Опишите действия клиентского интерфейса, отправляющего асинхронный запрос и получающего ответное сообщение.
7. Создайте программу PRC, позволяющую клиенту отправлять строку текста для распечатки на сервере. Опишите, для чего может быть нужна подобная служба.
8. Допустим, что сервер NFS претерпел аварию и был перезагружен. Каким образом сервер теперь узнает информацию о файловых системах, монтированных его клиентами? Насколько это важно?
9. Представьте, что из монтированного каталога системы NFS выполняется следующая команда

   ```
   echo hello > krishna.txt
   ```

   Какая последовательность запросов NFS будет создана в этом случае (файл krishna.txt не существует)?
10. Как изменится последовательность запросов, если файл krishna.txt уже существует?

11. Клиенты NFS заменяют операцию удаления открытого файла на переименование его на сервере и последующее удаление при закрытии. Что произойдет с файлом, если перед его удалением клиентская система выйдет из строя? Выскажите предположения о вариантах решения проблемы.
12. Системный вызов write является асинхронным. Он не ожидает сброса данных на постоянный носитель. Почему операция записи в NFS должна производиться синхронно? Как отказ клиента или сервера влияет на отложенную запись?
13. Почему NFS не подходит для работы в больших сетях?
14. Почему система RFS пригодна для использования только в однородных средах?
15. Насколько хорошо RFS удовлетворяет требованиям прозрачности сети, независимости и прозрачности расположения?
16. Какие улучшения AFS реализованы в системе DFS?
17. Каковы функции экспортера протокола в DFS?
18. Чем отличаются маркеры доступа DFS от обратной связи AFS?
19. Какие из файловых систем, описанных в этой главе, предусматривают мобильность пользователей или файлов?
20. Сравните пространство имен, видимых пользователем, в средах NFS, RFS, AFS и DFS.
21. Сравните семантику корректности данных при кэшировании на стороне клиента в системах NFS, RFS, AFS и DFS.

10.21. Дополнительная литература

1. Bach, M. J., Luppi, M. W., Melamed, A. S., and Yueh, K., «A Remote-File Cache for RFS», Proceedings of the Summer 1987 USENIX Technical Conference, Jun. 1987, pp. 273–279.
2. Bhide, A., Elnozahy, E., and Morgan, S., «A Highly Available Network File Server», Proceedings of the Winter 1991 USENIX Technical Conference, Jan. 1991, pp. 199–205.
3. Bhide, A., and Shepler, S., «A Highly Available Lock Manager for HA–NFS», Proceedings of the Summer 1992 USENIX Technical Conference, Jun. 1992, pp. 177–184.
4. Chartok, H., «RFS in SunOS», Proceedings of the Summer 1987 USENIX Technical Conference, Jun. 1987, pp. 281–290.
5. Cheriton, D. R., «The V Distributed System», Communications of the ACM, Vol. 31, No. 3, Mar. 1988, pp. 314–333.

6. Chutani, S., Anderson, O. T., Kazar, M. L., Leverett, B. W., Mason, W. A., and Sidebotham, R. N., «The Episode File System», Proceedings of the Winter 1992 USENIX Technical Conference, Jan. 1992, pp. 43–59.
7. Gerber, B., «AFS: A Distributed File System that Supports Worldwide Networks», Network Computing, May 1992, pp. 142–148.
8. Hitz, D., Harris, G., Lau, J. K., and Schwartz, A. M., «Using UNIX as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers», Proceedings of the Winter 1990 USENIX Technical Conference, Jan. 1990, pp. 285–295.
9. Hitz, D., Lau, J., and Malcolm, M., «File System Design for an NFS File Server Appliance», Proceedings of the Winter 1994 USENIX Technical Conference, Jan. 1994, pp. 235–245,
10. Honeyman, P., Huston, L. B., and Stolarchuk, M. T., «Hijacking AFS», Proceedings of the Winter 1992 USENIX Technical Conference, Jan. 1992, pp. 175–181.
11. Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., and Sidebotham, R. N., «Scale and Performance in a Distributed File System», ACM Transactions on Computer Systems, Vol. 6, No. 1, Feb. 1988, pp. 55–81.
12. Juszczak, C., «Improving the Performance and Correctness of an NFS Server», Proceedings of the Winter 1989 USENIX Technical Conference, Jan. 1989, pp. 55–63.
13. Juszczak, C., «Improving the Write Performance of an NFS Server», Proceedings of the Winter 1994 USENIX Technical Conference, Jan. 1994, pp. 247–259.
14. Kazar, M. L., «Synchronization and Caching Issues in the Andrew File System», Proceedings of the Winter 1988 USENIX Technical Conference, Feb. 1988, pp. 27–36.
15. Kazar, M. L., Leverett, B. W., Anderson, O. T,, Apostolides, V., Bottos, B. A., Chutani, S., Everhart, C. F., Mason, W. A., Tu, S. T., and Zayas, E. R., «Decorum File System Architectural Overview», Proceedings of the Summer 1990 USENIX Conference, Jun. 1990.
16. Levy, E., and Silberschatz, A., «Distributed File Systems: Concepts and Examples», ACM Computing Surveys, Vol. 22, No. 4, Dec. 1990, pp. 321–374.
17. Macklem, R., «Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol», Proceedings of the Winter 1991 USENIX Technical Conference 1991, pp. 53–64.
18. Moran, J., Sandberg, R., Coleman, D., Kepecs, J., and Lyon, B., «Breaking Through the NFS Performance Barrier», Proceedings of the Spring 1990 Europe Users' Group Conference, Apr. 1990, pp. 199–206.

19. Morris, J. H., Satyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S. H., and Smith, F. D., Andrew: «A Distributed Personal Computing Environment», Communications of the ACM, Vol. 29, No. 3, Mar. 1986, pp. 184–201.
20. Nowitz, D. A., «UUCP Administration», UNIX Research System Papers, Tenth Edition, Vol. II, Saunders College Publishing, 1990, pp. 563–580.
21. Open Software Foundation, «OSF DCE Application Environment Specification», Prentice-Hall, Englewood Cliffs, NJ, 1992.
22. Open Software Foundation, «OSF DCE Administration Guide — Extended Services», Prentice-Hall, Englewood Cliffs, NJ, 1993.
23. Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., and Hitz, D., «NFS Version 3 Design and Implementation», Proceedings of the Summer 1994 Technical Conference, Jun. 1994, pp. 137–151.
24. Postel, J., and Reynolds, J., «The File Transfer Protocol», RFC 959, Oct. 1985.
25. Plummer, D. C., «An Ethernet Address Resolution Protocol», RFC 826, Nov. 1982.
26. Rifkin, A. P., Forbes, M. P., Hamilton, R. L., Sabrio, M., Shah, S., and Yueh, K., «RFS Architectural Overview», Proceedings of the Summer 1986 USENIX Conference, Jun. 1986, pp. 248–259.
27. Sandberg, R., Goldberg, D., Kleiman, S. R., Walsh, D., and Lyon, B., «Design and Implementation of the Sun Network Filesystem», Proceedings of the Summer 1985 USENIX Technical Conference, Jun. 1985, pp. 119–130.
28. Sandberg, R., «Sun Network Filesystem Protocol Specification», Sun Microsystems, Inc., Technical Report, 1985.
29. Satyanarayanan, M., Howard, J. H., Nichols, D. A., Sidebotham, R. N., Spector, A. Z., and West, M. J., «The ITC Distributed File System: Principles and Design», Tenth ACM Symposium on Operating Systems Principles, Dec. 1985, pp. 35–50.
30. Sidebotham, R. N., «VOLUMES — The Andrew File System Data Structuring Primitive», Proceedings of the Autumn 1986 European UNIX Users' Group Conference, Oct. 1986, pp. 473–480.
31. Spector, A. Z., and Kazar, M. L., «Uniting File Systems», Unix Review, Vol. 7, No. 3, Mar. 1989, pp. 61–70.
32. Steiner, J. G., Neuman, C., and Schiller, J. I., «Kerberos: An Authentication Service for Open Network Systems», Proceedings of the Winter 1988 USENIX Technical Conference, Jan. 1988, pp. 191–202.
33. Stolarchuk, M. T., «Faster AFS», Proceedings of the Winter 1993 USENIX Technical Conference, Jan. 1993, pp. 67–75.
34. Sun Microsystems, Inc., «XDR: External Data Representation Standard», RFC 1014, DDN Network Information Center, SRI International, Jun. 1987.

35. Sun Microsystems, Inc., «RPC: Remote Procedure Call, Protocol Specification, Version 2», RFC 1057, DDN Network Information Center, SRI International, Jun. 1989.
36. Sun Microsystems, Inc., «Network File System Protocol Specification», RFC 1094, DDN Network Information Center, SRI International, Mar. 1989.
37. Sun Microsystems, Inc., «NFS Version 3 Protocol Specification», RFC 1813, DDN Network Information Center, SRI International, Jun. 1995.
38. Tannenbaum, A. S,, and Van Renesse, R., «Distributed Operating Systems», ACM Computing Surveys, Vol. 17, No. 4, Dec. 1985, pp. 419–470.
39. Tannenbaum, A. S., Van Renesse, R., Van Staveren, H., Sharp, G. J., Mullender, S. J., Jansen, J., and Van Rossum, G., «Experiences with the Amoeba Distributed Operating System», Communications of the ACM, Vol. 33, No. 12, Dec. 1990, pp. 46 –63.
40. Wittle, M., and Keith, B., «LADDIS: The Next Generation in NFS File Server Benchmarking», Proceedings of the Summer 1993 USENIX Technical Conference, Jun. 1993, pp. 111–128.

Глава 11
Усовершенствованные файловые системы

11.1. Введение

Развитие операционных систем должно идти в ногу с усовершенствованием вычислительной техники и аппаратных архитектур. Как только появляется новый или более быстрый компьютер, операционная система должна изменяться таким образом, чтобы востребовать преимущества нового оборудования. На самом деле так и происходит, часто развитие компонентов компьютера и соответствующих частей операционных систем происходит параллельно. Появляющиеся разработки изменяют требования к использованию ресурсов, поэтому создателям операционных систем также приходится вносить изменения в существующие правила.

С начала 1980-х годов произошли большие сдвиги в производительности процессоров, а также в объемах и скорости доступа к оперативной памяти [8]. Например, в 1982 году система UNIX работала преимущественно на машинах VAX 11/780, обладающих процессором с производительностью 1 миллион операций в секунду (MIPS) и объемом оперативной памяти 4–8 Мбайт. Система разделяла ресурсы между несколькими пользователями. К 1995 году на рабочих местах уже стояли персональные машины с производительностью около 100 MIPS и 32 или более мегабайтами оперативной памяти. К сожалению, технологии дисковых накопителей развивались не столь быстрыми темпами. Хотя жесткие диски дешевели с каждым днем и имели все большие объемы, их скорость за столь продолжительный период времени изменилась всего лишь в два раза. Операционная система UNIX, созданная изначально для работы с относительно быстрыми дисками на машинах с медленными процессорами и небольшими объемами памяти, должна была адаптироваться к имеющейся тенденции.

Применение традиционных систем (основанных преимущественно на вводе-выводе) на современных компьютерах привело к невозможности использования преимуществ быстрых процессоров и оперативной памяти. Как описано в [21], если время, нужное приложению для выполнения процессорных операций, равно c секунд, а время ввода-вывода составляет i секунд, то уве-

личение скорости CPU дает общий прирост производительности системы не более чем в (1+c/i) раз. Если значение i сравнимо с c, уменьшение c даст лишь небольшие выгоды. Становится очевидной необходимость поиска путей сокращения временных затрат на операции дискового ввода-вывода. Это один из явных путей увеличения производительности операционных систем.

В середине и конце 1980-х годов большинство систем UNIX использовало либо s5fs, либо FFS (см. главу 9) в качестве файловой системы для локальных дисковых накопителей. Обе системы подходят для общих приложений разделения времени, однако их недостатки проявляются при применении в коммерческих средах. Появление интерфейса vnode/vfs упростило задачу добавления в UNIX новых файловых систем. Однако интерфейс изначально разрабатывался для небольших специализированных реализаций, в которых не стояла проблема замены s5fs или FFS на иные файловые системы. Ограничения s5fs и FFS стали толчком к развитию нескольких усовершенствованных файловых систем, обладающих большими средствами и имеющих более высокую производительность. В начале 90-х годов некоторые из этих систем стали стандартными для основных вариантов UNIX. В этой главе мы обсудим недостатки традиционных файловых систем, ознакомимся с различными путями их устранения, а также рассмотрим некоторые файловые системы, ставшие альтернативами s5fs и FFS.

11.2. Ограничения традиционных файловых систем

Файловая система s5fs приобрела популярность прежде всего из-за простого дизайна и структуры. Однако она была слишком медленной и неэффективной, что привело к созданию FFS. Обе системы обладают некоторыми ограничениями, которые можно типизировать примерно так, как это сделано ниже.

- **Производительность.** Хотя производительность FFS намного выше по сравнению с s5fs, она недостаточна для коммерческих файловых систем. Принятая в FFS организация дисков дает возможность использования только части пропускной способности носителей. Более того, алгоритмы ядра основаны на большом количестве синхронных операций ввода-вывода, следствием чего являются слишком большие промежутки времени, необходимые для завершения системных вызовов.
- **Восстановление после сбоя.** Применение семантики буферного кэша приводит к потере данных и метаданных файлов в результате отказа, оставляющего систему в некорректном состоянии. Восстановление после сбоя проводится при помощи запуска программы sfck, просматривающей всю файловую систему на предмет обнаружения проблем и их решения по мере своих возможностей. При больших объемах дисков эта процедура может занять много времени, так как программа всегда

осуществляет проверку и перестройку диска целиком. Это приводит к неудовлетворительным по длительности задержкам до того момента, как машина перегрузится и снова станет доступной.

- **Безопасность.** Доступ к файлам в традиционных системах основан на привилегиях, ассоциированных с идентификаторами пользователей и групп. Доступ к файлу может быть разрешен только его владельцу, пользователям определенной группы или сразу всем. Такой механизм является недостаточно гибким в крупных вычислительных средах, для которых требуются более четко определенные средства управления доступом. С этой целью применяется технология списков контроля доступа (ACL), позволяющих владельцам файлов точно определять ограничения доступа к ним для тех или иных пользователей и групп. Индексные дескрипторы UNIX не поддерживают списки доступа, поэтому разработчикам файловых систем требуется искать альтернативные пути реализации ACL. Это, возможно, потребует изменения структур и методик размещения файлов на диске.
- **Размер.** Традиционные файловые системы обладают некоторыми ограничениями на размер как отдельных файлов, так и систем в целом. Отдельный файл или система должны занимать не более чем один раздел диска. Конечно, можно расположить на накопителе один раздел, занимающий все пространство диска, но и это не решит проблему, так как жесткие диски обычно имеют объем 1 Гбайт и меньше.[1] Для большинства целей такой вместимости достаточно, однако некоторые приложения работают с файлами больших размеров (например, системы управления базами данных или мультимедиа-приложения). Ограничение размера одного файла до 4 Гбайт (обусловленное длиной поля размером 32 бита) также является отрицательным фактором.

Рассмотрим подробнее некоторые проблемы производительности систем и средств восстановления после сбоев, а также некоторые варианты их решения.

11.2.1. Разметка диска в FFS

В отличие от файловой системы s5fs, разработчики FFS внесли в нее возможность оптимизации размещения блоков файлов, что позволило увеличить скорость последовательного доступа. Система по мере способностей старается размещать блоки каждого файла последовательно. Потенциальная возможность такого размещения сильно зависит от степени заполненности и фрагментации диска. Практика показала, что система может размещать файлы оптимально, если диск заполнен не более чем на 90% [10], [11].

---

[1] Современные жесткие диски имеют большие объемы. Однако за эти годы изменились и типичные размеры файлов и прикладных программ, так что описанная в книге проблема продолжает оставаться актуальной. — *Прим. перев.*

Наиболее важной проблемой являются задержки, имеющие место при перемещении головки между двумя отрезками файла. Файловая система FFS за один проход считывает или пишет только один блок данных. Если приложению нужно произвести последовательное чтение файла, оно выполнит серию операций чтения блоков. Между двумя операциями ядру необходимо проверить, не находится ли следующий блок в кэше, и, если это необходимо, произвести запрос ввода-вывода. В результате, если два блока расположены на находящихся рядом друг от друга секторах диска, для начала следующего чтения необходимо ждать, пока диск повернется до достижения начала второго блока. Следовательно, запаздывание второй операции чтения составит время почти полного оборота диска, что значительно ухудшает производительность системы.

Для предупреждения таких ситуаций в системе FFS приблизительно подсчитывается время, требуемое ядру для вызова следующей операции чтения, и вычисляется количество секторов, на которое успеет продвинуться головка диска за этот интервал. Такое число получило название задержки вращения (rotation delay) или rotdelay. Блоки располагаются на диске с учетом этого фактора, находясь на расстоянии друг от друга как раз на величину задержки (см. рис. 11.1). Для обычного диска один полный оборот занимает примерно 15 мс, ядру требуется порядка 4 мс на вызов следующей операции чтения. При размере блока 4 Кбайт и плотности размещения в 8 блоков на каждой дорожке величина задержки вращения равняется 2.

Описанный алгоритм защищает от необходимости ожидания полного оборота диска, но, с другой стороны, он ограничивает пропускную способность диска примерно до одной трети (в рассматриваемом примере). Увеличение размера блока до 8 Кбайт уменьшит значение фактора задержки до единицы, что даст возможность использовать примерно половину от пропускной способности диска. Это значение близко к максимальной скорости считывания с диска, при этом ограничение зависит прежде всего от архитектуры файловых систем. Если файловая система умеет считывать или записывать посекторно (или сразу по несколько секторов) в отличие от поблочных операций, она достигнет скорости ввода-вывода информации, близкой к максимальной пропускной способности диска.

Рис. 11.1. Размещение блоков на дорожке диска в системе FFS

При чтении файлов такие проблемы обычно не возникают. Это происходит благодаря высокоскоростному кэшированию диска. При каждом вызове read дорожка сохраняется в кэше целиком. Если при следующем чтении потребуется блок, находящийся на той же дорожке, то в ответ на запрос будет выдана информация из кэша со скоростью шины ввода-вывода. Такой подход дает возможность не терять время на дисковые операции. Системы кэширования диска являются обычно сквозными, следовательно до завершения операции каждая запись появится на соответствующем месте диска. Если кэш не будет сквозным, сбой диска может привести к потере данных, для которых известно, что операция записи завершилась успешно. Дисковое кэширование в состоянии увеличить скорость чтения, но операции записи по-прежнему остаются более длительными и зависят от фактора задержки вращения.

11.2.2. Преобладание операций записи

Наблюдения за использованием файловой системы и доступом к информации показали, что запросы на чтение данных или метаданных файлов численно превосходят операции записи в 1–2 раза. Однако среди дисковых запросов ввода-вывода, наоборот, преобладают запросы на запись. Такое необычное поведение файловой системы обусловлено буферным кэшем UNIX. Кэширование позволяет избавиться примерно от 80–90% операций чтения, осуществляя их без обращения к диску, так как приложения чаще всего обращаются к файлам, физически расположенным близко друг к другу.

Операции записи, как правило, также завершаются изменением копии данных в кэше, без необходимости осуществлять дисковый ввод-вывод. Такой подход, однако, требует периодической записи информации на диск, поскольку в противоположном случае сбой носителя может привести к потере больших объемов данных. В большинстве реализаций системы UNIX постоянно загружен демон update, который периодически сбрасывает «грязные» блоки на диск. Более того, некоторые операции требуют внесения изменений в индексные дескрипторы, а также в блоки прямой и косвенной адресации синхронно для того, чтобы файловую систему можно было восстановить после сбоя (более подробно этот вопрос рассмотрен в следующих разделах). Эти факторы являются причиной того, что запросы на запись информации на диск являются доминирующими. Увеличение объема памяти приводит к использованию дисковых кэшей больших размеров, что сокращает для операций чтения и так невысокий свойственный им дисковый трафик.

Некоторая часть операций записи является избыточной, поэтому от них можно отказаться. Типичные примеры использования файловой системы показали локальность ссылок: приложение с большой вероятностью будет

вносить изменения в один и тот же блок данных. Более того, многие файлы имеют очень малую продолжительность существования, они создаются, изменяются и удаляются в течение нескольких секунд. Для таких файлов операции записи не являются критичными.

Проблема потерь времени на запись файлов проявляется сильнее, если файловая система является экспортируемой из NFS. Технология NFS требует, чтобы все операции записи завершались сбросом информации на постоянный носитель. Это сильно увеличивает трафик, так как все операции записи данных и большинство записей метаданных необходимо производить синхронно.

Наконец, операция *позиционирования головки диска* (то есть перемещения головки на нужную дорожку диска) вносит существенный вклад в завершение операции ввода-вывода. Несмотря на то, что последовательный доступ к файлу позволяет избавиться от большинства операций поиска в правильно настроенных системах FFS, использование среды разделения времени (или систем, являющихся файл-серверами) сводит эти преимущества на нет. Она более разнородна, в результате чего диск используется преимущественно произвольно. Среднее время позиционирования становится в несколько раз выше задержки вращения для последовательных блоков FFS.

Как уже говорилось в разделе 11.2.1, применение дискового кэширования помогает избавиться от проблемы задержек вращения при чтении информации. К сожалению, это не относится к ее записи. Так как операции записи превалируют в дисковом обмене, разработчикам операционных систем необходимо найти другие пути решения описанной здесь проблемы.

11.2.3. Модификация метаданных

Существуют определенные системные вызовы, которые требуют проведения нескольких изменений метаданных. Для защиты файловой системы от повреждений в результате сбоя такие изменения обязательно должны вноситься в четко определенной последовательности. Например, при удалении файла (после сброса последней связи с ним) ядру необходимо произвести удаление вхождения каталога, освобождение индексного дескриптора и блоков на диске, ранее занимаемых файлом. Такие операции должны выполняться в вышеуказанном порядке для подтверждения корректности данных в случае отказа системы.

Представьте ситуацию, при которой файловая система сначала освобождает индексный дескриптор и только потом удаляет вхождение каталога. Между этими двумя операциями происходит сбой системы. В результате перед перезагрузкой каталог будет обладать ссылкой на неразмещенный индексный дескриптор. Если же сначала удалять вхождение каталога, возможное повреждение системы ограничивается существованием индексного дескриптора без ссылок, имеющего счетчик ссылок, не равный нулю. Единственная

неувязка для такого дескриптора в том, что его нельзя использовать повторно. Это менее критичная проблема для системы, которая может быть легко решена при помощи утилиты fsck. В предыдущем случае, если индексный дескриптор был передан другому файлу, утилита не будет знать, какое из вхождений каталогов является корректным.

Представим еще один сходный пример. При обрезке файла перед записью модифицированного индексного дескриптора на диск система освобождает дисковые блоки. Возможна ситуация, когда эти блоки запрашиваются для размещения информации другого файла, чей индексный дескриптор должен быть записан на диск раньше, чем дескриптор обрезаемого файла. Если в этот момент времени произойдет сбой системы, оба индексных дескриптора будут ссылаться на одни и те же блоки, что приведет к повреждению, видимому на прикладном уровне. Такая ошибка не может быть исправлена утилитой fsck, так как программа знает, какой из двух дескрипторов обладает более «законными» правами на обладание дисковыми блоками.

В традиционных файловых системах порядок операций поддерживается через синхронные операции записи. Это приводит к низкой производительности систем, особенно в тех случаях, когда запись на диск проводится непоследовательно, что требует дополнительного времени на позиционирование. К сожалению, операции NFS требуют синхронности записи, как правило, всех блоков данных. Становится очевидной необходимость разработки методики уменьшения общего количества синхронных операций записи, а также их локализации с целью снижения затрат на поиск секторов.

11.2.4. Восстановление после сбоя

Строгий порядок действий при записи метаданных помогает контролировать степень повреждений, возникающих вследствие отказа системы, но не дает полной защиты от проблем. В большинстве случаев упорядоченность дает гарантии возможности восстановления системы. Однако если секторы диска повреждены из-за аппаратных сбоев, полностью восстановить систему удается далеко не всегда. Для перестройки файловой системы после сбоя служит утилита fsck. Это программа прикладного уровня, работающая непосредственно на самом низком уровне доступа к устройству. Утилита fsck производит следующие действия.

1. Чтение и проверку всех имеющихся индексных дескрипторов и построение битовой карты используемых блоков данных.
2. Запись номеров индексных дескрипторов и блокировку адресов всех каталогов.
3. Проверку структуры дерева каталогов для удостоверения в действительности всех ссылок.
4. Проверку содержимого каталогов с целью учета наличия всех файлов.

5. Если какие-либо каталоги не были присоединены к дереву на шаге 2, программа переносит их в каталог lost+found.
6. Если какие-либо файлы не относятся ни к одному каталогу, программа переносит их в каталог lost+found.
7. Проверку всех битовых карт и подсчет статистики для каждой группы цилиндров.

Как можно видеть, программа fsck выполняет весьма значительный объем действий. Некоторым машинам с развитыми файловыми системами на это необходимо большое количество времени. Во многих средах такие длительные задержки до начала нормальной работы являются неприемлемыми. Следовательно, необходимо разработать некое альтернативное решение, позволяющее быстро восстанавливать систему после сбоев.

Также следует сказать о том, что утилита fsck имеет весьма ограниченные возможности восстановления. Ее применение гарантирует возврат системы в корректное состояние, однако надежные файловые системы нуждаются в нечто большем. Идеально было бы, конечно, полное восстановление после сбоев, но оно потребует, чтобы каждая операция завершалась сбросом информации на постоянный носитель. Такое правило применяется в NFS и некоторых не-UNIX-системах, таких как MS-DOS, однако оно имеет следствием низкую производительность систем. Оптимальным решением представляется уменьшение степени повреждений после сбоев, не ухудшающее при этом эффективность работы. Прочитав раздел 11.7, вы увидите, что реализация поставленной задачи возможна.

11.3. Кластеризация файловых систем (Sun-FFS)

Простейшим методом поддержки высокой производительности системы является кластеризация операций ввода-вывода. Большинство операций доступа к файлам в UNIX представляют собой последовательное чтение или запись файла, которое может требовать сразу нескольких системных вызовов. Очевидно, что ограничение возможностью обработки только одного блока (обычно равного 8 Кбайт) при вводе-выводе является неразумным. Во многих не-UNIX-системах файлы размещаются внутри одного или более *экстентов* (extents), то есть крупных непрерывных областей диска. Такой механизм размещения позволяет производить чтение или запись больших порций данных за одну дисковую операцию. Хотя *поблочное* размещение, принятое в UNIX, обладает несколькими преимуществами, такими как динамический рост размера файлов, оно не в силах повысить эффективность ввода-вывода.

С целью решения проблемы в файловой системе FFS, реализованной в SunOS, была применена технология *кластеризации файлов* [11], которая

позже была перенесена в SVR4 и 4.4BSD. В этой главе мы будем называть ее *Sun-FFS*. Целью разработанной методики было поддержание высокой производительности системы при увеличении дробления операций ввода-вывода. При этом дисковая структура файловой системы должна оставаться прежней. Реализация поддержки Sun-FFS требует внесения незначительного количества изменений во внутренние процедуры ядра.

Программа размещения дисковых блоков FFS прекрасно умеет распределять последовательные блоки файлов, используя сложный алгоритм, способный предугадывать последующие запросы на размещение. Технология Sun-FFS не отступила от достигнутого и продолжает поддерживать правило поблочного размещения данных.

В Sun-FFS фактор задержки вращения приравнивается к 0, так как главной целью методики было избавление от таких ожиданий. Поле суперблока maxcontig содержит количество сохраняемых последовательных блоков до появления задержки вращения. Это поле обычно устанавливается равным 1, что не имеет никакого значения, если задержка вращения нулевая. В Sun-FFS это поле используется для хранения выбранного *размера кластера*. Таким образом, в суперблок был добавлен новый параметр, важный для Sun-FFS, без изменения его структуры.

В любом случае, когда запрос на чтение требует доступа к диску, предпочтительнее считывать кластер целиком. Такая возможность была достигнута путем изменения взаимодействия с процедурой bmap(). В традиционной реализации FFS процедура bmap() использовала в качестве аргумента номер одного логического блока и возвращала для него физический номер. В Sun-FFS процедура возвращает дополнительную переменную contigsize, в которой указывается размер физически последовательного участка, начиная от указанного блока. Значение contigsize в большинстве случаев равняется maxcontig, даже тогда, когда файл может иметь и большее количество последовательно расположенных данных.

В Sun-FFS переменная contigsize применяется для чтения кластера целиком за один прием. Система, как правило, производит упреждающее чтение, кроме варианта, основанного на кластерах. В некоторых случаях программа размещения не может обнаружить целый кластер, поэтому значение contigsize, возвращаемое процедурой bmap(), будет меньше, чем maxcontig. Логика упреждающего чтения основана на размере contigsize, возвращаемого bmap(), а не на идеальном размере кластера.

Кластеризация записи требует внесения изменений в процедуру ufs_putpage(), производящей сброс страниц на диск. В Sun-FFS эта процедура оставляет страницы в кэше и успешно завершает работу, до тех пор пока кластер полностью не появится в кэше или последовательная запись не будет оборвана. Если это происходит, процедура вызывает bmap() с целью нахождения физического расположения страниц и записывает их все вместе за одну операцию. Если программа размещения не может поместить страницы на диск

последовательно, процедура bmap() вернет меньшую длину, что приведет к тому, что ufs_putpage() разобьет запись на две или более операции.

Несмотря на то, что в Sun-FFS были представлены лишь небольшие нововведения, такие как cache wiping, эти изменения показали преимущества кластеризации. Измерения производительности продемонстрировали, что скорость последовательного чтения с применением кластеризации увеличилась примерно в два раза, в то время как скорость произвольного доступа осталась такой же или незначительно большей по сравнению с традиционной файловой системой FFS. Технология кластеризации не смогла улучшить производительность записи NFS, так как в этой системе существует требование синхронной записи всех изменений на постоянный носитель. Для расширения преимуществ кластеризации на запись в NFS требуется произвести оптимизацию сбора записей в системе (технология описывалась в разделе 10.7.3).

11.4. Журналы

Во многих современных файловых системах применяется технология, получившая название *журналирования* (logging или journaling). Она позволяет избавиться от ограничений традиционных файловых систем, обсуждаемых в разделе 11.2. Основная концепция новой методики заключается в фиксировании всех изменений системы в файл журнала в режиме добавления. Журнал ведется последовательно, запись в него производится большими порциями за раз, что в результате дает высокую производительность и увеличивает эффективность использования диска. После сбоя системы необходимо проверить только конец журнала. Такой подход уменьшает время восстановления и увеличивает надежность системы.

Приведенное описание технологии, конечно, является сильным упрощением. Несмотря на очевидность преимуществ применения журналов, приходится иметь дело с весьма сложными вещами и находить компромиссы. На сегодня существует множество различных реализаций файловых систем с ведением журналов, используемых как в исследовательских, так и потребительских целях, основанных на совершенно отличающихся друг от друга архитектурах. Давайте начнем описание технологии с рассказа об имеющих принципиальное значение качествах, отличающих журнальные файловые системы от остальных, затем рассмотрим некоторые наиболее важные их реализации подробнее.

11.4.1. Основные характеристики

При разработке любой журнальной файловой системы необходимо найти ответ на ряд вопросов.

- **Какую информацию следует вносить в журнал?** Журнальные файловые системы развивались в двух направлениях. В одних принято веде-

ние записи всех изменений, в то время как в других в журнал вносятся модификации только метаданных. Журнализация метаданных позволяет ограничить объем записей выбранными операциями. Например, в иных случаях разумно не вносить в журнал изменения временных меток, владельцев или привилегий, записывая только те данные, которые влияют на корректность системы.

- **Операции или значения?** В журнал можно записывать либо отдельные операции, либо результаты операций. Первый вариант удобен, к примеру, при фиксировании изменений карты размещения блоков диска. Так как каждое такое изменение имеет дело лишь с небольшим количеством битов, запись будет компактнее. Однако если это операция записи данных, в журнал лучше внести содержимое измененного блока полностью.
- **Дополнение или замещение?** *Расширенные* (log-enhanced) *журнальные файловые системы* поддерживают традиционные дисковые структуры, такие как индексные дескрипторы или суперблоки. В таких системах журнал служит в качестве вспомогательного средства. В *структурированных* (log-structured) *журнальных файловых системах* журнал является представлением файловой системы на диске. Такой подход требует полного ведения журнала (то есть сохранения как данных, так и метаданных).
- **Журналы повторного или обратного выполнения?** Существуют два типа журналов: одни имеют только возможность *отмены* изменений (redo-only)[1], вторые разрешают их *отмену и восстановление* (undo-redo)[2]. В журнале первого типа осуществляется запись только модифицированных данных. В журнале второго типа сохраняются как старые, так и новые значения данных. Такой подход упрощает задачу восстановления после сбоя, но требует более точного соблюдения порядка ведения записей и модификаций метаданных (более подробно см. в разделе 11.7.2).
- **Хранить ли «мусор»?** Несмотря на то, что некоторые реализации систем позволяют создавать журналы бесконечной длины, помещая устаревшие части журнала на третичные носители, более популярным вариантом является поддержание файла конечного размера. Такой подход требует хранения устаревших частей журнала, интерпретируемых как логически замкнутый цикличный файл. На практике ведение такого журнала поддерживается самой системой, либо при помощи отдельной операции.
- **Как осуществлять группировку?** С целью достижения приемлемого уровня производительности файловая система должна производить запись данных в журнал большими порциями, по возможности объединяя

[1] Иначе — повторного выполнения. — *Прим. ред.*

[2] То есть обратное выполнение и повторное соответственно. — *Прим. ред.*

несколько мелких операций записи в одну. При принятии решения о частоте и объеме действий каждой операции записи нужно найти по возможности наилучшее соотношение между надежностью и производительностью, так как не сброшенные на носитель данные могут быть утеряны при сбое.

- **Как получать данные из журнала?** В структурированных файловых системах необходимо предусмотреть эффективный способ извлечения данных из журнала. Как правило, кэш большого объема позволяет избавиться от большинства операций чтения, что позволяет редко обращаться к дискам, однако при этом система должна уметь обрабатывать данные, не обнаруженные в кэше за приемлемый промежуток времени. Это требует применения эффективного механизма индексирования с целью нахождения произвольных блоков файла журнала.

11.5. Структурированные файловые системы

В структурированных файловых системах используется последовательное журналирование, при котором изменения дисковой структуры сохраняются в режиме дополнения. Главной целью существования такого журнала является сбор изменений в файловой системе в элемент большого объема и его запись на диск посредством неделимой операции. Такой подход требует детального пересмотра дисковой структуры файловой системы, а также процедур ядра, осуществляющих доступ к ней.

Преимущества методики кажутся впечатляющими. Так как запись данных осуществляется в только конец журнала, он всегда остается последовательным, что позволяет избавиться от необходимости поиска по диску. При каждой записи в журнал передается сразу же большой объем данных, обычно это целая дорожка диска, что не требует ожидания позиционирования головки (задержек вращения) и позволяет системе полностью использовать пропускную способность диска. Все компоненты файла (данные и метаданные) можно записать одной неделимой операцией, что увеличивает надежность системы и ставит ее в один ряд с системами, основанными на транзакциях. Восстановление после сбоя выполняется весьма быстро. Для этого файловой системе необходимо найти последнее корректное вхождение журнала и применить его для перестройки структуры. Все незаконченные операции, расположенные после такого вхождения, просто-напросто сбрасываются системой.

До сих пор все сказанное относилось к записи журнала. Возникает вопрос: как будет действовать система при необходимости получения данных из него? Традиционные механизмы обнаружения информации на диске (суперблоки, размещенные в фиксированном месте диска, группы цилиндров и индексные

дескрипторы) теперь недоступны; следовательно, нужно искать ее в журнале. Проблема в большинстве случаев решается путем использования кэша большого объема (современные системы, как правило, обладают внушительным количеством оперативной памяти и быстрыми процессорами, но медленными дисками). В системах, находящихся в стабильном состоянии, кэш большого объема в состоянии помочь избавиться примерно от 90% операций чтения. С другой стороны, для дисковых блоков, к которым требуется осуществлять доступ (таких блоков обычно много при первоначальной загрузке системы), необходимо найти способ обнаружения данных в журнале за приемлемый промежуток времени. Следовательно, полностью структурированная файловая система должна обладать эффективной методикой доступа к ее содержимому.

В 4.4BSD поддерживается структурированная файловая система под названием BSD-LFS [18]. Она основана на решении, сходном примененному в операционной системе Sprite [15]. Оставшаяся часть раздела будет посвящена описанию ее структуры, реализации и решению проблемы оптимального баланса между надежностью и производительностью.

11.6. Структурированная файловая система 4.4BSD

В BSD-LFS создается единый журнал для целого диска, который является единственным средством представления файловой системы. Запись данных производится в конец журнала, коллекция «мусора» создается процессом cleaner, что позволяет сделать журнал циклическим. Журнал разбивается на сегменты фиксированной величины (обычно 0,5 Мбайт). В каждом сегменте имеется указатель на следующий. В результате создается логически непрерывный журнал, при этом не требуется физически размещать сегменты друг за другом (следовательно, при переходе к новому сегменту иногда может потребоваться произвести его поиск на диске).

В файловой системе BSD-LFS оставлены структуры каталогов и индексных дескрипторов, а также схема косвенной адресации, используемые для адресации логических блоков файлов большого объема. Таким образом, если найден индексный дескриптор файла, доступ к нему может быть осуществлен традиционным способом. Важной характеристикой системы является технология поиска индексных дескрипторов. В оригинальной реализации FFS дескрипторы располагались на диске статически в четко оговоренных местах различных групп цилиндров, что давало системе возможность вычислять адрес индексного дескриптора на диске по его номеру.

В BSD-LFS индексные дескрипторы сохраняются на диске как часть журнала и не имеют фиксированных адресов. При каждом изменении дескриптор

записывается в новое место журнала. Такой подход требует применения дополнительной структуры данных, называемой картой индексных дескрипторов (inode map), в которой хранятся текущие дисковые адреса каждого объекта inode. Система BSD-LFS размещает эту карту в физической памяти, сбрасывая ее в журнал периодически при достижении определенных контрольных точек.

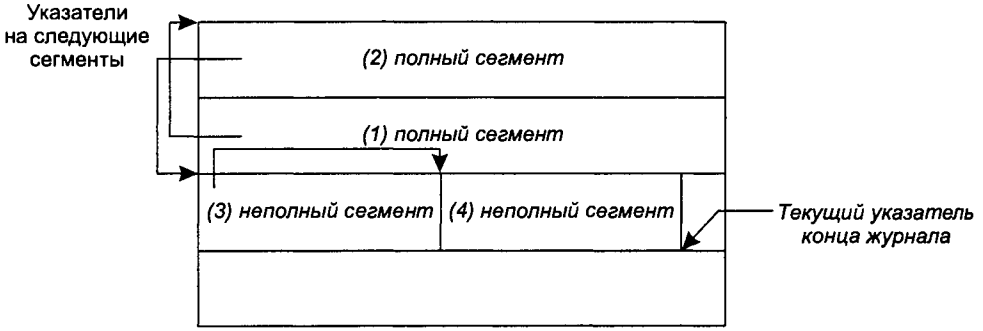


Рис. 11.2. Структура журнала файловой системы BSD-LFS

Несмотря на то что система старается производить запись целого сегмента сразу же, это часто просто невозможно. Часть сегмента может быть сброшена в журнал в результате уменьшения доступной памяти (переполнения кэша), запросов fsync или необходимости вызова операций NFS. Таким образом, *сегмент*, применяемый для физического деления диска, может состоять из одного или более *неполных сегментов*, создаваемых неделимой операцией записи в журнал (рис. 11.2).

Каждый неполный сегмент имеет заголовок, содержащий следующую информацию, используемую во время восстановления после отказа и сбора мусора:

- ✦ контрольные суммы для отслеживания ошибок носителей и незаконченных операций записи;
- ✦ дисковые адреса каждого индексного дескриптора, расположенного в неполном сегменте;
- ✦ номер индексного дескриптора и его версии, а также логические номера блоков для каждого файла, блоки данных которого расположены в сегменте;
- ✦ время создания, флаги и т. д.

Система также поддерживает *таблицу использования сегментов*, в которой хранится информация о количестве незанятых байтов каждого сегмента, а также о времени последней модификации сегментов. Процесс cleaner использует эти данные для очистки сегментов.

11.6.1. Запись в журнал

Система BSD-LFS собирает «грязные» блоки до тех пор, пока их не станет достаточно для заполнения целого сегмента. Неполный сегмент создается в случае необходимости, то есть при запросах к NFS, вызове fsync или уменьшении количества доступной памяти. Если дисковый контроллер поддерживает ввод-вывод методом сборки-рассоединения (scatter-gatter I/O), позволяя тем самым сбрасывать данные из непоследовательно расположенных участков памяти, то блоки будут записываться прямо из буферного кэша. Иначе ядро может запросить для выполнения передачи временные 64-килобайтовые буферы.

При подготовке к передаче дисковые блоки сортируются по логическим номерам блоков файлов. В это же время происходит назначение каждому блоку дискового адреса. Для вступления новых адресов в силу необходимо на том же этапе произвести изменения индексных дескрипторов (и, если необходимо, блоков косвенной связи). Такие индексные дескрипторы попадают в тот же сегмент вместе с другими «грязными» блоками метаданных.

Так как журнал ведется в режиме только добавления, каждый раз при модификации дискового блока запись изменений производится в новое место журнала. Это означает, что все предыдущие копии блока становятся устаревшими и могут быть удалены процессом очистки. Операция изменения данных и индексного дескриптора файла схематично показана на рис. 11.3.

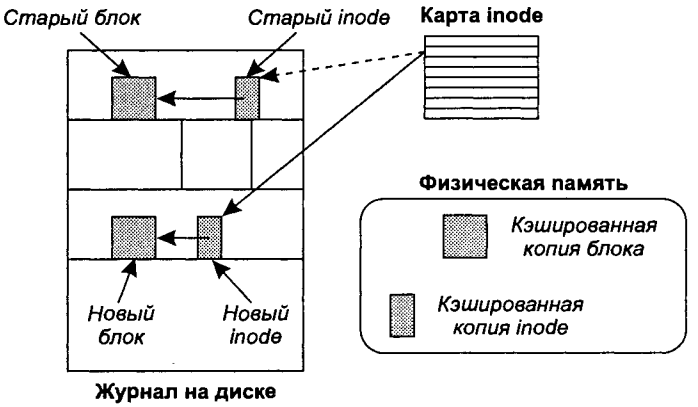


Рис. 11.3. Запись файла в файловой системе BSD-LFS

Каждая операция записи производит сброс всех «грязных» данных кэша. Это означает, что журнал содержит всю информацию, необходимую для полного восстановления системы. Карта индексных дескрипторов и таблица использования сегментов включает в себя избыточную информацию, которую можно получить из журнала, однако этот процесс будет слишком медленным. Эти две структуры данных располагаются в стандартном файле ifile, имеющем атрибут «только для чтения». Файл может быть доступен для пользователей

точно так же, как и любой другой. Его особенность состоит в том, что вносимые изменения не записываются системой вместе с каждым сегментом. Вместо этого в системе определены определенные *контрольные точки*, при достижении которых содержание ifile сбрасывается на диск. Хранение ifile в виде обычного файла позволяет динамически изменять общее количество индексных дескрипторов в системе.

11.6.2. Получение данных из журнала

Эффективные файловые операции требуют использования кэша большого объема, позволяющего производить максимум запросов без необходимости доступа к диску. Важно правильно интерпретировать отсутствие данных в кэше. Структуры данных системы BSD-LFS позволяют легко решить эту задачу. Поиск файлов осуществляется путем просмотра компонентов каталога по отдельности и получения номера индексного дескриптора следующего компонента (точно так же, как в системе FFS). Единственным отличием операции поиска в BSD-FFS является методика обнаружения индексных дескрипторов на диске. Вместо вычисления дискового адреса из номера индексного дескриптора система находит адрес в карте дескрипторов, используя их номера в качестве индекса к таблице.

Блоки данных, расположенные в кэше, идентифицируются и хэшируются при помощи vnode и номеров логических блоков. Однако блоки косвенной связи не так уж и легко увязать с этой схемой. В системе FFS они идентифицируются через vnode дискового устройства и физический номер блока. Так как система LFS не присваивает блокам дисковые адреса до тех пор, пока сегмент не станет готов к записи, не существует прямого способа отобразить «косвенные» блоки. В LFS применяется следующая методика, позволяющая обойти описанную проблему: каждый номер блока косвенной связи является обратным по отношению к номеру первого блока, к которому он относится. Каждый блок двойной косвенной связи имеет номер, равный или меньший на единицу, чем номер первого «косвенного» блока, на который он указывает, и т. д.

11.6.3. Восстановление после сбоя

Восстановление в системе BSD-LFS происходит быстро и просто. Для этого сначала производится поиск последней контрольной точки и инициализация в памяти карты индексных дескрипторов и таблицы использования сегментов по данным из журнала. Все изменения этих структур, внесенные после контрольной точки, восстанавливаются путем обратной прокрутки части журнала, расположенной вслед за ней. Перед тем как воспроизвести каждый неполный сегмент, система проверяет его временные метки для того, чтобы убедиться, что сегмент был изменен уже после контрольной точки. Восста-

новление завершается при обнаружении предыдущего варианта сегмента. Контрольная сумма, хранящаяся в заголовке каждого неполного сегмента, используется для подтверждения его завершенности и целостности. Если вычисляемая сумма не совпадает с указанной, сегмент (скорее всего, это последний вариант изменений) удаляется из журнала, после чего восстановление завершается. При выводе системы из строя могут быть утеряны только последний неполный сегмент, который еще не был записан на носитель, а также изменения журнала, внесенные при последней до сбоя операции записи.

Процедура восстановления системы работает весьма проворно. Ее скорость сопоставима с промежутком времени, прошедшим после достижения последней контрольной точки. Однако система не умеет определять аппаратные ошибки, повлекшие повреждения одного или нескольких дисковых секторов. Для комплексной проверки применяется программа, сходная с fsck, которая может выполняться в фоновом режиме после того, как система была введена в действие, путем «проигрывания» журнала.

11.6.4. Процесс cleaner

При достижении конца диска журнал становится циклическим. Если это произошло, файловой системе необходимо убедиться в том, что последующие записи в журнал не заменят собой важные данные. Это требует применения механизма сбора мусора, который выбирает из сегментов актуальные данные и сбрасывает их в новое месторасположение, что дает возможность повторного использования освободившегося пространства.

Сбор мусора может выполняться параллельно с другими действиями системы. С этой целью происходит чтение сегмента журнала и поиск в нем корректных вхождений. Элементы журнала могут стать недействительными в том случае, если после них следуют более свежие записи того же объекта (например, при проведении операций над одним и тем же файлом или каталогом) или при отсутствии объекта (например, при удалении файла). Если все вхождения сегмента являются актуальными, процедура соберет их и запишет в конец журнала (рис. 11.4). После этого сегмент становится доступным для повторного использования.

В системе BSD-LFS сбор мусора осуществляется прикладными процессами, называемые cleaner (процессами очистки). Они используют для своей работы набор специальных системных вызовов и файл ifile. Сначала процесс производит выбор сегмента из таблицы использования сегментов для последующей очистки. После этого он считывает его в свое адресное пространство. Для каждого неполного сегмента процесс последовательно просматривает блоки с целью определения, какие из них до сих пор являются актуальными. Он также выполняет анализ каждого индексного дескриптора на предмет сравнения его номера версии с указанным в таблице дескрипторов. Исполь-

зуемые блоки и индексные дескрипторы записываются обратно в файловую систему без внесения изменений в их индексные дескрипторы и счетчики доступа к ним. Система LFS сбрасывает их в следующий неполный сегмент, то есть на новое место. Работа процедуры завершается удалением устаревших индексных дескрипторов и блоков, а также маркировки сегмента как доступного для повторного использования.

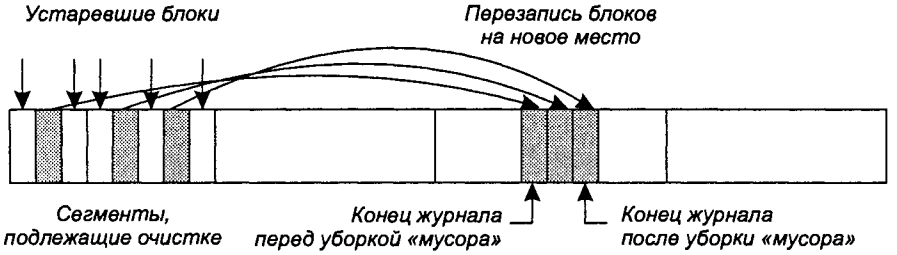


Рис. 11.4. Сбор мусора в журнале метаданных

Для работы программы очистки в систему были добавлены следующие четыре вызова:

lfs_bmapv	Вычисляет дисковые адреса для набора пар <индексный дескриптор, логический блок>. Если адрес блока совпадает с адресом очищаемого сегмента, блок считается актуальным
lfs_markv	Добавляет в журнал набор блоков без изменения информации о модификации индексных дескрипторов и количестве операций доступа к ним
lfs_segwait	Переводит процесс в режим сна на определенный период времени или до тех пор, пока не будет записан другой сегмент
lfs_segclean	Помечает сегмент как «чистый», что дает возможность его последующего использования

11.6.5. Анализ системы BSD-LFS

В файловой системе BSD-LFS существуют три проблемных ситуации. Во-первых, если операция над каталогом требует изменения более одного объекта метаданных, такие модификации может потребоваться производить в разных неполных сегментах. Для обнаружения этого, а также для корректного восстановления в случае незавершенной операции необходимо использовать дополнительные процедуры.

Во-вторых, размещение дисковых блоков в системе происходит не при первоначальном создании блока в памяти, а при записи сегмента на носитель. При этом необходимо тщательно отслеживать резерв свободного пространства, так как в ином случае система может показать, что вызов write завершился успешно, и только через некоторое время ядро обнаружит, что на диске отсутствует свободное место.

И последней проблемой BSD-LFS является требование дополнительного объема физической памяти, используемой не только для хранения буферного кэша, но и для больших структур данных и промежуточных буферов, необходимых для ведения журнала и сбора мусора.

В работах [18] и [19] подробно описываются эксперименты по сравнению производительности файловой системы BSD-LFS и традиционных вариантов FFS и Sun-FFS. Результаты измерений показали, что система BSD-LFS в большинстве ситуаций опережает традиционный вариант FFS (кроме случая сильной загруженности большим количеством вычислительных задач, где производительность LFS немного уступает). По сравнению с Sun-FFS система LFS более эффективна в тестах интенсивного использования метаданных (то есть преимущественно выполняющих вызовы create, remove, mkdir и rmdir). При оценке быстродействия операций read и write и общих многопользовательских тестах преимущества системы уже не так очевидны. Sun-FFS работает быстрее в большинстве тестов интенсивного ввода-вывода. Ее преимущества видны еще нагляднее, если в LFS выполняется процесс очистки. Обе системы сравнимы по производительности в обычных ситуациях многопользовательской работы при применении таких средств испытаний, как Andrew benchmark [6].

Измерения выявили, что система BSD-LFS имеет не самые лучшие показатели по отношению к остальным. Например, Sun-FFS работает с той же или даже чуть большей скоростью при намного меньших затратах на ее реализацию. Для поддержки LFS необходимо переписать заново не только файловую систему, но и основные утилиты, которые должны знать о реальной дисковой структуре, такие как newfs и fsck. Реальным преимуществом BSD-LFS является быстрота восстановления после сбоев и высокая продуктивность операций обработки метаданных. В разделе 11.7 будет описана технология ведения журнала метаданных, что позволяет добиться высоких показателей производительности системы с низкими затратами.

Необходимо упомянуть еще одну структурированную файловую систему под названием *Write-Anywhere File Layout* (*WAFL*), продукт корпорации Network Alliance Corporation для линейки специализированных серверов FAServer, работающих с NFS [5]. Разработчики WAFL создали структурированную файловую систему, использующую энергонезависимую память (NVRAM) и дисковый массив RAID-4, что обеспечило сверхбыструю реакцию на запросы NFS. В системе WAFL реализовано полезное средство под названием *моментальных снимков* (snapshots), являющихся неизменяемыми копиями активной файловой системы, доступными в режиме «только для чтения». Файловая система способна поддерживать большое количество «слепков» самой себя, сохраненных в разное время. Эти снимки можно использовать для получения предыдущих версий файлов или для «воскрешения» случайно удаленных файлов (операция undelete). Системные администраторы обладают правом использования моментальных снимков, представляющих целостный образ файловой системы в определенный момент времени, для ее восстановления.

11.7. Ведение журнала метаданных

В системах журналирования метаданных журнал является дополнением к обычной структуре файловой системы. Это значительно упрощает задачу его реализации, так как не требуется вносить модификации в так называемые *неизменяющие* (nonmutative) операции (не вносящие изменений в файловую систему). Дисковая структура файловой системы остается все той же, доступ к данным и метаданным на диске (не оказавшимся в кэше) может осуществляться традиционными способами. Журнал считывается только при восстановлении после сбоя или при сборе мусора. Все остальные компоненты файловой системы используют журнал только в режиме добавления.

Такой подход дает возможность задействовать главное преимущество журнальной технологии — реализовать быстрое восстановление после сбоя и высокоскоростные операции обработки метаданных без груза недостатков структурированных файловых систем (среди которых сложность реализации, требование создания специализированных версий утилит, падение производительности во время «уборки мусора»). Запись метаданных минимально влияет на скорость обычных операций ввода-вывода, однако требует внимательности при ее реализации с целью защиты от снижения общей производительности работы системы.

Журналы метаданных, как правило, содержат изменения индексных дескрипторов, блоков прямой и косвенной адресации каталогов. В них также могут вноситься изменения суперблоков, данные о группах цилиндров и карты размещения дисков. Система может избирательно использовать эту информацию при восстановлении после сбоев.

Журнал бывает как частью файловой системы, так и независимым внешним объектом. Обычно выбор производится из соображений более эффективного использования диска и общей производительности. Например, в файловой системе Cedar [3] журнал реализован в виде обычного файла фиксированного размера, использующего заранее выделенные ему блоки в середине диска (с целью быстрого доступа к ним). Файл журнала является таким же, как и остальные: он обладает именем и индексным дескриптором и может быть доступен без применения каких-либо дополнительных средств. В системе Calaveras [23] все файловые системы, имеющиеся на машине, разделяют между собой единый журнал, расположенный на отдельном диске. В файловой системе Veritas [26] журнал хранится отдельно от файловой системы, а системный администратор вправе выделить для него отдельный диск.

11.7.1. Функционирование в обычном режиме

В качестве первого примера рассмотрим схему ведения журнала, являющегося доступным только для операций восстановления (redo-only), хранящего новые значения изменяемых объектов. Подобная схема применяется в фай-

ловой системе Cedar [3]. Журнал не используется при записи данных файлов, которые обрабатываются традиционными методами. На рис. 11.5 схематично показаны действия операции setattr, производящей изменения одного индексного дескриптора. При ее вызове ядро выполняет определенную последовательность действий.

1. Изменяет копию в кэше и помечает ее как «грязную».
2. Создает вхождение журнала, которое содержит заголовок, идентифицирующий модифицируемый объект, а также его новые данные.
3. Записывает вхождение в конец журнала. После завершения записи считается, что операция зафиксировала информацию на диске.
4. Реально сброс индексного дескриптора на диск осуществляется позже. Такая технология называется *модификацией по месту* (in-place update).

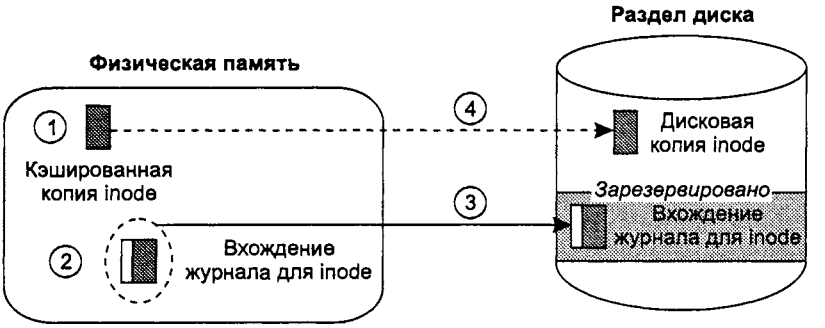


Рис. 11.5. Реализация записи журнала метаданных

Этот простой пример показывает, как журнал влияет на производительность системы. С одной стороны, каждое изменение метаданных сбрасывается на диск дважды — первый раз в качестве элемента журнала и второй раз при модификации по месту. С другой стороны, операции записи по месту обычно являются отложенными, то есть они часто пропускаются и выполняются в пакетном режиме. Например, один и тот же индексный дескриптор до сброса на диск может быть несколько раз изменен или же путем единичной операции производится запись сразу же нескольких дескрипторов в один дисковый блок. Ведение журнала метаданных представляется вполне оправданной мерой, лишние действия системы компенсируются малостью числа модификаций по месту.

Как правило, для журнала может быть применена пакетная обработка. Многие файловые операции производят изменения сразу нескольких объектов метаданных. Например, функция mkdir изменяет родительский каталог и его индексный дескриптор, а также производит размещение и инициализацию нового каталога и его индексного дескриптора. Некоторые файловые системы собирают все изменения, сделанные одной операцией, в одно и то же

вхождение журнала. Существуют системы, которые даже умеют «коллекционировать» в один элемент журнала изменения, произведенные несколькими операциями.

Такое решение влияет не только на производительность, но в большей степени на гарантии надежности и целостности файловой системы. При крахе она может потерять изменения, которые не были сброшены в журнал. Если файловая система применяется для доступа к NFS, она не сумеет ответить на клиентские запросы до тех пор, пока изменения гарантированно не окажутся в журнале. При попытке модификации одного объекта несколькими операциями одновременно все действия необходимо производить последовательно, так как в обратном случае может быть повреждена целостность объекта. Более подробно об этом рассказывается в разделе 11.7.2.

Журнал имеет фиксированный размер, при достижении конца он продолжает заполняться циклически. Файловая система должна уметь предупреждать перезапись актуальных данных. Вхождение журнала находится в памяти до тех пор, пока все его объекты не будут сброшены на соответствующие места на носителе (то есть будет произведена модификация по месту). Существуют два условия циклического перехода. Во-первых, такой переход может иметь место при сборе мусора (точно так же, как и в BSD-LFS). Процесс очистки должен постоянно «находиться» в районе начала журнала и освобождать его путем перемещения активных вхождений в конец журнала [3]. Простейшим вариантом является упреждение модификаций по месту процессом очистки. Если журнал имеет достаточную величину, позволяющую придерживать фиксирование изменений по месту и выполнять свою очистку только при пиковой загрузке, сбор мусора может быть произведен в совокупности. Это не требует файла журнала громадного размера — для небольших серверов и систем разделения времени достаточно всего лишь нескольких мегабайтов.

11.7.2. Целостность журнала

В журнальных файловых системах при изменении нескольких объектов одной операцией необходимо следить за корректностью журнала (см. раздел 11.2.3). Система также должна обеспечивать ее при одновременных вызовах операций для одних и тех же наборов объектов. В этом разделе будет рассказано о методике поддержания корректности журнала регистрации отмен-восстановлений. Проблемам, связанным с этими журналами, будет посвящен раздел 11.8.3.

При использовании журнала регистрации восстановлений (redo-only) восстановление после сбоя производится путем его обратной «прокрутки» и записи каждого вхождения на его актуальное место на диске. Это подразумевает, что журнал содержит самые свежие копии объектов метаданных. Следовательно, при нормальном функционировании система никогда не должна производить модификацию по месту до того, как объект будет внесен

в журнал. Такой журнал часто называют *журналом намерений* (intent log) [26], поскольку каждое его вхождение содержит объекты, которые файловая система намеревается записать на диск.

Представьте, что некая операция имеет целью модификацию двух объектов метаданных, А и Б. Надежная файловая система может предоставить возможность либо указания последовательности обработки, либо гарантии транзакций для множественных изменений. *Гарантии порядка следования* означают, что после восстановления на диске будет находиться новое содержание объекта Б только в том случае, если и объект А содержит новые данные. *Гарантии транзакций* более строги: после восстановления будут внесены изменения для обоих объектов, но не для одного из них.

В журналах регистрации восстановлений поддержка гарантий порядка следования может быть осуществлена путем задержки модификации по месту объектов метаданных до того, как их журнальные вхождения будут сброшены на диск. Система записывает объекты в журнал в том порядке, как они изменялись или создавались. В предыдущем примере запись вхождения для объекта А будет произведена до записи элемента для Б (данные об изменении А и Б могут также быть объединены в одном вхождении журнала). Такой подход позволяет учесть очередность изменений даже в том случае, если модификация объекта Б по месту произошла раньше, чем А.

Гарантии транзакций в журналах регистрации восстановлений поддерживаются в том случае, если ни один объект нельзя сбросить на диск до тех пор, пока операция записи вхождений журнала для обоих блоков успешно не завершится. Это может быть простейшей задачей в том случае, если два блока при сбросе уже объединены в одном вхождении журнала, что бывает далеко не всегда. Представьте, что модификация по месту для объекта А имела место до записи вхождения журнала для Б. Между этими двумя действиями система вышла из строя. Восстановить предыдущую копию А или новую копию Б невозможно. Следовательно, нужно осуществлять запись обоих вхождений журнала на диск до того, как будет произведена даже запись кэшированной копии. В журнал также необходимо добавлять информацию, которая будет указывать на то, что оба изменения относятся к одной и той же транзакции. Это позволит при восстановлении системы не производить повторных частичных транзакций.

Фактически существует более строгое требование, относящееся к параллельным операциям над одним и тем же объектом. *Нельзя производить даже чтение модифицированного объекта до тех пор, пока он не будет внесен в журнал.* На рис. 11.6 продемонстрировано, как может возникнуть состязательность при несоблюдении этого правила. Процесс П1 изменяет объект А и собирается произвести его запись, сначала в журнал, затем — на диск. Однако в тот же момент времени процесс П2 выполняет чтение объекта А и на его основе изменяет содержимое объекта Б. Затем он записывает Б в журнал и собирается сделать его запись на диск. Если в этот момент времени система

откажет, журнал будет содержать новое значение Б, однако новые данные А не окажутся ни на диске, ни в журнале. Так как изменения Б зависят от изменений А, возникшая ситуация является потенциально тупиковой.

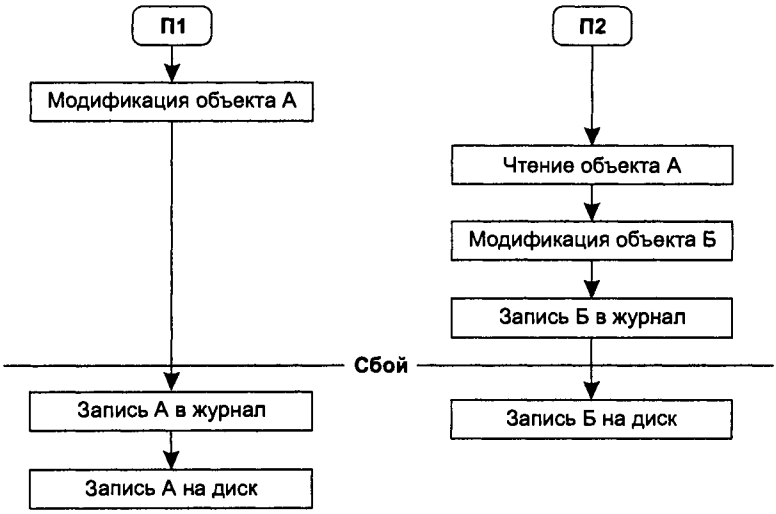


Рис. 11.6. Пример состязательности при использовании журнала восстановления

Для получения более понятной картины представим более жизненный пример. Пусть процесс П1 производит удаление файла из каталога, в то время как П2 создает в том же каталоге файл с точно таким же именем. Процесс П1 удаляет имя файла из блока А каталога. Процесс П2 обнаружит, что каталог не содержит файла с таким именем, и продолжит создавать вхождение каталога в блоке Б. При восстановлении после сбоя в системе будут существовать предыдущая копия блока А и новый блок Б, оба будут иметь ссылку на одно и тоже имя файла в каталоге.

11.7.3. Восстановление

При крахе системы ее восстановление осуществляется путем обратной прокрутки журнала и использования его вхождений для модификации метаданных. В этом разделе описывается процесс реставрации системы из журнала регистрации операций восстановления. Эти действия на основе журналов отмен-восстановлений будут обсуждаться чуть позже, в разделе 11.8.3. Основной проблемой является определение начала и конца журнала, так как он ведется циклически. В [23] излагается один из вариантов решения этой задачи. При нормальном функционировании файловая система присваивает каждому вхождению журнала номер. Число, являющееся номером, монотонно возрастает и указывает на позицию вхождения в журнале. После достижения конца журнала (и перехода к его началу) номера записываемых элементов

продолжают возрастать. Следовательно, в любой момент времени существует взаимосвязь между номером вхождения и его месторасположением в журнале (смещением от начала, измеряемым 512-байтовыми порциями), которая может быть рассчитана по формуле:

```
entry location = entry number % size of log
или местораположение элемента = номер элемента % размер журнала
```

Файловая система постоянно отслеживает номера первого и последнего активных элементов журнала и записывает их значения в заголовок каждого вхождения журнала. Для обратной прокрутки журнала системе необходимо найти вхождение, обладающее максимальным номером. Такой номер будет указывать на конец журнала, заголовок вхождения также будет содержать номер элемента текущего начала журнала.

После идентификации конца и начала журнала система производит восстановление путем записи объектов метаданных из каждого элемента в их действительные месторасположения на диске. Заголовок вхождения содержит информацию о физическом размещении и данные идентификации каждого составляющего его объекта. Такой подход приводит при сбое системы к потере всего лишь небольшого количества метаданных, которые не были записаны в журнал. Неполные вхождения журнала обнаружить очень просто, так как в каждом элементе имеется либо его контрольная сумма, либо трейлер[1]. Такие неполные вхождения при восстановлении удаляются системой.

Время, требуемое для восстановления, не зависит от размера файловой системы, но пропорционально размеру журнала на момент краха. Объем журнала зависит от загруженности системы и частоты модификаций по месту. Системы сбора метаданных обычно отрабатывают за секунды, что отличает их от систем, применяющих утилиту fsck, требующую нескольких минут работы, а иногда и нескольких часов. Однако и в журнальных системах необходимо периодически запускать утилиты проверки диска, так как ведение журнала не спасает от повреждений, причинами которых являются ошибки на жестких дисках. Такие программы могут выполняться в фоновом режиме уже после восстановления и загрузки системы.

11.7.4. Анализ

Методика журналирования метаданных обладает важнейшими преимуществами этой технологии, а именно свойствами быстрого восстановления после сбоев и высокой скорости операций с метаданными. При этом для реализации таких систем нужно затратить значительно меньше усилий по сравнению со структурированными журнальными системами. Восстановление после сбоя реализовано путем обратной прокрутки журнала и записи объектов метаданных

[1] Запись с контрольной суммой в конце блока данных. — *Прим. ред.*

по факту их расположения на диске. Процедура, как правило, занимает меньше времени, чем проверка дисков утилитами, такими как fsck.

Ведение журнала метаданных увеличивает скорость операций, производящих изменение сразу нескольких объектов метаданных (например, mkdir или rmdir), так как система умеет собирать все модификации в одно вхождение журнала и записывать их при помощи одной-единственной операции. Это уменьшает общее количество синхронных операций записи. Система также поддерживает либо гарантии порядка следования, либо гарантии транзакций (в зависимости от конкретной ее реализации) для зависимых друг от друга изменений метаданных. Такой подход делает системы регистрации изменений метаданных более надежными по сравнению с традиционными архитектурами.

Влияние журналирования на общую производительность системы представляется не совсем однозначным. Ведение журнала никак не отражается на операциях, не изменяющих файловую систему, и лишь немного сказывается при записи данных. Ведение журнала в целом уменьшает количество операций записи метаданных по месту путем задержки операций. Для поддержания приемлемого уровня производительности системы такое уменьшение числа записей должно скомпенсировать затраты на поддержание журнала.

В работе [23] показано, что журнал может оказаться узким местом системы, и описаны некоторые методики оптимизации, позволяющие избавиться от этой проблемы. В книге также приводится описание большого количества экспериментов по сравнению производительности двух схожих файловых систем, отличающихся друг от друга наличием журнала. Журнальная файловая система сильно опережает соперницу в тестах интенсивного использования метаданных, однако в тесте LADDIS (эмулирующем многопользовательский доступ к NFS) лишь немного отличается от нее по скорости [25].

Методика записи метаданных обладает некоторым количеством ограничений и недостатков. Хотя она и позволяет минимизировать потери изменений метаданных при сбое, для обычных файлов потери никак не ограничены (кроме выполняемых демоном update). Это также означает невозможность гарантировать корректность транзакций для всех операций: если операция выполняет изменение не только файла, но и его индексного дескриптора, модификация не будет произведена атомарно. Следовательно, в результате краха системы только один из компонентов такой транзакции может быть восстановлен.

В целом ведение журнала метаданных привносит дополнительную надежность и ускоряет восстановление системы, а также дает небольшое увеличение производительности без необходимости изменений дисковой структуры файловой системы. Методика относительно легка для практической реализации, так как для этого достаточно скорректировать только ту часть системы, которая отвечает за операции записи метаданных.

В сообществе разработчиков пользователей системы UNIX некоторое время велись жаркие споры о том, какие системы лучше — структурирован-

ные или ведения журналов метаданных. Результатом стало одобрение методики записи метаданных, на основе которой было создано несколько удачных коммерческих продуктов, в том числе: *Veritas File System* (VFS) компании Veritas Corporation, *Journaling File System (JFS)* корпорации IBM и система *Episode File System* компании Transarc (см. раздел 11.8). Реализация ведения записи метаданных не требует внесения изменений в коды, отвечающие за передачу данных, поэтому в систему можно добавлять такие дополнительные возможности, как кластеризация (см. раздел 11.3) или сбор записей NFS (см. раздел 10.7.3). В результате можно создать систему, обладающую высокой производительностью операций обработки не только метаданных, но и обычных данных.

11.8. Файловая система Episode

В 1989 году разработка системы Andrew File System перешла от Университета Карнеги—Меллона к корпорации Transarc (подробнее об AFS можно прочесть в разделе 10.15). Позже система AFS была преобразована в Episode, ставшую компонентом локальной файловой системы *распределенной среды вычислений* (Distributed Computing Environment, DCE) OSF. В файловой системе Episode были представлены некоторые дополнительные средства, отличающие ее от традиционных файловых систем UNIX, такие как усовершенствованная защита, файлы больших размеров, журналы и отделение понятия физического хранения от абстракции логической структуры файловой системы. В разделе 10.18 была описана система DCE DFS (Distributed File System), в этой части книги будет рассказано о структуре и возможностях системы Episode.

11.8.1. Основные понятия

В системе Episode были представлены некоторые новые понятия, называемые *агрегатами*, *контейнерами*, *наборами файлов* и *объектами anode*. *Агрегат* (aggregate) является обобщением раздела и указывает на логически последовательный массив дисковых блоков. Он скрывает детали физического разбиения дисков от остальной части файловой системы. Агрегат может состоять как из одного, так и из нескольких дисковых разделов[1] и предоставлять такие мощные возможности, как зеркалирование (mirroring) и расщепление (striping) дисков. Агрегаты позволяют отдельным файлам оккупировать более одного физического диска, что дает возможность создания файлов циклопических размеров.

[1] Журнальная файловая система (JFS) корпорации IBM была одной из первых систем, позволяющих логическим томам физически занимать несколько дисковых разделов.

Набор файлов — это логическая файловая система, состоящая из дерева каталогов, основанием которого является корневой каталог набора данных. Каждый такой набор файлов можно монтировать и экспортировать независимо[1]. Агрегат может включать в себя один или более наборов файлов. Наборы файлов позволено переносить из одного агрегата в другой прямо во время работы системы и использовать их для обычных файловых операций. Взаимосвязь между агрегатами, наборами файлов и физическими дисками показана на рис. 11.7.

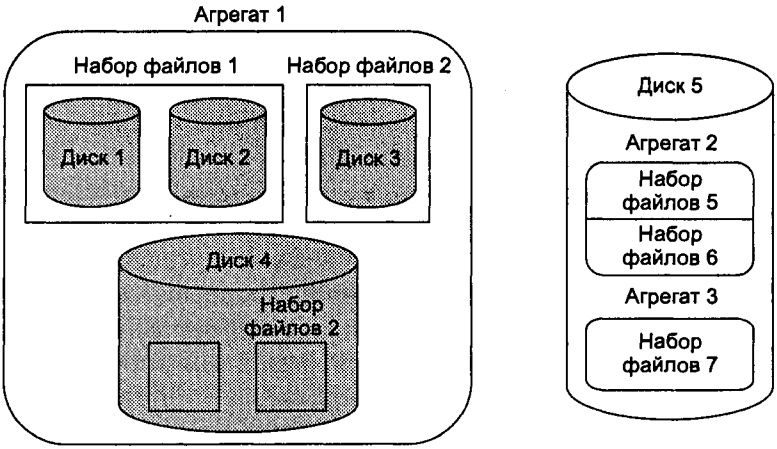


Рис. 11.7. Организация хранения в файловой системе Episode

Контейнер — это объект, в котором хранятся данные. Он имеет блочную структуру. Каждый набор файлов находится внутри контейнера, в котором размещаются все данные и метаданные файлов, принадлежащих к этому набору. Каждый агрегат имеет три дополнительных контейнера, для битовой карты, журнала и таблицы набора файлов агрегата (более подробно об этом будет рассказано чуть позже).

Объект *anode* аналогичен индексному дескриптору UNIX (inode). Для каждого файла и контейнера в системе Episode выделяется один объект anode.

11.8.2. Структура системы

Агрегат состоит из нескольких контейнеров. В *контейнере набора файлов* (fileset container) хранятся все файлы и их объекты anode. Объекты anode постоянно находятся в *таблице anode набора файлов*, в заголовке контейнера, после которого размещаются непосредственно данные и блоки косвенной связи. Контейнер не занимает определенное непрерывное пространство внутри агрегата, следовательно, он легко может расширяться и расти динамиче-

[1] Современные инструменты DCE позволяют экспортировать только сразу все наборы файлов агрегата.

ски. Таким образом, адреса блоков файлов в системе Episode указывают на номера блоков внутри агрегата, но не внутри контейнера.

Контейнер битовых карт (bitmap container) предоставляет агрегату широкие возможности по размещению блоков. Для каждого фрагмента агрегата в этом контейнере хранится информация о том, где он размещен и применяется ли он для данных, вошедших или не вошедших в журнал. Эти сведения необходимы специализированным функциям, вызываемым в случае повторного использования фрагмента, внесенного в журнал, для нежурнальных данных, а также в обратном случае.

Контейнер журнала содержит журнал отмен-восстановлений агрегата, в котором ведется запись исключительно метаданных. О преимуществах журналов отмен-восстановлений будет рассказано в разделе 11.8.3. Журнал имеет фиксированный размер и используется циклически. Хотя текущие реализации системы обычно располагают журнал в том же агрегате, который он представляет, это не является обязательным требованием.

Таблица наборов файлов агрегата (см. рис. 11.8) содержит суперблок и дескрипторы anode для каждого контейнера, принадлежащего агрегату. Вхождения каталогов ссылаются на файлы через идентификатор набора файлов (fileset ID) и номер anode в наборе. Для обнаружения файла необходимо сначала произвести поиск дескриптора anode набора в таблице наборов файлов агрегата и затем использовать индексирование в таблице объектов anode набора файлов. Привлечение кэширования значительно увеличивает скорость всех операций.

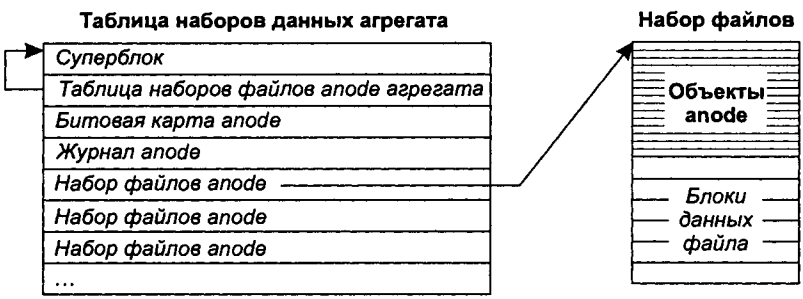


Рис. 11.8. Таблица наборов файлов агрегата и набор файлов

Контейнеры поддерживают три режима хранения: простой (in-line), фрагментированный и блочный. Каждый anode имеет некоторое дополнительное пространство, при использовании простого режима в нем хранятся небольшие объемы данных. Такой режим очень удобен для символических ссылок, списков управления доступом и файлов небольших размеров. Блочный режим позволяет использовать контейнеры огромных размеров и поддерживает четыре уровня косвенной адресации. В таком случае максимальный размер файла может быть равен 2^{31} дисковым блокам[1].

[1] Позже размер файла стал ограничиваться величиной (2^{32} × размер фрагмента).

11.8.3. Ведение журнала

В системе Episode используется журнал отмен-восстановлений, в котором сохраняются только метаданные, с предоставлением гарантий транзакций, ранее описанных в разделе 11.7.2. Журналы отмен-восстановлений (undo-redo) предлагают высокую степень гибкости, так как в каждом их элементе хранится не только предыдущее, но и новое значение объекта. При восстановлении после сбоя файловая система может прокрутить журнал так, что присвоит новые значения дисковым объектам, либо откатиться назад, восстановив предыдущие значения.

Гарантии транзакций в журналах восстановлений (redo-only) требуют использования *двухфазного протокола блокировки*, обязывающего все блокировать все объекты, участвующие в транзакции, до тех пор, пока все данные не будут сброшены на диск. Протокол гарантирует, что никакая иная транзакция не сможет прочесть данные, не появившиеся на диске. Такой подход уменьшает параллелизм системы и отрицательно влияет на ее производительность. В файловой системе Episode выход из положения был найден при помощи механизма под названием *классов эквивалентности* (equivalence class). Классы содержат все активные транзакции, имеющие дело с одними и теми же объектами метаданных. Свойством классов эквивалентности является возможность сбрасывать на диск все транзакции сразу, но не часть из них.

В случае сбоя процедура восстановления производит прокрутку всех укомплектованных эквивалентных классов, производя откат назад в том случае, если класс оказывается неполным. Такой подход позволяет добиться высокой степени параллелизма работы при нормальном режиме функционирования системы. Однако это удваивает размер каждого элемента журнала, увеличивает трафик ввода-вывода на диск журнала и усложняет процедуру восстановления.

В системе Episode буферный кэш тесно связан со средствами ведения журнала. Вызовы высокого уровня не производят изменения буферов напрямую, используя для этой цели функции поддержки журнала. Журнальные процедуры сравнивают буферы с элементами журнала и гарантируют, что буфер не был сброшен на диск раньше, чем запись в журнал удачно завершилась.

11.8.4. Другие возможности

Клонирование (cloning). Наборы файлов можно дублировать или переносить из одного агрегата в другой при помощи процесса, получившего название клонирования наборов файлов. Каждый дескриптор anode клонируется индивидуально, клоны разделяют блоки данных с оригинальными дескрипторами anode, используя технологии копирования при записи. Клонированный набор файлов доступен только для чтения и находится в том же агрегате, что и его оригинал. Клоны можно создавать весьма быстро, не мешая доступу

к оригинальному набору файлов. Клоны часто применяются различные инструментами управления системой, например программами резервного копирования, имеющими дело не с оригиналом, а его фантомом. Если происходит изменение какого-либо блока оригинального набора, необходимо сделать новую копию, а версия, представленная клоном, остается неизменной.

Безопасность. Система Episode поддерживает списки управления доступом (ACL), совместимые со стандартом POSIX, которые являются более гибкими по сравнению с привилегиями «владельца-группы-других», принятых в традиционных вариантах UNIX. Список ACL может быть ассоциирован с любым каталогом или файлом. Список представляет собой набор элементов, каждый из которых состоит из идентификатора пользователя или группы и набора прав, делегируемых этому пользователю или группе. К каждому объекту приложимы шесть типов полномочий: *чтение* (read), *запись* (write), *выполнение* (execute), *управление* (control), *добавление* (insert) и *удаление* (delete) [12]. Изменять ACL может пользователь, обладающий правом управления объектом. Разрешения на добавление и удаление относятся только к каталогам и позволяют осуществлять соответственно создание и удаление файлов в этом каталоге. Стандартный набор символов подстановки позволяет одному элементу списка ACL описывать привилегии сразу нескольких пользователей или групп. Например, для предоставления прав пользователю rohan на чтение, выполнение и добавление элементов в каталог в соответствующем вхождении журнала пишется:

```
user:rohan:rwx-i-
```

Каждый дефис указывает на отказ в предоставлении права. В приведенном примере rohan не обладает правами управления и удаления.

11.9. Процесс watchdog

При реализации файловой системы определяются некоторые правила, касающиеся таких тем, как именование, управление доступом и хранение. Эти правила относятся ко всем файлам системы. Часто возникает необходимость изменения соглашений, принятых по умолчанию, для некоторых файлов, к которым необходим более дифференцированный подход. Например, бывает нужно:

- позволить пользователям реализовывать различные механизмы управления доступом;
- отслеживать и записывать в журнал данные обо всех обращениях к определенному файлу;
- производить некие автоматизированные действия при поступлении почты;
- хранить файлы в сжатом виде, автоматически распаковывая их при чтении.

Перечисленными возможностями обладает расширение FFS, разработанное в Вашингтонском университете [1]. В его основу заложена идея ассоциации с файлом или каталогом процесса прикладного уровня, называемого *watchdog*, который производит перехват определенных файловых операций, предлагая собственную реализацию этих функций. Процессы watchdog не обладают какими-либо специальными привилегиями и выполняют свою работу полностью незаметно для приложений, осуществляющих доступ к файлу. При этом альтернативные действия над файлом осуществляются только при вызове операций, для которых назначено замещение.

Для ассоциации процесса-«цербера»[1] с файлом применяется системный вызов wdlink. Такой файл становится «охраняемым» (guarded file). В качестве входных аргументов вызова указываются имена файла и программы наблюдения за ним. Имя программы хранится в 20-байтовом поле индексного дескриптора файла (такое поле было зарезервировано в системе BSD UNIX для применения в будущем). Для того чтобы имя уложилось в 20 символов, в поле размещается ссылка на вхождение публичного каталога /wdogs, в котором хранятся символические ссылки на реальные имена программ watchdog.

Если какой-либо процесс пытается открыть охраняемый файл, ядро посылает процессу watchdog сообщение (загружая его в память, если это не было сделано ранее). Программа наблюдения может использовать собственные правила разрешения или запрета доступа, либо передать эти функции ядру системы. Если открытие файла разрешено, программа перечисляет ядру набор операций, которые необходимо произвести с охраняемым файлом. Набор операций может быть неодинаков для разных представлений файла.

После открытия файла каждый раз, когда пользователь вызывает операцию, являющуюся «подзащитной», ядро перенаправляет ее процессу watchdog (рис. 11.9). Программа watchdog осуществляет одно из трех перечисленных действий:

- разрешает выполнение операции. При этом возможен обмен дополнительными данными между ядром и процессом watchdog (например, при операциях read или write). Для предупреждения лишних циклов процессу-«опекуну» дается право на прямой доступ к охраняемому файлу;
- запрещает выполнение операции, при этом ядру передается код ошибки;
- узнает о факте операции. При этом не выполняется никаких дополнительных действий, процесс watchdog предоставляет ядру возможность обычного выполнения операции. До передачи управления ядру программа может предпринять некоторые действия, например, учет.

[1] Слово *watchdog* переводится с английского языка буквально как «сторожевой пес». — *Прим. перев.*

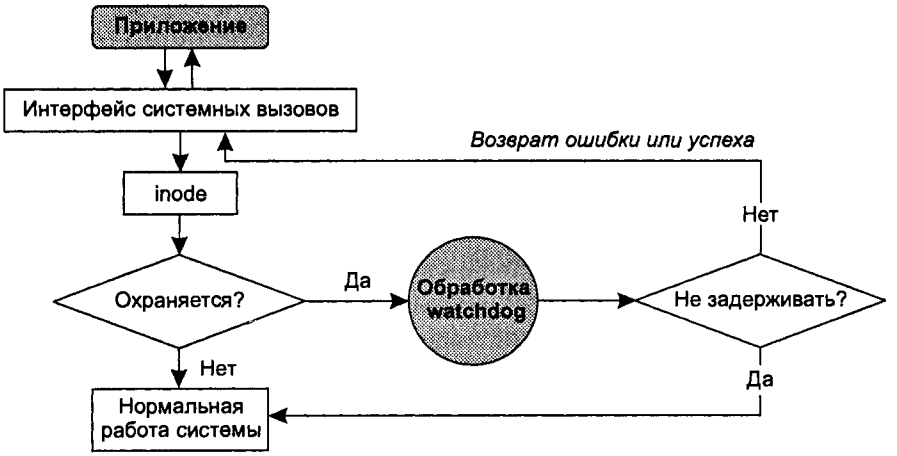


Рис. 11.9. Процесс watchdog

11.9.1. Наблюдение за каталогами

Процесс watchdog может быть ассоциирован не только с файлом, но и с каталогом. Такой процесс контролирует операции над каталогом и может быть использован для управления доступом ко всем файлам, находящимся внутри него (после чего управление осуществляется по отношению ко всем каталогам полного имени). Процесс наблюдения за каталогом обладает некоторыми дополнительными возможностями. По умолчанию он охраняет все файлы, находящиеся в каталоге и не связанные с собственными процессами watchdog.

Это свойство системы имеет несколько весьма интересных применений. Пользователи могут осуществлять доступ к несуществующим файлам охраняемого каталога. Если с ним ассоциирован процесс watchdog, он будет создавать некую иллюзию существования файлов. В этом случае ни одна из операций над файлом может не передаваться ядру.

11.9.2. Каналы сообщений

Взаимодействие между ядром и процессами наблюдения осуществляется при помощи передачи сообщений. С каждым процессом watchdog ассоциируется уникальный *канал сообщений WMC* (Watchdog Message Channel), создаваемый при помощи системного вызова createwmc. Вызов возвращает дескриптор файла, который затем может быть использован процессом watchdog для отправки и получения сообщений от ядра.

Каждое сообщение содержит поле типа и идентификатор сеанса. С каждой новой открытой копией файла ассоциируется уникальный сеанс взаимо-

связи с процессом watchdog. На рис. 11.10 показаны структуры данных, поддерживаемые ядром. Вхождение таблицы открытых файлов охраняемого файла ссылается на вхождение в глобальной таблице сеансов. Оно является точкой входа канала WMC со стороны ядра, содержащего очередь непрочитанных сообщений. WMC также указывает на процесс наблюдения.

Процесс-перехватчик осуществляет чтение сообщений и отправку ответов при помощи файлового дескриптора, возвращенного ему вызовом createwmc. Дескриптор указывает на вхождение WMC в таблице открытых файлов, в которой содержится ссылка на вход канала, относящийся к ядру. Таким образом, доступ к очереди сообщений возможен как для процесса watcdog, так и для ядра системы. Они оба вправе как передавать сообщения в очередь, так и забирать их оттуда.

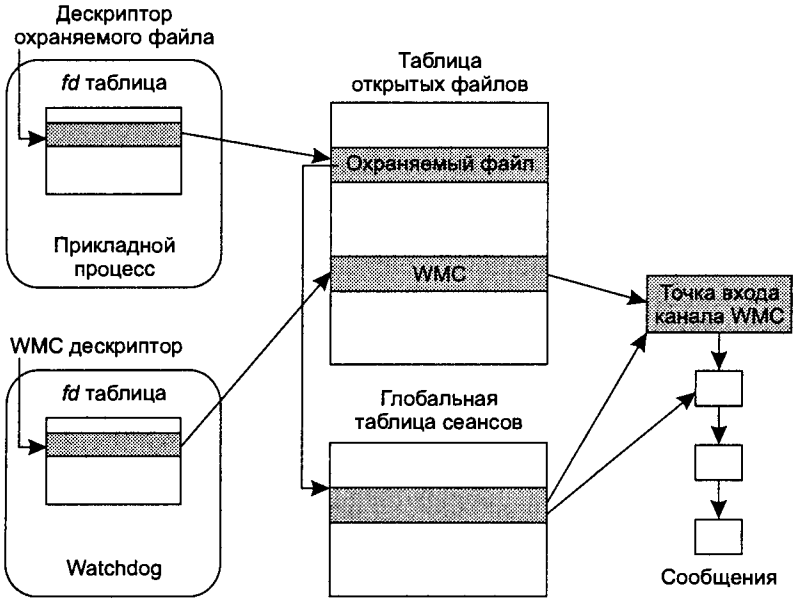


Рис. 11.10. Структуры данных процесса watchdog

Управление всеми процессами watchdog выполняется ведущим процессом (master). Он осуществляет контроль создания процессов watchdog (при открытии соответствующего охраняемого файла) и их завершения (обычно когда последние закрывают файлы). Головной процесс может сделать некоторые наиболее часто используемые процессы watchdog резидентными даже в том случае, если с ними не ассоциирован ни один открытый файл, что позволяет избавиться от задержек, возникающих при старте процесса.

11.9.3. Приложения

В оригинальной реализации системы описано несколько интересных приложений:

wdacl	Используется для ассоциации с файлом списка управления доступом. Единственный процесс watchdog может управлять доступом сразу ко многим файлам
wdcompact	Сжимает данные (и производит обратное преобразование) «на лету»
wdbiff	Просматривает пользовательский ящик электронной почты и уведомляет пользователя о прибытии новых писем. Программа может быть расширена до возможности автоответа и автоперенаправления
wdview	При помощи этой программы различным пользователям системы показывается отличное друг от друга представление одного и того же каталога
wddate	Позволяет пользователям читать данные о текущей дате и времени из файла. Сам по себе файл не содержит информации, программа watchdog производит чтение показаний системного таймера при каждом запросе файла

При помощи процессов-перехватчиков можно организовать графическое представление дерева каталогов. При создании или удалении файла пользователем процесс watchdog может запросить графический пользовательский интерфейс (GUI) о необходимости перерисовать «изображение» каталога с учетом внесенных в него изменений. Как показывает приведенный пример, процессы наблюдения являются универсальным инструментом, позволяющим расширить файловую систему самыми различными способами, количество которых ограничено только воображением разработчиков. Возможность переопределения отдельных функций операционной системы на прикладном уровне является весьма практичной и удобной. Средство watchdog является несомненным достоинством современных операционных систем.

11.10. Portal File System в 4.4BSD

Технология watchdog позволила процессам прикладного уровня перехватывать вызовы других процессов по отношению к «опекаемому» файлу. *Файловая система порталов* (portal file system) в 4.4BSD предоставляет аналогичные возможности. Система определяет пространство имен файлов, которые могут быть открыты процессами. Если какой-либо процесс пытается открыть файл в файловой системе порталов, ядро посылает сообщение демону portal, обрабатывающему запрос на открытие и возвращающему процессу дескриптор. Набор корректных имен файлов и их интерпретация обуславливаются не самой файловой системой, а демоном portal.

Демон создает сокет UNIX [22] и с целью разрешения поступления входящих запросов вызывает системную функцию listen на этот сокет. Затем он монтирует файловую систему порталов (обычно в каталоге /p). После этого демон выполняет цикл, в котором периодически вызывает функцию accept в ожидании запросов для их обработки после получения.

Рассмотрим ситуацию, когда пользователь открывает файл в системе порталов. В результате этого события происходит следующая последовательность действий (рис. 11.11).

1. Сначала ядро вызывает функцию namei() для анализа имени файла. При достижении функцией точки монтирования системы порталов происходит вызов portal_lookup() для разбора оставшейся части имени.
2. Процедура portal_lookup() размещает новую структуру vnode и сохраняет полное имя в закрытом объекте данных vnode.
3. Затем ядро инициирует операцию VOP_OPEN по отношению к vnode, в результате которой вызывается portal_open().
4. Операция portal_open() передает полное имя демону portal, которому возвращается управление после системного вызова accept.
5. Демон обрабатывает имя и создает файловый дескриптор.
6. Демон отправляет дескриптор обратно вызвавшему процессу через соединение сокетов, установленное ядром.
7. Ядро копирует дескриптор в первый неиспользованный элемент (slot) таблицы дескрипторов вызвавшего процесса.
8. Демон portal разрывает соединение и вызывает accept для перехода в ожидание нового запроса.

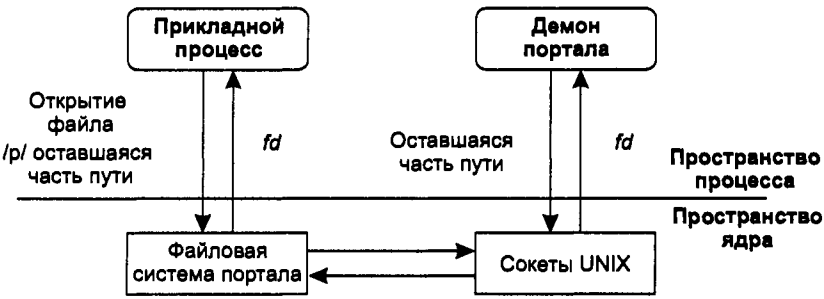


Рис. 11.11. Открытие файла в файловой системе порталов

11.10.1. Применение порталов

Файловая система порталов имеет несколько применений. Демон portal самостоятельно определяет собственные возможности и методику интерпретации пространства имен. Одним из важных приложений на основе этой системы

является сервер соединений. Сервер открывает сетевые соединения в интересах других процессов. Используя порталы, сервер может создать TCP-соединение путем простого открытия файла под названием

```
/p/tcp/node/service
```

где node — имя удаленной машины, с которой необходимо установить соединение, service указывает на службу TCP (например, ftp или rlogin). В качестве примера можно привести имя файла /p/tcp/archana/ftp, при открытии которого устанавливается FTP-соединение с узлом archana.

Демон производит все действия, необходимые для настройки сетевого соединения. Он определяет сетевой адрес удаленной машины, связывается с процессом portmap на этом узле с целью определения номера порта необходимой службы, создает сокет TCP и соединяется с сервером. Демон передает файловый дескриптор соединения обратно вызывающему процессу, который может использовать его в дальнейшем для взаимодействия с сервером.

Вышеописанная технология делает TCP-соединения доступными для *простейших приложений*. Такими приложениями называют задачи, использующие только stdin, stdout и stderr и не знающие о наличии каких-либо других устройств в системе. Например, пользователь может перенаправить вывод сценария shell или awk на удаленный узел путем открытия соответствующего файла портала.

Так же как и в технологии процессов watchdog, файловая система порталов позволяет прикладным процессам перехватывать файловые операции других процессов и предлагать свою собственную их реализацию. Однако между этими двумя технологиями существуют некоторые различия. Демон portal обрабатывает только системный вызов open, в то время как процесс watchdog способен перехватить любые выбранные им операции. С другой стороны, демон portal может определять пространство имен. Это стало доступным благодаря способности функции namei() 4.4BSD к передаче остатка имени операции portal_lookup() после достижения точки монтирования. Процессы watchdog обычно имеют дело с уже существующей файловой иерархией, хотя они и обладают ограниченной возможностью расширения пространства имен.

11.11. Уровни стековой файловой системы

Появление интерфейса vnode/vfs стало важнейшей вехой развития, позволяющей разрабатывать новые файловые системы для UNIX. Создание vnode/vfs привело к появлению множества файловых систем (о них вы можете прочесть в главах 9–11). Но несмотря на это интерфейс обладает несколькими ограничениями:

- интерфейс неоднороден в различных вариантах UNIX. Как уже описывалось в разделе 8.11, существует множество различий в наборах операций, определенных в ОС SVR4, 4.4BSD и OSF/1, а также в деталях их семантики. Например, операция VOP_LOOKUP в SVR4 просматривает за раз только один аргумент, в то время как в 4.4BSD она может работать одновременно с несколькими аргументами;
- даже в системах, созданных одним и тем же производителем, реализация интерфейса отличается друг от друга для различных версий. Например, в SunOS 4.0 была произведена замена операций буферного кэширования, таких как VOP_BMAP или VOP_BREAD, на страничные операции, например VOP_GETPAGE и VOP_PUTPAGE. В системе SVR4 были добавлены новые операции, такие как VOP_RWLOCK и VOP_RWUNLOCK, которые позже были встроены в SunOS;
- файловая система и подсистема управления памяти остаются сильно зависящими друг от друга. В результате становится невозможным создание файловой системы общей направленности без четкого понимания архитектуры управления памятью;
- интерфейс изначально не обладает свойством прозрачности. Ядро обращается ко многим полям vnode напрямую, не используя процедурный интерфейс. В результате изменение структуры vnode, поддерживающее обратную совместимость на бинарном уровне, является весьма сложной задачей;
- интерфейс не поддерживает *наследование*. Новая файловая система не может унаследовать какие-либо средства уже существующей файловой системы;
- интерфейс не обладает свойством расширяемости. В файловую систему невозможно добавлять новые функции либо изменять семантику уже существующих операций.

Перечисленные факторы являются препятствием в развитии технологий файловых систем. Разработка систем общего применения, таких как Episode (корпорации Transarc) или Veritas (от Veritas Corporation), подтвердила это. Для переноса файловой системы на другие варианты UNIX (или реализации новой версии файловой системы) требовался труд больших команд разработчиков. Более того, многие поставщики ОС не желали заниматься разработкой файловых систем с нуля. Чаще всего они добавляли новые средства, такие как репликация, шифрование или списки управления доступом, в уже существующие файловые системы.

Независимые исследования Калифорнийского университета (Лос-Анджелес) [4] и компании SunSoft [16], [20] привели к созданию интегрированных сред, основанных на уровнях наращиваемой файловой системы, ориентиро-

ванных в большой степени на поддержку развития модульных файловых систем. Реализация UCLA позже была встроена в 4.4BSD, решение SunSoft все еще находится на стадии создания модели. В этом разделе мы поговорим о важнейших возможностях наращиваемых файловых систем, а также рассмотрим несколько приложений, основанных на этой технологии.

11.11.1. Инфраструктура и интерфейс

Интерфейс vfs позволяет различным файловым системам сосуществовать на одной машине. Он определяет набор операций, реализация которых зависит от конкретной файловой системы. При вызове пользователем операции над файлом ядро динамически передает запрос той файловой системе, к которой этот файл относится. Ответственность за выполнение операции полностью возлагается на конкретную файловую систему.

Структура наращиваемых уровней обеспечивает монтирование различных файловых систем на основе друг друга. Каждый файл представляется стеком объектов vnode, где для каждой файловой системы в стеке ассоциируется один объект vnode. Когда пользователь вызывает файловую операцию, ядро передает ее самому верхнему vnode. Такой объект vnode вправе произвести одно из двух действий, для первого случая операция выполняется полностью с передачей результатов вызвавшему процессу. Альтернативным вариантом является проведение некоторых мероприятий, после чего операция передается следующему vnode стека. В этом случае операция может постепенно обойти все уровни стека. При завершении операции результат будет передан через все уровни в обратном направлении, давая возможность каждому объекту vnode произвести некоторые дополнительные действия.

Такой подход позволяет создавать наращиваемые (или инкрементные) файловые системы. Например, производитель может предложить модуль шифрования-дешифровки, рекомендуемый им для установки поверх любой физической файловой системы (рис. 11.12). Модуль будет перехватывать все операции ввода-вывода, производя шифрование данных при записи и их раскодирование при чтении. Все остальные операции будут передаваться на физический уровень без изменений.

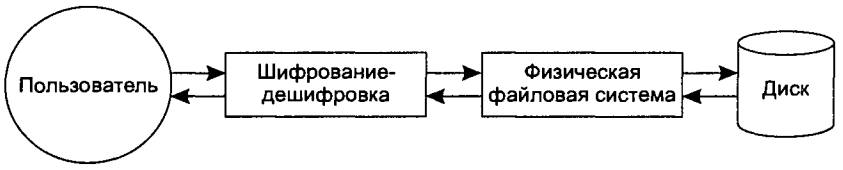


Рис. 11.12. Уровень шифрования-дешифровки

Технология наращивания предоставляет возможность создавать конфигурации, разветвляющиеся по входу или выходу. Стек, *разветвляющийся по*

входу (fan-in), позволяет нескольким более высоким уровням использовать один и тот же более низкий уровень стека. Например, уровень сжатия может производить сжатие данных при записи и распаковку при чтении, а программе резервного копирования необходимо иметь возможность копирования упакованных данных на носитель непосредственно в сжатом виде. В результате выполняется операция разветвления по входу, схематично изображенная на рис. 11.13.

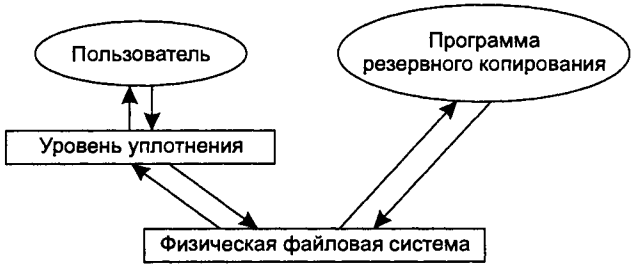


Рис. 11.13. Схема разветвления по входу

Стеки разветвления по выходу позволяют верхнему уровню управлять несколькими более низкими уровнями. Такая возможность может быть использована менеджером иерархического хранения (hierarchical storage manager, HSM). Менеджер HSM хранит недавно использованные файлы на локальных дисках и переносит давно не использовавшиеся файлы на оптические диски или стримеры (рис. 11.14). На уровне HSM производится перехват всех операций доступа к файлам с целью не только отслеживания использования файлов, но и загрузки их с третичных носителей, если это необходимо. В работе [24] представлена реализация HSM на основе инфраструктуры, комбинирующей возможности наращиваемых уровней и технологии watchdog.

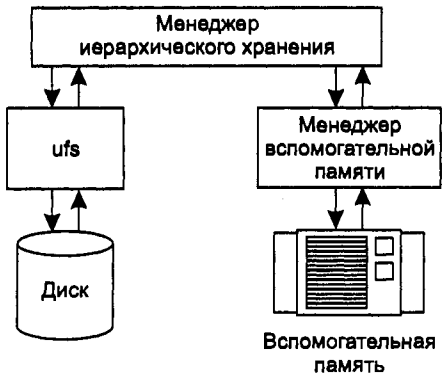


Рис. 11.14. Менеджер иерархического хранения

11.11.2. Модель SunSoft

В публикации [16] описывается работа по реализации стекового интерфейса vnode корпорации Sun Microsystems. Позднее работа над проектом была передана Международной рабочей группе по наращиваемым файлам UNIX (UNIX International Stackable Files Working Group) и впоследствии была продолжена [17] SunSoft. Прототип SunSoft [20] снял многие проблемы и ограничения интерфейса [16].

В этой реализации объект vnode содержит лишь указатель на связанный список объектов pvnode, по одному на каждую файловую систему, находящуюся на определенном уровне (рис. 11.15). Структура pvnode содержит указатели на vfs, вектор vnodeops и собственные закрытые данные. Каждая операция сначала передается pvnode верхнего уровня, далее она может быть перенаправлена на более низкий уровень (если это необходимо).

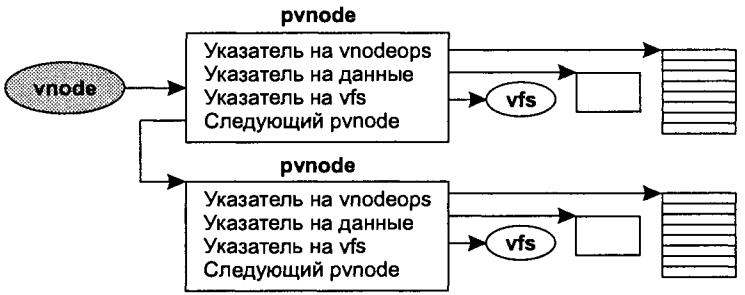


Рис. 11.15. Прототип SunSoft

Разработчики прототипа руководствовались следующими важными соображениями:

- в текущем интерфейсе структура vnode содержит некоторое количество полей данных, доступ к которым осуществляется ядром напрямую. Эти поля должны быть перемещены в объект закрытых данных. Чтение и запись таких данных должны осуществляться через процедурный интерфейс;
- структура vnode должна содержать ссылку на другую структуру vnode как элемент закрытых данных. Например, vnode корневого каталога файловой системы указывает на точку монтирования;
- операции vfs должны передаваться на более низкие уровни стека. Для этого многие операции vfs необходимо преобразовать в операции vnode, после чего они могут использоваться любым объектом vnode файловой системы;
- многие операции текущего интерфейса взаимодействуют с несколькими объектами vnode. Для корректной работы такие операции следует разделить на несколько мелких операций так, чтобы каждая из них

обрабатывала только один vnode. Например, операцию VOP_LINK целесообразно раздробить на две, одна из которых будет производить перенос идентификатора файла и инкрементирование счетчика ссылок (используя vnode файла), а вторая будет отвечать за добавление вхождения файла в vnode каталога;

- для гарантии неделимости дробных операций, вызванных одной и той же операцией высокого уровня, необходимо средство транзакций;
- пространство имен <vnode, offset> для страниц кэша не совсем приемлемо в отношении наращиваемого интерфейса vnode, так как в нем страница может относиться сразу к нескольким объектам vnode. Соответственно, требуется переработать интерфейс взаимодействия с подсистемой виртуальной памяти.

11.12. Интерфейс файловой системы 4.4BSD

Интерфейс файловой системы в 4.4BSD опирается на реализацию наращиваемых уровней файловой системы Ficus, разрабатываемой в UCLA [4]. Другие возможности файловой системы 4.4BSD описывались ранее в разделах 8.11.2 и 11.10. Здесь мы остановимся на той части интерфейса, которая относится к стекам, и опишем некоторые интересные варианты файловых систем на его основе.

В 4.4BSD для помещения уровня файловой системы в стек vnode применяется системный вызов mount. Для извлечения его из стека используется вызов unmount. Точно так же, как и в модели SunSoft, каждая операция должна быть сначала помещена на самый верхний уровень стека. Любой из уровней может как завершить выполнение операции и возвратить ее результаты, так и столкнуть ее для обработки на более низкий уровень (производя при этом некоторые дополнительные действия при необходимости).

В 4.4BSD имеется возможность присоединения уровня файловой системы сразу к нескольким вхождениям пространства имен. Это позволяет обращаться к одному и тому же файлу, используя несколько различных полных имен (не прибегая к отдельным ссылкам). Более того, остальные уровни стека могут быть неодинаковыми для каждой точки монтирования, в результате чего у одной и той же операции может быть различная семантика. Такой подход совместим с технологией разветвления по входу (см. раздел 11.11.1). Например, файловая система может быть одновременно монтирована в каталогах /direct и /compress, при этом каталог /compress соответствует уровню сжатия стека. Такая конфигурация системы позволяет производить шифрование-дешифровку «на лету» посредством запроса файлов через /compress. Однако программа резервного копирования вправе при этом обращаться к файлам через /direct напрямую, обходя операцию их разуплотнения.

Файловые системы могут добавлять в интерфейс новые операции. При начальной загрузке операционной системы ядро динамически перестраивает вектор vnodeops как объединение операций, поддерживаемых каждой файловой системой. Для практической реализации этой возможности все файловые системы должны поддерживать стандартную функцию bypass с целью обработки неизвестных им операций. Эта функция производит проталкивание операции и ее аргументов на следующий уровень стека. Функция bypass не обладает информацией о количестве и типе аргументов операции, поэтому система 4.4BSD без огладки на нее упаковывает аргументы операции vnode в некую структуру. Затем она возвращает указатель на эту структуру в качестве единственного параметра для всех операций. Если уровень не способен распознать операцию, то он просто передает указатель следующему уровню. Если же уровень обладает информацией о том, как поступить для этой операции, он также знает, как требуется интерпретировать структуру ее аргументов.

В следующем разделе кратко описываются два представляющих интерес приложения интерфейса наращиваемой файловой системы.

11.12.1. Файловые системы nullfs и union mount

Файловая система nullfs [10] является по большей части промежуточной системой, позволяющей монтировать большие поддеревья файловой иерархии в какое-либо другое место файловой системы. В результате получается, что для каждого файла в поддереве поддерживается два полных имени. Большинство операций передается оригинальной файловой системе. Такая возможность имеет несколько интересных применений. Например, пользователь, смонтировавший поддеревья каталогов, расположенных в различных файловых системах, может присоединить их к общему каталогу. После этого он будет видеть единое дерево, содержащее все файлы и каталоги.

Файловая система union mount [14] предоставляет возможности, сходные с представленными в Translucent File System (TFS), описываемой в разделе 9.11.4. Система дает возможность объединить (или слить) в единое целое файловые системы, подключенные к ней. Самый верхний уровень стека является логически наиболее свежим. В системе именно он единственный доступен для записи. Когда пользователь производит поиск файла, ядро проходит вниз от одного уровня к другому до тех пор, пока не обнаружит уровень, содержащий необходимый файл. Если пользовать пытается внести в этот файл изменения, ядро сначала производит его копирование на самый верхний уровень. Точно так же, как и в TFS, если пользователь удаляет файл, ядро создает вхождение whiteout в верхнем уровне, что позволяет предупредить поиск этого файла на более низких уровнях при последующих операциях просмотра. Для передачи и удаления вхождений whiteout в системе предусмотрены специальные операции, которые позволяют восстановить случайно «стертые» пользователем файлы.

11.13. Заключение

В этой главе были рассмотрены некоторые усовершенствованные файловые системы. Некоторые из них заменили уже существующие реализации, такие как FFS или s5fs, в то время как другие расширили функциональность традиционных файловых систем в различных направлениях. Такие системы обладают более высокой производительностью, способностью к быстрому восстановлению после отказа, увеличенной надежностью либо дополнительными возможностями. Некоторые из описанных систем уже получили коммерческое признание, в большинстве новых реализаций UNIX представлен тот или иной вариант файловой системы, использующей технологию ведения журналов.

Появление интерфейса vnode/vfs стало важным шагом, оказавшим положительное влияние на дальнейшее развитие технологий. Именно он позволил интегрировать новые реализации файловых систем в ядро UNIX. Появление технологии наращиваемых уровней помогло избавиться от многих ограничений, присущих интерфейсу vnode/vfs, и стало толчком к развитию инкрементных файловых систем. Поддержка технологии уже реализована в операционной системе 4.4BSD, многие разработчики коммерческих ОС пребывают в стадии ее признания[1].

---

[1] Многократное упоминание автором имевшихся на то время разработок принуждает сказать, что описанные системы потерпели неудачу вследствие недостаточной переносимости и устойчивости. Они оказались не готовы к портированию файловых систем на другие ОС и встретили трудности при адаптации системного кода. В теории стековых файловых систем было много, но на практике работала только горстка: Ficus UCLA (все связанные публикации можно посмотреть в электронной библиотеке Association for Computing Machinery (ACM) — http://portal.acm.org), lofs (loopback virtual file system, SunOS 5.5.1, 1992) и 4.4BSD-Lite (1995, основа будущей FreeBSD 3.0). Но на их ошибках учились создатели других операционных систем, предложившие новые интерфейсы наращиваемых vnode. Это Herd of Unix-Replacing Daemons (HURD) от Free Software Foundation (FSF), представляющая собой набор серверов, выполняющийся под управлением микроядра Mach 3.0 с файловой системой прикладного уровня. В ней была представлена концепция транслятора, программы, предоставляющей сервис доступа к полному имени (открытие файла, просмотр имен, создание каталогов и др.). Разработки Sun вылились в создание объектно-ориентированной экспериментальной ОС Spring (1994), являющейся набором взаимодействующих серверов на вершине микроядра (кэширование, ввод-вывод, отображение в память, именование объектов, защита и др.). Из Spring выросла Solaris MC (Multi-Computer) File System (1996), наследовавшая от Spring объектно-ориентированный подход и интегрированная в интерфейс vnode Solaris, обеспечивающая функции распределенной файловой инфраструктуры на основе специальной Proxy File System. Она была разработана для работы в закрытой среде, но не в сетевом окружении, так как требовала высокой производительности сети и хостов, в результате не была поддержана в дальнейшем. В 1999 г. исследователи из Колумбийского университета, Калифорния, представили и сопроводили результатами тестов образец файловой системы, названной Wrapfs (wrapper file system) (www.cs.columbia.edu/~ezk/research/wrapfs). Wraps — система, переносимая на широкий диапазон операционных и файловых систем, в первую очередь Solaris, Linux и FreeBSD. Это достигается тем, что интерфейс vnode использует указатели на закрытые данные для каждой структуры интерфейса и монтирует самого себя в корне существующей файловой иерархии, действуя как посредник между точкой монтирования со стороны пользователя и нижним уровнем монтируемой файловой системы. Эта система вобрала в себя лучшее из всех былых прототипов, и другие разработчики «пребывают в стадии ее признания». — *Прим. ред.*

11.14. Упражнения

1. Каким образом в системе FFS используется фактор задержки при чередовании дисковых блоков? Как это влияет на схему использования и размеры дискового кэша?
2. Как влияет на производительность сервера NFS кластеризация файловой системы?
3. В чем разница между кластеризацией системы и технологией сбора записей (см. раздел 10.7.3)? Опишите ситуации, в которых применение той или иной технологии является более выигрышным. Какую выгоду дает их объединение?
4. Каким образом кластеризация файлов сдерживает преимущества применения энергонезависимой памяти? Опишите положительные примеры применения NVRAM в системах, поддерживающих кластеризацию.
5. Каковы положительные стороны задержки операции записи на диск?
6. Представьте, что файловая система производит запись всех метаданных синхронно, откладывая запись данных на время выполнения демона update. К возникновению каких проблем в защите системы это может привести? Какие блоки данных необходимо записывать синхронно для предупреждения этого?
7. Приведет ли к увеличению производительности структурированной файловой системы сбор мусора во время малой занятости системы? Какие ограничения на использование файловой системы наложит такая процедура?
8. Файловая система может использовать битовые карты, находящиеся в памяти, с целью отслеживания активных и устаревших участков журнала. Проведите анализ, несколько подходит такая возможность для структурированных файловых систем и систем сбора метаданных.
9. Представьте, что файловая система сбора метаданных записывает изменения индексных дескрипторов и блоков каталогов в журнал, при этом запись блоков косвенной связи производится синхронно. Опишите ситуацию, при которой данный подход может привести к некорректному состоянию системы вследствие отказа.
10. Рационально ли хранить модификации всех файловых систем в системах сбора метаданных в едином журнале? Какие преимущества и недостатки вы видите у этой методики?
11. Обладает ли какими-либо преимуществами хранение журнала метаданных на том же физическом диске, где расположена сама файловая система?
12. Почему в файловой системе Episode применяется двухступенчатый протокол блокировки? Объясните, почему он не требуется для файловой системы Cedar.

13. Предположим, что какой-либо пользователь желает вести запись всех обращений к его файлам со стороны других пользователей. Может ли он реализовать эту операцию, используя либо технологию процессов watchdog, либо технологию порталов? Как это повлияет на поведение системы?
14. Представьте, что на диске содержатся каталоги системы union mount. Что произойдет в том случае, если на диске закончится свободное пространство и пользователь с целью его освобождения удалит набор файлов, находящихся на более низком уровне?

11.15. Дополнительная литература

1. Bershad, B. N., and Pinkerton, C. B., «Watchdogs — Extending the UNIX File System», Computing Systems, Vol. 1, No. 2, Spring 1988, pp. 169–188.
2. Chutani, S., Anderson, O. T., Kazar, M. L., Mason, W. A., and Sidebotham, R. N., «The Episode File System», Proceedings of the Winter 1992 USENIX Technical Conference, Jan. 1992, pp. 43–59.
3. Hagmann, R., «Reimplementing the Cedar File System Using Logging and Group Commit», Proceedings of the 11th Symposium on Operating Systems Principles, Nov. 1987, pp. 155–162.
4. Heidemann, J. S., and Popek, G. J., «File-System Development with Stackable Layers», ACM Transactions on Computer Systems, Vol. 12, No. 1, Feb. 1994, pp. 58–89.
5. Hitz, D., Lau, J., and Malcolm, M., «File System Design for an NFS File Server Appliance», Proceedings of the Winter 1994 USENIX Technical Conference, Jan. 1994, pp. 235–245.
6. Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., and Sidebotham, R. N., «Scale and Performance in a Distributed File System», ACM Transactions on Computer Systems, Vol. 6, No. 1, Feb. 1988, pp. 55–81.
7. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3 BSD UNIX Operating System», Addison-Wesley, Reading, MA, 1989.
8. Mashey, J. R., «UNIX Leverage — Past, Present, Future», Proceedings of the Winter 1987 USENIX Technical Conference, Jan. 1987, pp. 1–8.
9. McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S., «A Fast File System for UNIX», ACM Transactions on Computer Systems, Vol. 2, No. 3, Aug. 1984, pp. 181–197.
10. McKusick, M. K., «The Virtual Filesystem Interface in 4.4BSD», Computing Systems, Vol. 8, No. 1, Winter 1995, pp. 3–25.
11. McVoy, L. W., and Kleiman, S. R., «Extent-like Performance from a UNIX File System», Proceedings of the 1991 Winter USENIX Conference, Jan. 1991, pp. 33–43.

12. Open Software Foundation, «OSF DCE Administration Guide — Extended Services», Prentice-Hall, Englewood Cliffs, NJ, 1993.
13. Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G., «A Trace-Driven Analysis of the UNIX 4.2 BSD File System», Proceedings of the 10th Symposium on Operating System Principles, Dec. 1985, pp. 15–24.
14. Pendry, J. S., and McKusick, M. K., «Union Mounts in 4.4BSD-Lite», Proceedings of the Winter 1995 USENIX Technical Conference, Jan. 1995, pp. 25–33.
15. Rosenblum, M, and Ousterhout, J. K., «The LFS Storage Manager», Proceedings of the Summer 1990 USENIX Technical Conference, Jun. 1990, pp. 315–324.
16. Rosenthal, D. S. H., «Evolving the Vnode Interface», Proceedings of the Summer 1990 USENIX Technical Conference, Jun. 1990, pp. 107–118.
17. Rosenthal, D. S. H., «Requirements for a „Stacking" Vnode/VFS Interface», UNIX Internationa] Document SF–01–92–N014, Parsippany, NJ, 1992.
18. Seltzer, M., Bostic, K., McKusick, M. K., and Staelin, C., «An Implementation of a Log-Structured File System for UNIX», Proceedings of the Winter 19993 USENIX Technical Conference, Jan. 1993, pp. 307–326.
19. Seltzer, M., and Smith, K. A., «File System Logging Versus Clustering: A Performance Comparison», Proceedings of the Winter 1995 USENIX Technical Conference, Jan. 1995, pp. 249–264.
20. Skinner, G. C., and Wong, T. K., «Stacking Vnodes: A Progress Report», Proceedings of the Summer 1993 USENIX Technical Conference, Jun. 1993, pp. 161–174.
21. Staelin, C., «Smart Filesystems», Proceedings of the Winter I991 USENIX Conference, Jan. 1991, pp. 45–51.
22. Stevens, W. R., and Pendry, J. S., «Portals in 4.4BSD», Proceedings of the Winter 1995 USENIX Technical Conference, Jan. 1995, pp. 1–10.
23. Vahalia, U., Gray, C., and Ting, D., «Metadata Logging in an NFS Server», Proceedings of the Winter 1995 USENIX Technical Conference, Jan. 1995, pp. 265–276.
24. Webber, N., «Operating System Support for Portable Filesystem Estensions», Proceedings of the Winter 1993 USENIX Technical Conference, Jan. 1993, pp. 219–228.
25. Wittle, M., and Keith, B., «LADDIS: The Next Generation in NFS File Server Benchmarking», Proceedings of the Summer 1993 USENIX Technical Conference, Jun. 1993, pp. 111–128.
26. Yager, T., «The Great Little File System», «Byte», Feb, 1995, pp. 155-158.

Глава 12
Выделение памяти ядром

12.1. Введение

Операционная система управляет всей доступной физической памятью машины и производит ее выделение для остальных подсистем ядра и прикладных задач. При изначальной загрузке системы ядро резервирует часть оперативной памяти под собственные коды и неизменяемые структуры данных. Эта часть ядра никогда не выгружается из памяти и не может быть использована для каких-то других целей[1]. Оставшейся памятью системы ядро управляет динамически, выделяя ее для различных клиентов (процессов и подсистем ядра). Такая память обычно освобождается при необходимости.

В UNIX оперативная память системы делится на порции фиксированного размера, также называемые *страницами*. Размер страницы памяти является некоторой степенью числа 2, в большинстве случаев ее объем составляет 4 Кбайт. UNIX является системой виртуальной памяти, поэтому логически последовательные страницы памяти не всегда имеют тот же порядок размещения в физической памяти компьютера. Следующие три главы книги посвящены описанию виртуальной памяти. Подсистема управления памятью отвечает за соответствие логических (или виртуальных) страниц с реальным расположением данных в физической памяти. В ответ на запрос о выделении блока логически смежных страниц памяти подсистема может предоставить несколько физически непоследовательных страниц.

Такой подход упрощает задачу выделения памяти. С этой целью ядро поддерживает связанный список свободных страниц. Если процессу необходимо получить некоторое количество страниц, ядро отберет их из числа свободных. При освобождении страниц ядро снова возвратит их в список. При этом фактическое расположение страниц в памяти компьютера не важно. Работа с памятью на уровне страниц реализована в системе 4.3BSD путем применения процедур memall() и memfree(), в системе SVR4 для этой цели применяются процедуры get_page() и freepage().

---

[1] Во многих современных системах UNIX (например, в AIX) часть ядра может находиться в страничной памяти.

Распределитель памяти страничного уровня (page-level allocator) имеет два принципиально важных клиента (см. рис. 12.1). Один из них называется *страничной подсистемой* (paging system) и представляет собой часть виртуальной системы памяти. Он производит выделение памяти для прикладных процессов, размещая страницы в их адресном пространстве. Во многих системах UNIX страничная система также управляет страницами, используемыми под дисковые буферы блоков. Другим клиентом является *распределитель памяти ядра* (kernel memory allocator), предоставляющий буферы памяти разного размера для множества подсистем ядра. Чаще всего ядру необходимы области памяти различной длины на короткие промежутки времени.

Ниже перечислены несколько задач, использующих память ядра.

- Процедура преобразования полного имени может запрашивать буфер (обычно размером 1024 байта) в целью копирования полного имени из пользовательского адресного пространства.
- Процедура allocb() выделяет буферы STREAMS произвольных размеров.
- Во многих реализациях системы UNIX в памяти размещаются структуры zombie, в которых находится информация о статусе выхода и использовании ресурсов завершенных процессов.
- В SVR4 ядро инициирует многие объекты (структуры proc, объекты vnode, блоки дескрипторов файлов и т. д.) динамически по мере необходимости.

Большинство таких запросов требуют памяти в меньшем объеме, чем размер одной страницы, следовательно, применение распределителя памяти ядра для таких задач является неподходящим. Для выделения памяти более мелкими порциями необходимо использовать другой механизм. Простейшим решением проблемы является запрещение динамического выделения памяти [1]. В ранних реализациях UNIX использовались таблицы фиксированного размера для объектов vnode, структур proc и т. д. Если в таких системах запрашивалась память для временного хранения полных имен или сетевых сообщений, ядро выделяло для них буферы из буферного кэша. Кроме этого, в отдельных ситуациях применялись *непредусмотренные изначально* (ad hoc) схемы выделения памяти, например в драйверах терминалов использовались *clists*.

Описанный подход имеет несколько недостатков. Технология является абсолютно негибкой, так как размеры всех таблиц и кэшей определяются на момент загрузки системы (чаще всего даже на стадии компиляции системы) и не могут быть изменены при необходимости. Размеры таких таблиц, принятые по умолчанию, выбираются разработчиками систем на основе анализа их использования в обычных рабочих группах. Хотя системные администраторы, как правило, обладают возможностью настройки размеров таблиц, чаще всего такие действия реализуются методом проб и ошибок. Если задать слишком маленький размер таблиц, это приведет к их переполнению и воз-

можному отказу системы без предупреждения. Если размер таблиц слишком велик, теряются впустую большие объемы памяти, а приложениям становится доступно меньшее ее количество. Это приводит к снижению общей производительности системы.

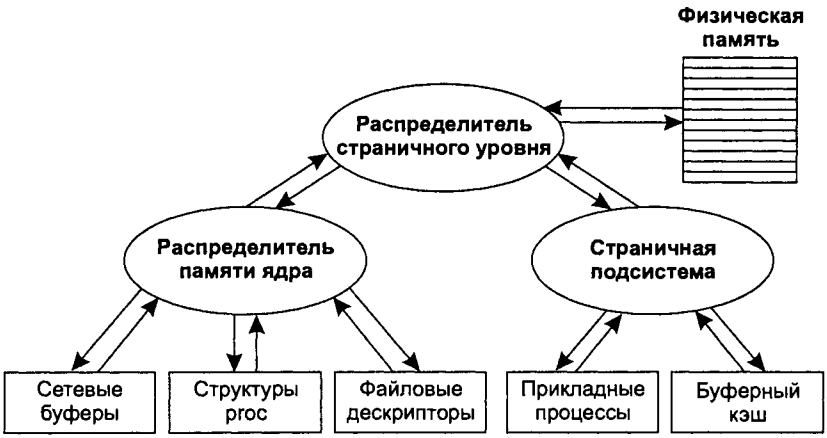


Рис. 12.1. Распределители памяти в ядре системы

Становится очевидным, что ядру необходимо общецелевое средство выделения памяти, которое умеет эффективно обрабатывать как объемные, так и небольшие области памяти. В следующем разделе будут кратко описаны требования, предъявляемые к распределителю памяти, а также критерии, по которым можно судить о его различных реализациях. Затем вы увидите описание и анализ некоторых распределителей памяти, применяемых в современных версиях UNIX.

12.2. Требования к функциональности

Распределитель памяти ядра (kernel memory allocator или KMA) обслуживает запросы на выделение динамической памяти от различных клиентов, таких как анализатор полных имен, STREAMS или средства взаимодействия процессов. KMA не занимается запросами на обслуживание страничной памятью прикладных процессов, за это отвечает страничная подсистема.

При загрузке системы ядро сначала производит резервирование памяти для собственных кодов и статических структур данных, а также некоторых встроенных областей, таких как буферный кэш. Распределитель страничного уровня осуществляет управление остальной частью физической памяти, которая может быть использована для ее динамического выделения в результате как запросов ядра, так и прикладных процессов.

Распределитель страничного уровня изначально запрашивает часть пространства у KMA, который должен эффективно использовать этот пул (pool) памяти. Некоторые реализации не позволяют изменять общее количество памяти, зарезервированное KMA. Другие разрешают распределителю памяти ядра забирать дополнительные объемы памяти у страничной подсистемы. В некоторых системах допускается даже двунаправленный обмен, разрешающий KMA передавать обратно освободившуюся память, ранее запрошенную у страничной подсистемы.

Если распределитель KMA сталкивается с отсутствием свободной памяти, то он блокирует вызывающий процесс до тех пор, пока некоторый объем памяти не освободится. Вызывающий процесс может передавать KMA флаг, указывающий на необходимость возврата с ошибкой (обычно NULL) вместо блокировки. Такая операция чаще всего применяется обработчиками прерываний, которые могут предпринять некоторые корректирующие действия при неудачном выполнении запроса. Например, если сетевое прерывание не может получить память для хранения входящего пакета, оно вправе просто пропустить такой пакет, надеясь на то, что отправитель повторно перешлет его позже.

Распределитель памяти ядра должен отслеживать, какие части пула в данный момент заняты или свободны. Освободившаяся часть памяти должна быть доступна для других запросов. В идеале обработка запроса на выделение памяти может не завершиться успешно только в том случае, если память действительно вся занята, то есть только когда общее количество свободной памяти, доступной распределителю, меньше, чем объем, указанный в запросе. В реальности сбой выделения происходит чаще в результате *фрагментации* памяти: даже если ее общего количества достаточно для удовлетворения запроса, доступная память чаще всего представляет собой непоследовательные фрагменты.

12.2.1. Критерии оценки

Важным критерием оценки распределителя памяти является возможность минимизировать ее потери. Объем физической памяти ограничен, следовательно, распределитель должен уметь эффективно использовать доступное пространство. Одной из величин, измеряющих эффективность, является *фактор использования* (utilization factor), являющийся соотношением общего объема запрашиваемой памяти к объему, позволяющему удовлетворить эти запросы. Идеальный распределитель памяти будет иметь 100%-ное использование, однако на практике приемлемой величиной является 50% [9]. Основная причина потерь во фрагментации памяти такова: обычно свободная память разбита на множество участков, которые слишком малы, чтобы быть востребуемыми. Распределитель убавляет фрагментацию путем слияния смежных участков свободной памяти в единую порцию.

Распределитель KMA должен быть быстрым, так как он интенсивно используется различными подсистемами ядра (в том числе обработчиками прерываний, чья производительность критична для системы в целом). Также важны показатели среднестатистической и максимальной задержки. Стеки ядра невелики по объему, поэтому оно прибегает к динамическому размещению в тех случаях, когда обычный процесс просто запрашивает объект из собственного стека. Такой подход требует высокой скорости выделения стека ядра. Медленно работающий распределитель оказывает отрицательное влияние на производительность системы в целом.

Распределитель должен обладать простым программным интерфейсом, подходящим под нужды различных клиентов. Одним из подходов является реализация интерфейса в виде функций, сходных с malloc() и free(), вызываемых прикладными процессам и предлагаемых в стандартной библиотеке системы:

```
void* malloc (size_t nbytes);
void free (void *ptr);
```

Важным преимуществом такого интерфейса является то, что при вызове процедуры free() не нужно знать размеры освобождаемого сегмента памяти. Часто одна из функций ядра запрашивает некоторое количество памяти и передает его другой подсистеме, которая позже и осуществляет ее освобождение. Например, сетевой драйвер может запросить буфер для хранения входящего сообщения и направить его модулю более высокого уровня для обработки данных и последующего освобождения буфера. Обладание информацией о размере запрашиваемого объекта для такого модуля не принципиально. Если задачу слежения за подобной информацией будет выполнять KMA, это сильно упростит работу, выполняемую его клиентами.

Еще одним полезным средством является разрешение на освобождение лишь части выделенной клиенту ранее области памяти. Если клиенту нужно освободить лишь часть занимаемой памяти, распределитель должен уметь корректно обрабатывать такую ситуацию. Интерфейс malloc()/free() не поддерживает эту возможность. Процедура free() освобождает область целиком и возвращает ошибку, если при ее вызове указать адрес, не совпадающий с указанным malloc(). Возможность увеличения буфера прикладных процессов (например, путем применения функции realloc()) также является весьма полезной.

Выделенная для использования память должна быть соответствующим образом упорядочена для быстрого обращения к ней. Для многих реализаций архитектуры RISC это является обязательным требованием. В большинстве систем достаточно упорядочивания в виде *длинных слов* (longword), однако в 64-разрядных машинах, таких как DEC Alpha AXP [6], может потребоваться выравнивание по 8-разрядной границе. Еще одним параметром, связанным с вопросами выделения областей памяти, является минимальный ее размер, который обычно равняется 16 или 8 байтам.

Многие коммерческие продукты используют память в круговом режиме. Например, машина в течение дня обслуживает запросы к базе данных и обработку транзакций, а по ночам занимается резервным копированием и реорганизацией структуры базы. Перечисленные задачи будут иметь совершенно разные требования к памяти. Обработка транзакций, скорее всего, потребует нескольких небольших участков памяти ядра для реализации блокировки базы данных, в то время как операция резервного копирования будет нуждаться в максимальном количестве свободной памяти для своих действий.

Многие распределители памяти разбивают доступный им пул на отдельные участки или *сегменты* (buckets) для запросов различных типов. Например, сегменты одного типа могут быть только 16-битовыми, в то время как другого — 64-битовыми. Такие распределители должны уметь противостоять неравномерному или круговому использованию, описанному выше. В некоторых распределителях после того, как вся память была отдана одному сегменту, она не может быть использована в дальнейшем для запросов областей иных размеров. Это чревато дисбалансом — большим объемом неиспользуемой памяти в одном сегменте и недостаточностью ресурсов памяти для других сегментов системы. Распределитель памяти должен обладать функцией динамического переприсвоения ее участков от одного сегмента другому.

Еще одним важным критерием является взаимодействие KMA со страничной подсистемой. Распределителю нужно уметь забирать часть страничной памяти при исчерпании первоначального объема. Страничная система должна обладать возможностью восстановления неиспользуемой памяти KMA. Такой обмен между двумя подсистемами требуется координировать соответствующим образом, чтобы предупредить устаревание каждой из подсистем.

Рассмотрим несколько методик выделения памяти и проведем их анализ, исходя из критериев оценки, перечисленных в этом разделе.

12.3. Распределитель карты ресурсов

Карта ресурсов (resource map) — это набор пар <base, size> (<базовый адрес, размер>), используемый для отслеживания свободных областей памяти (см. рис. 12.2). Изначально область памяти описывается при помощи единственного вхождения карты, в котором указатель равен стартовому адресу области, а размер равен ее общему объему памяти (рис. 12.2, *а*). После этого клиенты начинают запрашивать и освобождать участки памяти, вследствие чего область становится фрагментированной. Ядро создает для каждого нового последовательного свободного участка памяти новое вхождение карты. Элементы карты сортируются в порядке возрастания адресов, что упрощает задачу слияния свободных участков.

При помощи карты ресурсов ядро может выполнить запросы на выделение памяти исходя из трех правил.

- **Первый подходящий участок.** Выделение памяти из первой по счету свободной области, имеющей достаточный для удовлетворения запроса объем. Это самый быстрый алгоритм из всех трех, но он не совсем оптимален применительно к соображениям уменьшения фрагментации.
- **Наиболее подходящий участок.** Выделение памяти из области наименьшего размера, достаточного для удовлетворения запроса. Основным недостатком метода является необходимость иногда пропускать некоторое количество сегментов, которые являются слишком мелкими.
- **Наименее подходящий участок.** Выделение области максимального размера до тех пор, пока не будет найден более подходящий вариант. Такой подход кажется не совсем логичным, однако его применение основано на ожидании, что выделенная область будет достаточно велика для использования в последующих запросах.

Ни один из приведенных способов не идеален для всех возможных случаев. В книге [8] предлагается детальный анализ этих, а также других механизмов выделения памяти. В системах UNIX применяется методика первого подходящего участка.

На рис. 12.2 показан простейший пример карты ресурсов, обслуживающий 1024-байтовую область памяти. Область поддерживает две операции:

```
offset_t rmalloc (size); /* возвращает смещение выделенного участка */
void rmfree (base, size);
```

Изначально вся область памяти свободна и описывается единственным вхождением карты ресурсов (рис. 12.2, *а*). Для примера мы произведем два запроса на выделение участков памяти размером 256 и 320 байтов соответственно. После этого мы освободим 128 байтов, начиная со смещения 256. На рис. 12.2, *б* показано состояние карты после завершения перечисленных операций. В этот момент времени карта содержит два свободных участка и, следовательно, два вхождения для их описания.

На следующем этапе происходит освобождение еще 128 байтов, начиная от смещения 128. Распределитель памяти определяет, что этот участок размещен последовательно с другим свободным участком, начинающимся с отметки 256. В этом случае распределитель объединяет их в единый сегмент размером 256 байтов. В результате карта ресурсов выглядит как показано на рис. 12.2, *в*. Последний рисунок описывает ситуацию, сложившуюся через некоторое время после выполнения еще нескольких операций. Необходимо отметить, что, несмотря на то что общий размер свободной памяти равняется 256 байтам, распределитель памяти не может выделить участок длиной, превышающей 128 байтов.

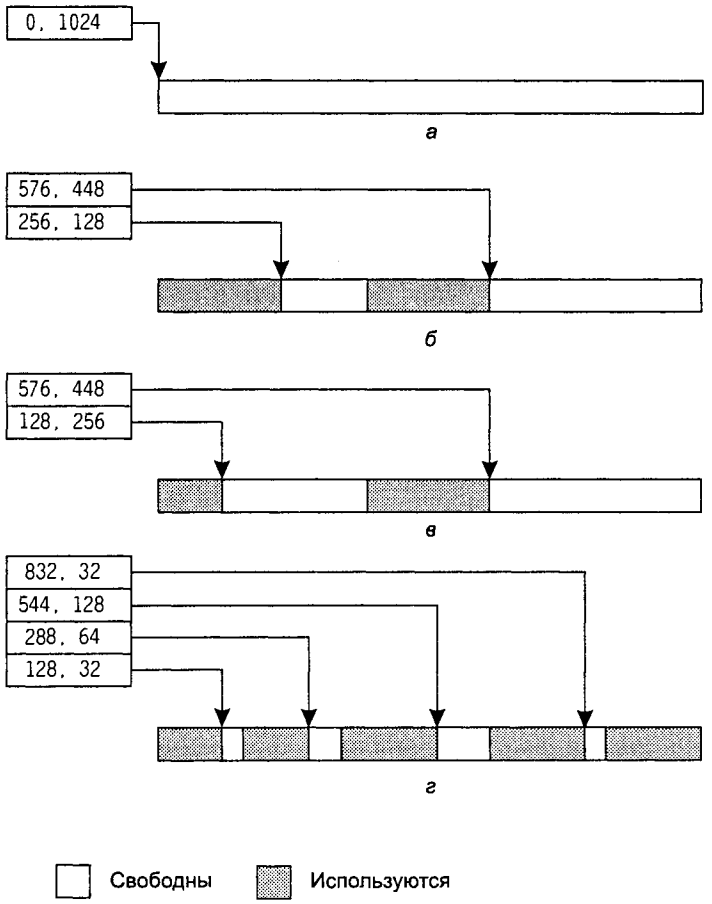


Рис. 12.2. Состояние карты ресурсов: *а* — изначальная конфигурация; *б* — после rmalloc(256), rmalloc(320) и rmfree(256,128); *в* — после rmfree(128,128); *г* — после еще некоторого количества операций

12.3.1. Анализ

Карта ресурсов применяется для простейшего случая распределения памяти. Она обладает несколькими преимуществами:

- ✦ алгоритм прост для практической реализации;
- ✦ карта ресурсов не ограничена в применении только для задач выделения и освобождения памяти. Она может быть использована и для обработки наборов различных других объектов, расположенных в определенном порядке и доступных для выделения и освобождения непрерывными участками (к таким объектам относятся, к примеру, вхождения таблицы страниц и семафоры);

- карта позволяет выделять точное количество байтов, равное запрошенному, без потерь памяти. На практике распределитель памяти всегда округляет количество выделяемой памяти до числа, делящегося на четыре или восемь, с точки зрения простоты и удобства выравнивания;
- от клиента не требуется всегда возвращать участок памяти, равный запрошенному. Как показывает предыдущий пример, клиент может освободить любую часть выделенного ранее участка, при этом распределитель памяти корректно отработает возникшую ситуацию. Такая возможность стала доступна потому, что в качестве аргумента процедуры rmfree() указывается размер освобождаемого участка, а учетная информация (то есть карта ресурсов) поддерживается системой отдельно от самой выделяемой памяти;
- распределитель соединяет последовательные участки памяти в один, что дает возможность выделять в дальнейшем области памяти различной длины.

Распределитель ресурсов имеет и ряд существенных недостатков:

- по истечении какого-то времени работы карта становится сильно фрагментированной. В ней оказывается большое количество участков малого размера. Это приводит к низкой востребованности ресурса. В частности, распределитель карты ресурсов плохо справляется с задачей обслуживания «больших» запросов;
- по мере увеличения фрагментации синхронно наращивается и сама карта ресурсов, так как для размещения данных о каждом новом свободном участке требуется новое вхождение. Если карта настроена на фиксированное количество вхождений, то в некоторый момент времени она может переполниться, а распределитель памяти потерять данные о какой-то доле свободных участков;
- если карта будет расти динамически, то для ее вхождений потребуется собственный распределитель. Эта проблема является «рекурсивной», и ниже вы увидите одно из ее решений;
- для решения задачи объединения свободных смежных областей памяти распределитель должен поддерживать карту, упорядоченную в порядке увеличения смещения от базового адреса. Операция сортировки весьма затратна, более того, она должна производиться по месту в том случае, если карта реализована в виде массива фиксированного размера. Нагрузка на систему, возникающая при сортировке, является весьма ощутимой даже в том случае, если карта размещается в памяти динамически и организована в виде связанного списка;
- часто требуется выполнять операцию последовательного поиска в карте с целью обнаружения достаточно большого для удовлетворения запроса участка. Эта процедура занимает много времени и выполняется медленнее при сильной фрагментации памяти;

- несмотря на наличие возможности возврата свободных участков памяти в хвост пула страничной подсистемы, алгоритм выделения и освобождения памяти не приспособлен для такой операции. На практике распределитель никогда не стремится достичь большей непрерывности вверенных ему областей памяти.

Карты ресурсов являются низкопроизводительными. Именно по этой причине они не подходят для применения в роли универсального распределителя памяти ядра. Средства взаимодействия процессов System V используют карты ресурсов для выделения наборов семафоров и областей данных сообщений. Подсистема виртуальной памяти 4.3BSD применяет методику для обработки вхождений таблицы страниц, указывающих на таблицы страниц прикладных процессов (см. подробнее в разделе 13.4.2).

Управление картами ресурсов может быть изменено в лучшую сторону сразу же в нескольких направлениях. Вхождение карты чаще всего удобнее размещать в первых нескольких байтах свободного участка памяти. Такой подход не требует выделения дополнительной памяти для хранения самой карты и ее новых вхождений. На первый свободный участок может указывать единственная глобальная переменная, в то время как внутри каждого свободного участка может храниться информация о его размере и указатель на следующий свободный участок в области памяти. В этом случае минимальный размер участка должен равняться по крайней мере двум словам, в одном из которых будет храниться размер, а во втором — указатель. Такое требование можно удовлетворить путем принудительного выделения или освобождения участков, длина которых кратна двум (словам). В файловой системе FFS (см. раздел 9.5) используется немного другой вариант такого алгоритма для обработки свободного места в блоках каталогов.

Несмотря на то, что перечисленные изменения подходят для обычного распределителя памяти, они не могут быть реализованы в специализированных случаях применения карты ресурсов, в которых не остается свободного места для хранения данных о вхождениях, например, при использовании карты для семафоров или вхождений таблицы страниц.

12.4. Простые списки, основанные на степени двойки

Списки свободной памяти, основанные на степени числа 2, чаще всего применяются для реализации процедур malloc() и free() в библиотеке С прикладного уровня. Методика использует набор списков свободной памяти. В каждом списке хранятся буферы определенного размера. Размер буфера всегда кратен степени числа 2. На рис. 12.3 показан пример шести списков, содержащих буферы размером 32, 64, 128, 256, 512 и 1024 байта соответственно.

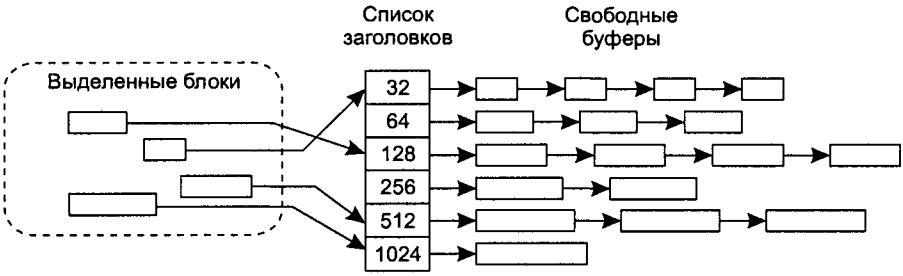


Рис. 12.3. Список свободных буферов с длиной, кратной степени числа 2

Каждый буфер имеет заголовок длиной в одно слово, этим фактом ограничивая возможности использования соотносимой с ним области памяти. Если буфер свободен, в его заголовке хранится указатель на следующий свободный буфер. В другом случае в заголовок буфера помещается указатель на список, в который он должен быть возвращен при освобождении. В некоторых реализациях заголовок содержит вместо этой информации размер выделенной области. Это позволяет обнаружить некоторые «баги», однако требует от процедуры free() вычисления местонахождения списка исходя из данных о размерах буферов.

Для выделения памяти клиент вызывает malloc(). В качестве входного аргумента функции передается желаемая величина участка. Распределитель вычисляет минимальный размер буфера, подходящий для удовлетворения запроса. Для этого необходимо прибавить к заданной величине слово, в котором будет размещен заголовок, и округлить полученное значение кверху до числа, являющегося степенью двойки. Буферы размером 32 байта подходят для выделения памяти объемом 0–28 байтов, 64-байтовые буферы — для объемов 29–60 байтов и т. д. После вычисления распределитель извлекает буфер из соответствующего списка и оставляет в заголовке указатель на список свободных участков памяти. Процесс, запрашивающий участок памяти, получает в ответ указатель на следующий после заголовка байт.

Если клиент желает освободить буфер, то он вызывает для этой цели процедуру free(), входным аргументом которой является указатель, возвращенный malloc(). При этом нет необходимости уточнять размер буфера. Очевидно, что результатом выполнения операции free() станет освобождение буфера целиком. Технология не поддерживает возможности частичного освобождения участка памяти. Процедура free() производит обращение к заголовку буфера, откуда извлекает указатель на список свободных буферов и затем переносит его в этот список.

Распределитель памяти может быть проинициализирован путем предварительного выделения ему определенного количества буферов в каждом списке. Другой вариант предполагает создание изначально свободного списка, для заполнения которого вызывается распределитель страничного уровня. Если список становится пустым, обработка последующего запроса malloc()

для выделения участка памяти определенного размера может быть произведена одним из перечисленных способов:

- запрос блокируется до освобождения буфера подходящей длины;
- запрос удовлетворяется путем выделения буфера большего размера. Поиск свободного буфера начинается со следующего списка (по возрастанию размеров буферов) и продолжается до тех пор, пока не будет обнаружен непустой список;
- происходит запрос дополнительного объема памяти от распределителя страничного уровня. При этом создается нужное количество буферов заданного размера.

Каждый метод обладает определенными достоинствами и недостатками. Наиболее подходящий вариант действий зависит от ситуации. Например, реализация алгоритма на уровне ядра может использовать в запросах на выделение памяти дополнительный аргумент приоритетности. В таком случае распределитель вправе блокировать низкоприоритетные запросы, если в списке не имеется ни одного свободного буфера указанного размера, при этом используя два последних метода для немедленной обработки высокоприоритетных запросов.

12.4.1. Анализ

Описанный алгоритм выделения памяти является весьма простым и быстрым. Его основным отличием от предыдущего является отсутствие необходимости длительного поиска в карте ресурсов и решение проблемы фрагментации области памяти. При использовании буферов нижний предел производительности всегда ограничен. Распределитель памяти предоставляет понятный программный интерфейс, важнейшим преимуществом которого является процедура free(), в качестве входного аргумента которой больше не нужно задавать размер буфера. В результате этого выделенный буфер может передаваться другим функциям или подсистемам, а также освобождаться, и для этого достаточно использовать всего указатель на него. С другой стороны, интерфейс не позволяет клиенту освобождать буферы частично.

В алгоритме имеется несколько значимых недостатков. Округление количества запрошенной памяти кверху до следующей степени двойки часто приводит к реальному использованию лишь части буфера, что в результате влечет неэкономное распределение памяти. Проблема становится еще более очевидной из-за необходимости хранения заголовков буферов внутри самих выделенных буферов. Многие запросы требуют объемов памяти, уже равных степени числа 2. Удовлетворение таких запросов имеет следствием почти 100%-ную потерю выделяемых участков памяти, так как чтобы поместить заголовок в буфер (а это всего 4 байта), распределитель округлит необходимое количество байтов до следующей степени двойки. Например, в ответ на просьбу о предоставлении 512 байтов памяти будет выделено уже 1024 байта.

Технология не поддерживает слияния смежных буферов, что позволило бы удовлетворить запросы на выделение областей памяти большей протяженности. Размер каждого буфера остается неизменным в течение его периода жизни. Большие буферы иногда можно использовать для удовлетворения запросов на области памяти малой длины. Несмотря на то, что некоторые реализации алгоритма позволяют «заимствовать» часть памяти у страничной системы, не существует механизма обратной передачи таких буферов после освобождения.

И хотя алгоритм степени двойки намного быстрее работы с картой ресурсов, его производительность может быть поднята еще больше. В частности, реализация перебора, показанная в листинге 12.1, является слишком медленной и неэффективной.

Листинг 12.1. Грубая реализация процедуры malloc()

```
void*malloc (size)
{
   int ndx = 0; /* индекс списка свободных буферов */
   int bufsize = 1 << MINPOWER; /* минимальный размер буфера */
   size+=4; /* выделение байтов для заголовка */
   assert (size <= MAXBUFSIZE);
   while (bufsize < size) {
      ndx++;
      bufsize <<= 1;
   }
   ... /* на этом этапе ndx является индексом к соответствующему списку
      свободных буферов */
}
```

В следующем разделе будет описан более совершенный алгоритм, в котором решены многие перечисленные проблемы.

12.5. Распределитель Мак-Кьюзика—Кэрелса

Маршалл Кирк Мак-Кьюзик и Майкл Дж. Кэрелс разработали усовершенствованный метод выделения памяти [12], который был реализован во многих вариантах системы UNIX, в том числе 4.4BSD и Digital UNIX. Методика позволяет избавиться от потерь в тех случаях, когда размер запрашиваемого участка памяти равен некоторой степени двойки. В нем также была произведена оптимизация перебора в цикле (см. листинг 12.1). Такие действия теперь нужно производить только в том случае, если на момент компиляции неизвестен размер выделенного участка.

Алгоритм подразумевает, что память разбита на набор последовательных страниц, и все буферы, относящиеся к одной странице, должны иметь одинаковый размер (являющийся некоторой степенью числа 2). Для управления

страницами распределитель использует дополнительный массив kmemsizes[]. Каждая страница может находиться в одном из трех перечисленных состояний.

- Быть свободной. В этом случае соответствующий элемент массива kmemsizes[] содержит указатель на элемент, описывающий следующую свободную страницу.
- Быть разбитой на буферы определенного размера. Элемент массива содержит размер буфера.
- Являться частью буфера, объединяющего сразу несколько страниц. Элемент массива указывает на первую страницу буфера, в которой находятся данные о его длине.

На рис. 12.4 показан простейший пример организации страницы размером 1024 байта. Массив freelistarr[] содержит заголовки всех буферов, имеющих размер меньше одной страницы.

Так как длина всех буферов одной страницы одинакова, нет нужды хранить в заголовках выделенных буферов указатель на список свободных буферов. Процедура free() находит страницу путем маскирования младших разрядов адреса буфера и обнаружения размера буфера в соответствующем элементе массива kmemsizes[]. Отсутствие заголовка в выделенных буферах позволяет экономить память при удовлетворении запросов с потребностью в памяти, кратной некоторой степени числа 2.

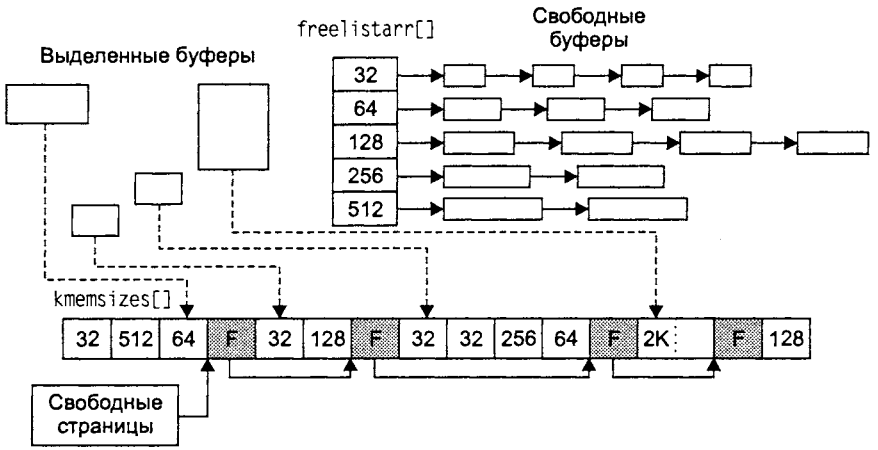


Рис. 12.4. Распределитель Мак-Кьюзика—Кэрелса

Вызов процедуры malloc() заменен макроопределением, которое производит округление значения длины запрашиваемого участка вверх до достижения числа, являющегося степенью двойки (при этом не нужно прибавлять какие-либо дополнительные байты на заголовок) и удаляет буфер из соответствующего списка свободных буферов. Макрос вызывает функцию malloc() для запроса одной или нескольких страниц тогда, когда список свободных буферов

необходимого размера пуст. В этом случае malloc() вызывает процедуру, которая берет свободную страницу и разделяет ее на буферы необходимого размера. Здесь цикл заменен на схему вычислений по условию. Реализация для области памяти, схематично изображенной на рис. 12.4, показана в листинге 12.2.

Листинг 12.2. Применение макроопределения, увеличивающего скорость работы malloc()

```
#define NDX(size) \
   (size) > 128 \
      ? (size) > 256 ? 4 : 3 \
      : (size) > 64 \
         ? 2\
         : (size) > 32 ? 1 : 0
#define MALLOC (space, cast, size, flags) \
   {   \
      register struct freelisthdr* flh; \
      if (size <= 512 &&\
      (flh = freelistarr [NDX(size)]!= NULL) { \
         space = (cast)flh->next; \
         flh->next = *(caddr_t *)space; \
      } else \
         space = (cast)malloc (size, flags);\
   }
```

Основным преимуществом такого решения является тот факт, что если выделяемый размер известен в момент компиляции, определение NDX() сжимается до константы времени компиляции, что позволяет сократить значительное количество инструкций. Второй макрос применяется для простейших ситуаций освобождения буфера, вызов самой функции осуществляется в редких случаях, например, когда требуется освободить буфер большого размера.

12.5.1. Анализ

Приведенный алгоритм значительно улучшает методику распределения памяти на основе степени числа 2, описанную в разделе 12.4. Он работает намного быстрее, потери памяти при его применении сильно сокращаются. Алгоритм позволяет эффективно обрабатывать запросы на выделения как малых, так и больших участков памяти. Однако описанная методика обладает и некоторыми недостатками, связанными с необходимостью использования участков, равных некоторой степени числа 2. Не существует какого-либо способа перемещения участков из одного списка в другой. Это делает распределитель не совсем подходящим средством при неравномерном использовании памяти, например, если системе необходимо много буферов малого размера на короткий промежуток времени. Технология также не дает возможности возвращать участки памяти, запрошенные ранее у страничной системы.

12.6. Метод близнецов

Метод близнецов (или buddy system) является схемой выделения памяти, сочетающей в себе возможность слияния буферов и распределитель по степени числа 2[1] [13]. В основе метода лежит создание буферов малого размера путем деления пополам больших буферов и слияния смежных буферов по мере возможности. При разделении буфера на два каждая половина называется *близнецом* (buddy) второй.

Рассмотрим метод близнецов на простом примере (см. рис. 12.5), в котором алгоритм применяется для обработки 1024-байтового блока с минимальным размером участков в 32 байта. Распределитель использует битовую карту для отслеживания каждой 32-битовой порции блока: если бит установлен, то соответствующий участок занят. Она также поддерживает список свободных буферов любого допустимого размера (по степеням двойки в диапазоне от 32 до 512). Изначально блок представляет собой единый буфер. Представим, что произойдет с ним при поступлении некоей последовательности запросов на выделение памяти и ответную реакцию распределителя.

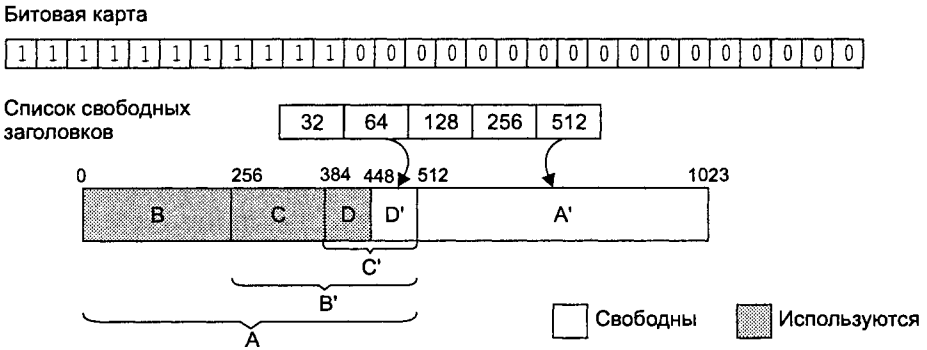


Рис. 12.5. Метод близнецов

1. allocate(256). Блок делится на два близнеца, A и A'; блок A' поступает в список свободных 512-байтовых буферов. Затем буфер A разбивается на B и B'. B' заносится в список свободных 256-байтных буферов, а буфер B передается клиенту.
2. allocate(128). Распределитель обнаруживает, что список свободных 128-байтовых буферов пуст. Тогда он проверяет список 256-байтовых буферов, изымает оттуда B' и разделяет его на C и C'. После этого буфер C' помещается в список свободных 128-байтовых буферов, а буфер B возвращается клиенту.

[1] Здесь описывается двоичная методика близнецов, которая является наиболее простой и популярной из систем близнецов. На самом деле можно реализовать и другие варианты алгоритма, разделяющие буферы на четыре, восемь и более частей.

3. allocate(64). Распределитель выясняет, что список свободных 64-байтовых буферов пуст. Тогда он обходит список 128-байтовых буферов, удаляет оттуда C′ и дробит его на D и D′. Последний передается в список свободных 64-байтовых буферов, а буфер D возвращается клиенту. Ситуация на этот момент времени показана на рис. 12.5.
4. allocate(128). Распределитель узнает, что списки свободных 128- и 256-байтовых буферов пусты. Он проверяет список свободных 512-байтовых буферов и удаляет оттуда буфер A′. Затем A′ разделяется на два близнеца E и E′, а буфер E — на F и F′. После этого буфер E′ переносится в список свободных буферов размером 256 байтов, а буфер F′ — в список 128-байтовых буферов. Буфер F передается клиенту.
5. release(C, 128). Буфер C возвращается в список свободных 128-байтовых буферов. Это приводит к состоянию, продемонстрированному на рис. 12.6.

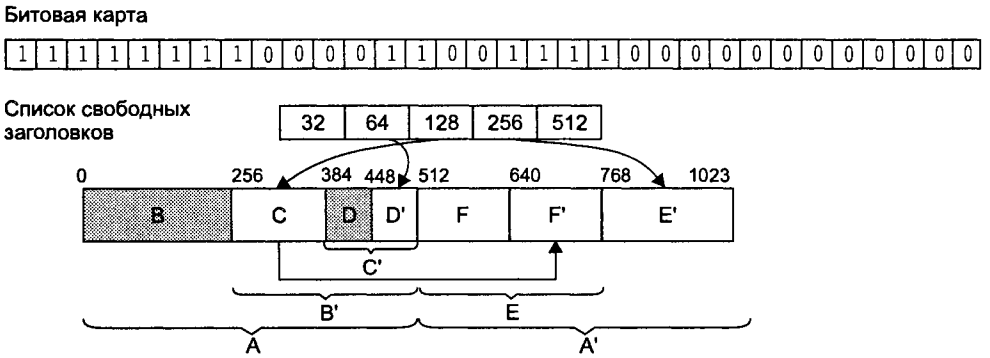


Рис. 12.6. Метод близнецов, стадия 2

До этого момента нам ни разу не потребовалось совмещать смежные буферы. Однако представьте, что следующий запрос к распределителю будет таков:

6. release(D,64). Обработав такой запрос, распределитель увидит, что буфер D′ также свободен. Тогда он совместит его со своим близнецом для получения буфера C′. Но в этот момент буфер не занят, поэтому распределитель объединит его с C′ для образования B′. После буфер B′ будет помещен в список свободных 256-байтовых буферов. Состояние рассматриваемого блока памяти на этом этапе продемонстрировано на рис. 12.7.

Описывая принцип работы метода близнецов, необходимо упомянуть следующие интересные моменты:

- размер запрашиваемой памяти обычно округляется до следующей степени числа 2;

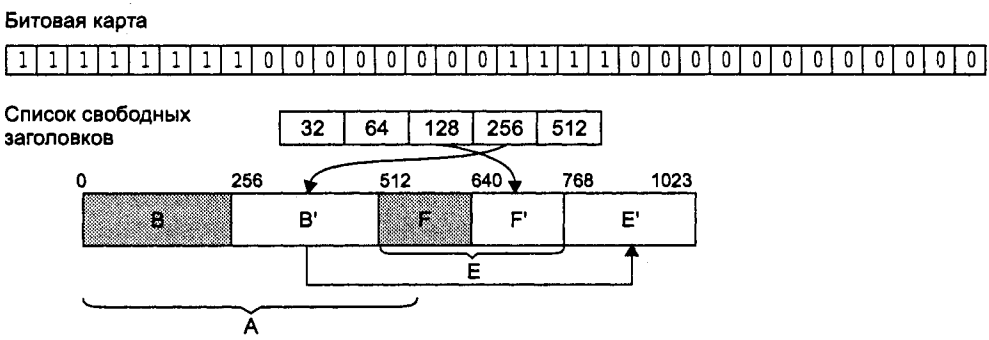


Рис. 12.7. Метод близнецов, стадия 3

- в каждом запросе рассматриваемого примера соответствующий список свободных буферов оказывался пустым. В реальности это чаще всего не так. Если в списке имеется буфер необходимого размера, распределитель будет использовать именно его, поэтому совмещение буферов становится ненужным;
- адрес и размер буфера есть в совокупности достаточная информация для обнаружения близнеца. Это происходит потому, что алгоритм размещает буфер, руководствуясь его длиной. Например, у 128-байтового буфера, имеющего смещение 256, близнец находится по адресу 384, в то время как 256-байтовый буфер с таким же смещением обладает близнецом, расположенным в точке 0;
- при обработке каждого запроса происходит обновление битовой карты с целью отражения в ней нового состояния буфера. При слиянии распределитель узнает из этой карты, свободен ли необходимый ему буфер-близнец;
- в нашем примере распределитель обрабатывает единственную страницу памяти размером 1024 байта. На самом деле он может обслуживать одновременно несколько различных страниц памяти. Единый набор заголовков свободных буферов может хранить данные о буферах всех страниц сразу. При этом слияние будет происходить по ранее приведенному алгоритму, так как близнец определяется исходя из смещения буфера на конкретной странице. В этом случае, однако, распределитель будет поддерживать отдельные битовые карты для каждой страницы памяти.

12.6.1. Анализ

Методика близнецов обладает замечательным свойством соединения смежных свободных буферов. Возможность изменения размеров участков памяти и их повторного использования в преобразованном виде придает алгоритму большую степень гибкости. Методика позволяет легко обмениваться памятью

между распределителем и страничной системой. Если распределителю в какой-то момент времени понадобился больший объем памяти, он вправе запросить новую страницу и затем при необходимости поделить ее на буферы. После того как процедура освобождения соберет страницу в единый блок, распределитель может вернуть ее обратно страничной системе.

Главным недостатком алгоритма является его производительность. Каждый раз при освобождении буфера распределитель пытается соединить вместе как можно больше участков памяти. При чередовании запросов на выделение и освобождение алгоритм будет объединять только что разделенные буферы. Операция объединения рекурсивна, что дает в результате замедление ее работы. В следующем разделе будет рассмотрена реализация усовершенствованного алгоритма близнецов в SVR4, преодолевающая это узкое место.

Еще одним недостатком методики является ее программный интерфейс. Процедура освобождения должна получать в качестве аргумента адрес и размер буфера. Более того, распределитель памяти поддерживает освобождение только целого буфера. Частичное освобождение является неэффективным, так как в этом случае неполный буфер не будет иметь близнеца.

12.7. Алгоритм отложенного слияния в SVR4

Основной проблемой простого метода близнецов являлась низкая производительность работы из-за необходимости постоянного разделения и слияния буферов. Обычно распределители памяти находятся в некотором равновесном состоянии, то есть количество используемых буферов каждого размера остается в примерно известном диапазоне. При таких условиях совмещение буферов не дает никаких преимуществ, время частично тратится впустую. Слияние необходимо только при постоянно меняющихся условиях использования буферов памяти.

Задержка слияния определяется как время, необходимое либо для соединения буфера со своим близнецом, либо для определения того факта, что близнец занят. Объединение приводит к появлению буфера размера увеличенной степени числа 2, процесс повторяется до тех пор, пока распределитель перестанет находить свободных близнецов (то есть операция является рекурсивной). В алгоритме близнецов каждая операция освобождения приводит, по крайней мере, к однократной задержке слияния, а чаще всего, не только к одной.

Интуитивно-логичное решение проблемы заключается в откладывании операции слияния до того момента, пока она не будет необходима, — вот тогда и нужно производить сращивание как можно большего числа буферов. Несмотря на то, что такой подход уменьшит время, затрачиваемое в среднем

для выделения и освобождения буферов, то небольшое количество запросов, которое требует вызова операции объединения, будет обслуживаться слишком медленно. Распределитель памяти может быть вызван критичными к скорости выполнения функциями, такими как обработчики прерываний, отсюда становится очевидной необходимость управления таким наихудшим вариантом. Необходимо создать такое средство, которое бы задерживало операции слияния буферов, кроме критичных случаев, а также умело распределять операции объединения на несколько запросов (что не так сильно увеличит время реакции на запрос, повлекший за собой необходимость вызова этой операции). В работе [10] описывается решение проблемы, основанное на ассоциации с каждым классом буферов *«водяных знаков»* (watermarks). В системе SVR [2] используется та же идея, но с большей эффективностью (см. следующие разделы).

12.7.1. Отложенное слияние

Освобождение буфера памяти производится в два этапа. Сначала буфер помещается в список свободных буферов, что делает его доступным для последующих запросов на выделение. После этого буфер помечается в битовой карте как свободный и по возможности присоединяется к смежным буферам (операция слияния). В обычной системе близнецов при каждой операции освобождения выполняются обе перечисленные стадии.

Методика отложенного слияния подразумевает выполнение первой ступени, в результате которой буфер становится *свободным локально* (доступным для выделения внутри своего класса, но не для слияния). Необходимость выполнения второго шага зависит от состояния класса буферов. В любой момент времени класс содержит N буферов, среди которых A активных буферов, L локально свободных буферов и G глобально свободных буферов (то есть помеченных свободными в битовой карте и доступных для слияния). Следовательно,

```
N = A + L + G
```

В зависимости от величины этих параметров класс может находиться в одном из трех состояний:

- **«безмятежном»** (lazy). Буферы расходуются сбалансированно (количество запросов на выделение и освобождение является примерно равным), слияние не является необходимым;
- **требующем восстановления равновесия** (reclaiming). Расход буферов близок к пограничному, операция слияния становится необходимой;
- **требующем ускоренного выполнения** (accelerated). Расход буферов неравновесный, требуется как можно более быстрое проведение слияния.

Состояние определяется параметром, называемым *допуском* (slack) и определяемым по следующей формуле:

```
slack = N - 2L - G
```

Система находится в «ленивом» состоянии, если допуск не меньше двух, состояние восстановления наступает при достижении значения единицы. Состояние разгона возникает при значении допуска 0. Алгоритм подразумевает, что величина допуска никогда не может стать отрицательной. В работе [2] предлагается детальное описание, объясняющее, почему допуск является эффективным средством оценки состояния класса буферов.

При освобождении буфера распределитель памяти SVR4 переносит его в список свободных буферов и проверяет состояние класса. Если список находится в безразличном состоянии, на этом операция завершается. Буфер не помечается свободным в битовой карте. Такой буфер является отложенным, что фиксируется определенным флагом в его заголовке (который имеется только у буферов, находящихся в списке свободных буферов). Он доступен для удовлетворения запросов на участки памяти совпадающих размеров, но не может быть сцеплен со смежными буферами.

Если список находится в состоянии, требующем восстановления равновесия, распределитель помечает буфер как свободный в битовой карте и производит по возможности его слияние. Если список нуждается в скорейшей реорганизации, распределитель совмещает два буфера — только что освобожденный и дополнительно задержанный буфер, если таковой имеется. Если буфер, подвергшийся слиянию, оказывается в списке следующего по величине размера (степени 2), распределитель оценивает состояние этого класса на предмет дальнейшего слияния буферов. Каждая такая операция изменяет значение допуска, следовательно, необходима переоценка.

Для эффективной реализации описанного алгоритма буфер двунаправленно связан со списком свободных буферов. Задержанные буферы передаются в начало списка, остальные буферы — в его конец. В этом случае задержанные буферы будут повторно предоставлены для выделения в первую очередь, что является наиболее приемлемым вариантом, так как размещение таких буферов происходит быстрее всего (нет необходимости обновлять карту). Более того, в состоянии дисбаланса дополнительный отсроченный буфер может быть быстро проверен и извлечен из головы списка. Если первый буфер окажется не задержанным, можно сказать, что в списке таких буферов больше нет.

Методика отложенного слияния существенно усовершенствовала технологию близнецов. В равновесном режиме все списки находятся в безразличном состоянии, поэтому на операции слияния время не тратится. Даже в том случае, если список кренится в сторону недостатка либо избытка буферов, распределитель памяти производит слияние по крайней мере двух буферов при обработке каждого запроса. Следовательно, в худшем варианте мы имеем двойную задержку слияния на класс, что не так плохо по сравнению с простой моделью.

В работе [2] анализируется производительность простого и модифицированного алгоритма близнецов при сравнении в различных тестовых рабочих средах. Из результатов измерений видно, что последний метод показывает улучшение от 10 до 32% по сравнению с обычной схемой. Однако, как этого и следовало ожидать, методика «ленивого» слияния обладает большим значением характеристики разброса и с натугой справляется с освобождением памяти в трудных условиях.

12.7.2. Особенности реализации алгоритма в SVR4

В системе SVR4 применяются две модели пулов памяти — малых (small) и больших (large). Каждый малый пул начинается с блока длиной 4096 байтов, поделенного на 256-байтовые буферы. Первые два буфера используются для хранения структур данных блока (таких как битовая карта), остальные буферы доступны для выделения или разделения. Малые пулы позволяют создавать буферы размером от 8 до 256 байтов, кратным степени числа 2. Большой пул памяти берет начало с единственного блока размером 16 Кбайт и используется для выделения буферов размером от 512 байтов до 16 Кбайт. В нормальном состоянии оба типа пулов памяти содержат большое количество активных буферов.

Распределитель может обмениваться со страничной подсистемой участками памяти, имеющими размер пула. При необходимости получения некоторого объема памяти он запрашивает от страничного распределителя область больше или меньше. После слияния буферов в пуле распределитель возвращает его страничной подсистеме.

Буферы большого пула подвергаются объединению в соответствии с алгоритмом отложенного слияния, так как размер пула соответствует классам буферов большого размера. Для малого пула сращивание буферов осуществляется только до достижения значения 256 байтов. Сборка 256-байтовых буферов пула требует применения отдельной функции. Ее выполнение занимает ощутимый промежуток времени и вследствие этого должно происходить в фоновом режиме. Для проведения слияния и возвращения свободных пулов страничному распределителю служит системный процесс под названием kmdaemon.

12.8. Зональный распределитель в системе Mach-OSF/1

Зональный (zone) распределитель памяти, используемый в системах Mach и OSF/1, обладает возможностью быстрого выделения памяти и сбора мусора в фоновом режиме. Каждому классу динамически размещенных объектов

(таких как структуры proc, удостоверения или заголовки сообщений) выделяется собственная зона, которая является всего лишь набором свободных объектов класса. Даже в том случае, если объекты различных классов имеют одинаковый размер, каждому классу назначается отдельная зона. Например, данные как о *преобразовании портов*, так и о *наборах портов* (см. главу 6) имеют размер 104 байта [7], однако оба класса обладают собственными зонами. Система также поддерживает зоны по степени числа 2, используемые клиентами, для которых не требуется закрытое множество объектов.

Изначально зоны заполняются путем выделения памяти от распределителя страничного уровня, который позже может предоставить дополнительные ресурсы при необходимости. Каждая страница вправе быть использована только для одной зоны; следовательно, все объекты, физически расположенные на одной странице, принадлежат одному и тому же классу. Свободные объекты каждой зоны управляются при помощи связанного списка, в голове которого находится структура zone. Объекты сами по себе выделяются из *зоны над зонами* (zone of zones), каждый элемент которой является структурой zone.

Любая подсистема ядра инициализирует зону, если это необходимо. Этой цели служит функция

```
zinit (size, max, alloc, name);
```

где size указывает на размер каждого объекта, max — максимальный размер зоны в байтах, alloc — объем памяти, добавляемый зоне каждый раз, когда список свободных объектов становится пустым (ядро округляет число до целого количества страниц), name является строкой, описывающей объекты зоны. Функция zinit() выбирает структуру zone из зоны над зонами и записывает в нее значения size, max и alloc. Затем происходит запрос первоначальной области памяти размером alloc байтов у распределителя страничной памяти и деление ее на объекты размером size байтов, которые после помещаются в список свободных объектов. Все активные структуры zone связаны списком, описываемым глобальными переменными first_zone и last_zone (см. рис. 12.8). Первым элементом списка является зона над зонами, из которой инициируется выделение всех остальных элементов.

Процесс выделения и освобождения участков памяти выполняется с очень высокой скоростью и требует всего лишь удаления (или добавления) объектов из списка свободных объектов. Если при попытке выделения список окажется пустым, распределитель запросит у страничной подсистемы дополнительный объем памяти, равный alloc. Если размер пула достигнет значения max, все последующие запросы на выделение завершатся неудачно.

12.8.1. Сбор мусора

Очевидно, что вышеописанная схема требует проведения процедуры сбора мусора, иначе в результате неравномерного использования памяти большая

ее часть станет недоступна. Сбор мусора производится в фоновом режиме, поэтому не замедляет выполнение отдельных операций. Распределитель поддерживает массив, называемый *картой страниц зоны*, где каждый элемент относится к одной из страниц зоны. Вхождение карты содержит два счетчика:

- in_free_list — количество объектов из списка свободных объектов на странице;
- alloc_count — общее количество объектов страницы.

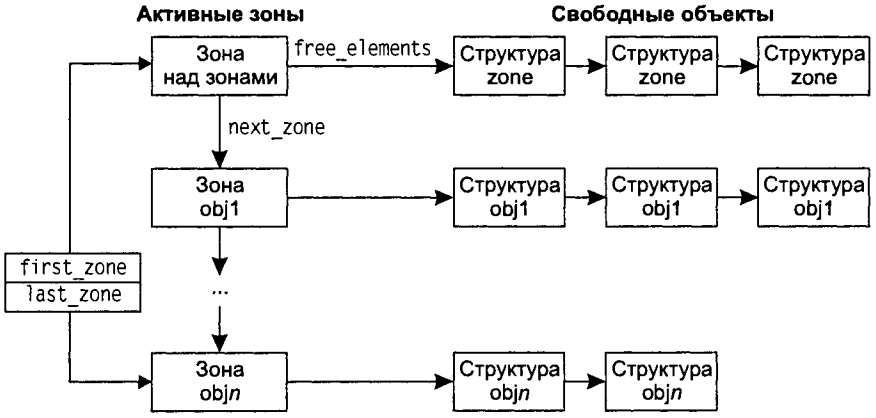


Рис. 12.8. Зональный распределитель

Значение alloc_count устанавливается при запросе страницы зоной от страничного распределителя памяти. Так как размер страницы может не быть равен размеру одного или нескольких объектов, существует возможность занятия одним объектом сразу двух страниц. В этом случае объект учитывается в счетчиках alloc_count обеих страниц. Счетчик in_free_list не изменяется при каждой операции выделения или освобождения. Вычисление его значения происходит при каждом вызове операции сбора мусора. Такой подход позволяет свести до минимума запаздывание обработки отдельных запросов.

Процедура сбора мусора zone_gc() запускается задачей swapper при каждом ее выполнении. Процедура просматривает все списки зон и для каждой из них дважды обходит список свободных объектов. При первом просмотре происходит выявление всех свободных элементов и инкрементирование счетчика in_free_count той страницы, к которой относится такой объект. Если к моменту завершения сканирования значения in_free_list и alloc_count окажутся равными, то это означает, что все объекты страницы свободны и страница может быть использована вновь. На втором этапе просмотра zone_gc() удаляет все такие объекты из списка. После этого происходит вызов kmemfree(), выполняющий передачу свободных страниц распределителю страничного уровня.

12.8.2. Анализ

Зональный распределитель является быстрым и эффективным. Он обладает простым программным интерфейсом. Объекты размещаются при помощи

```
obj = void* zalloc (struct zone* z);
```

где z указывает на зону для данного класса объектов, инициализированного ранее путем вызова zinit(). Объекты освобождаются посредством вызова

```
void zfree (struct zone* яб void* obj);
```

Процедура требует, чтобы клиенты освобождали объекты целиком, а также знали, какой зоне такие объекты принадлежат. Методика не предоставляет возможности освобождения лишь части выделенного ранее объекта.

Одним из интересных свойств методики является тот факт, что зональный распределитель использует самого себя для размещения структур zone свежесозданных зон. Это приводит к возникновению дилеммы «курицы и яйца» при первоначальной загрузке системы. Системе управления памятью требуются зоны для размещения собственных структур данных, в то время как зональной подсистеме необходим распределитель памяти страничного уровня, который является частью подсистемы управления памятью. Эта проблема решена путем использования статически сконфигурированного небольшого участка памяти, в котором создается и заполняется *зона над зонами.*

Объекты зон имеют размер, равный запрошенному, что избавляет от потерь памяти, присущих методам степени двойки. Процедура сбора мусора предоставляет механизм повторного использования памяти, свободные страницы могут быть возвращены страничной системе и позже запрошены для размещения других зон.

Основным спорным фактором методики является эффективность работы процедуры сбора мусора. Процедура выполняется в фоновом режиме, поэтому она не влияет напрямую на производительность отдельных запросов на выделение или освобождение памяти. Алгоритм сбора мусора является медленным и требует последовательного просмотра сначала всех свободных объектов, а затем — всех страниц зоны. Это влияет на общую скорость реакции системы, так как процесс сбора мусора занимает процессор до тех пор, пока не завершит свою работу.

В работе [14] утверждается, что процесс сбора мусора не сильно влияет на скорость параллельной компиляции. Однако не существует опубликованных где-либо оценок издержек процесса сбора мусора и трудно оценить его влияние на общую производительность системы. Алгоритм сбора мусора сложен и неэффективен, его стоит сравнить с составным распределителем (описываемым в разделе 12.10), который является более простым и быстрым альтернативным вариантом решения этой задачи.

12.9. Иерархический распределитель памяти для многопроцессорных систем

Выделение памяти на многопроцессорных системах разделения времени требует учета некоторых дополнительных факторов. Структуры данных, такие как списки свободных буферов или карты размещения, используемые в традиционных системах, в условиях выполнения на многопроцессорных машинах нуждаются в защите от доступа при помощи блокировки. В крупных параллельных системах это приводит к сложным условиям такой блокировки, в результате чего процессоры часто оказываются в режиме ожидания ее снятия.

Одно из решений этой проблемы реализовано в Dynix, многопроцессорном варианте системы UNIX для машин Sequent S2000 [11]. Здесь применена иерархическая схема выделения памяти, поддерживающая программный интерфейс System V. Многопроцессорные машины Sequent используются для крупных интерактивных сред обработки транзакций. Распределитель памяти при такой загрузке демонстрирует высокую производительность работы.

Архитектура распределителя приведена на рис. 12.9. Нижний *процессорный* уровень (per-CPU) производит наиболее быстрые операции, в то время как верхний уровень *слияния в страницу* (coalesce-to-page) используется для медленного процесса слияния. Есть также (на рис. не показан) еще один уровень распределителя, *слияния в vmblock* (coalesce-to-vmblock), который управляет выделением страниц внутри больших (4 Мбайт) областей памяти.

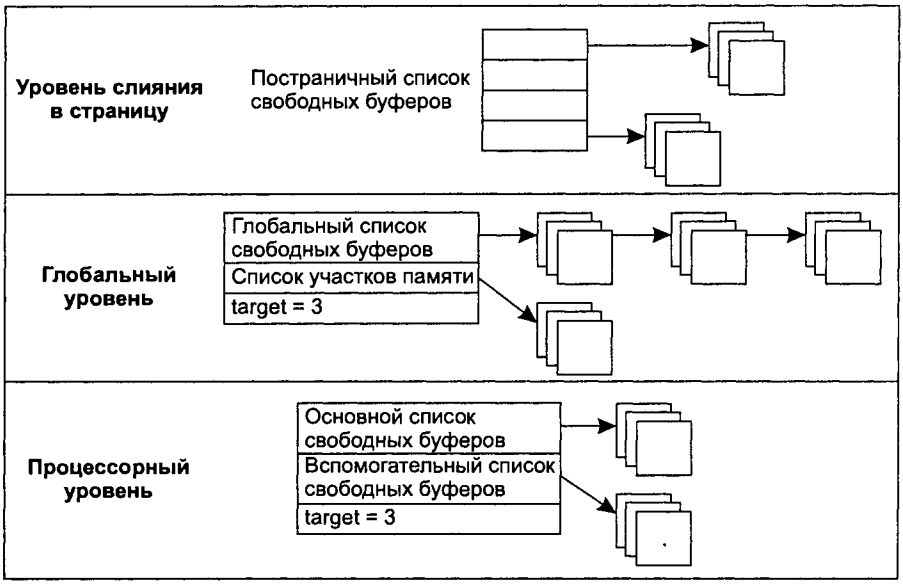


Рис. 12.9. Иерархический распределитель памяти для многопроцессорных систем

Процессорный уровень управляет одним набором пулов степени двойки для каждого процессора. Такие области изолированы от других процессоров и, следовательно, могут быть доступны без глобальной блокировки. Выделение и освобождение памяти происходит в большинстве случаев весьма быстро, так как в этом процессе участвует только локальный список свободных блоков.

Если список становится пустым, он может быть заполнен заново на *глобальном уровне* (global layer), поддерживающем собственные пулы памяти размером степени двойки. Освободившиеся буферы в кэше процессорного уровня могут быть переданы обратно на глобальный уровень. Для оптимизации обмена между этими двумя уровнями буферы группируются по значению target (например, на рис. 12.9 за один проход перемещаются три буфера), что позволяет защититься от ненужных операций со связанными списками.

Для реализации такой возможности на процессорном уровне поддерживается два списка свободных буферов — *основной* (main) и *вторичный* (aux). При выделении и освобождении буферов напрямую используется первичный список. Если он становится пустым, в него перемещаются буферы из дополнительного списка, который пополняется на глобальном уровне. Если основной список переполнится (то есть превысит target), то они будут отправлены во вторичный список, буферы которого будут возвращены на глобальный уровень. В этом случае доступ к глобальному уровню будет осуществлен по крайней мере target раз. Значение переменной target является настраиваемым. Увеличение значения target уменьшает количество операций доступа к глобальному уровню, но при этом одновременно удерживается большее количество буферов в кэшах процессорного уровня.

На глобальном уровне поддерживаются общие списки свободных буферов степени двойки. Каждый список поделен на группы буферов в соответствии с target. Случается, что нужно передать на глобальный уровень некоторое число блоков сверх значения этой переменной. Такие блоки добавляются в отдельный список сегментов памяти, являющийся промежуточной областью перед переносом в общий список свободных буферов.

Если объем общего списка превысит значение глобальной переменной target, «лишние» буферы будут возвращены на уровень *слияния в страницу*, на котором хранятся постраничные списки свободных буферов (все буферы одной страницы имеют одинаковый размер). На этом уровне буферы помещаются в список свободных буферов, к которому они относятся, и увеличивается счетчик свободных объектов страницы. Когда все буферы страницы будут возвращены в список, ее можно передавать обратно страничной системе. С другой стороны, уровень слияния в страницу может запрашивать дополнительную память от страничной системы с целью создания новых буферов.

На уровне слияния в страницу происходит сортировка всех списков по количеству свободных блоков на каждой из них. Выделение буферов выполняется из страницы, имеющей наименьшее количество свободных блоков.

Страницам с большим количеством свободных блоков дается дополнительный шанс на освобождение остальных свободных блоков, что увеличивает вероятность их возврата страничной системе. Результатом таких действий является высокая степень эффективности слияния.

12.9.1. Анализ

Алгоритм Dynix позволяет эффективно выделять память для многопроцессорных систем разделения памяти. Он поддерживает стандартный интерфейс System V и обеспечивает обмен участками памяти между распределителем и страничной системой. Кэширование на уровне процессоров уменьшает состязательность при глобальной блокировке, а двунаправленные связанные списки свободных объектов способствуют быстрому обмену буферами между процессорным и глобальным уровнем.

Интересно сравнить технологию слияния системы Dynix с зональным распределителем ОС Mach. Алгоритм Mach использует методику «пометки-и-подметания» (mark-and-sweep), где каждый раз происходит последовательное сканирование всего пула памяти. Это действие требует большого количества вычислений и вследствие этого производится в фоновом режиме. В системе Dynix всякий раз при освобождении блоков на уровне слияния на страницу изменяются и страничные структуры данных. Когда все буферы страницы освобождаются, страница может быть возвращена страничной системе. Это происходит в фоновом режиме как часть операции освобождения, что дает лишь небольшую прибавку ко времени ее выполнения, следовательно технология Dynix защищена от уменьшения производительности при наихудших условиях.

Результаты измерений [11] показали, что на однопроцессорной машине алгоритм Dynix быстрее алгоритма Мак-Кьюзика—Кэрелса в 3–5 раз. Увеличение быстродействия еще более заметно на многопроцессорных машинах (сотни и даже тысячи раз на 25 процессорах). Однако эти оценки были получены по сценарию, наиболее «легкому» для системы, при котором выделение памяти происходит из кэша аппаратного уровня. Измерения не проводились для более общего случая.

12.10. Составной распределитель системы Solaris 2.4

Составной (от англ. slab — кусок, «плита») распределитель был представлен в системе Solaris 2.4. Он позволил избавиться от многих проблем производительности, проигнорированных создателями других алгоритмов, описанных в этой главе. В результате методика показала наилучшую производительность и оптимальное использование памяти по сравнению со всеми другими реализациями. При разработке распределителя в первую очередь внимание было

обращено на три аспекта: повторное использование объектов, применение аппаратных кэшей и рабочую площадку распределителя.

12.10.1. Повторное использование объектов

Ядро использует распределитель для создания различных временных объектов, таких как индексные дескрипторы, структуры proc и сетевые буферы. Последовательность действий над объектом сводится к нижеперечисленным операциям.

1. Выделение памяти.
2. Создание (инициализация) объекта.
3. Использование объекта.
4. Уничтожение объекта.
5. Освобождение памяти.

Объекты ядра обычно являются достаточно сложными и содержат в себе дополнительные объекты, такие как счетчики ссылок, заголовки связанных списков, взаимные исключения и условные переменные. В момент создания объектов их поля устанавливаются в некое фиксированное, *начальное* состояние. При уничтожении объектов распределитель снова обращается к этим объектам и, как правило, перед освобождением памяти восстанавливает начальное состояние.

Приведем пример. Объект vnode содержит заголовок связанного списка резидентных страниц. При инициализации vnode список является пустым. Во многих реализациях UNIX [3] ядро производит сброс vnode только после того, как все его страницы были удалены из памяти. Следовательно, перед освобождением vnode (на стадии уничтожения) его связанный список снова становится пустым.

Если ядро заново использует объект для другого объекта vnode, ему не нужно повторно инициализировать заголовок связанного списка, так как это уже было выполнено на стадии уничтожения. Тот же принцип относится и к другим инициализированным полям. Например, ядро размещает объекты с первоначальным значением счетчика ссылок, равным единице, и удаляет их при освобождении последней ссылки, при этом счетчик ссылок снова становится равным единице. Взаимные исключения первично устанавливаются в *разблокированное* состояние, при освобождении для таких объектов блокировка также должна быть отменена.

Эти примеры показывают преимущества кэширования и повторного использования объектов по сравнению с выделением и инициализацией областей памяти. Кэши объектов эффективно используют доступное пространство, так как методика не прибегает к округлению до следующей степени числа 2. Зональный распределитель (раздел 12.8) также базируется на кэшировании объектов и использует память с высокой эффективностью. Он, однако, не из-

меняет состояние объектов, что не позволяет избавиться от дополнительных действий по их повторной инициализации.

12.10.2. Применение аппаратных кэшей

Традиционные распределители памяти порождают неочевидную, но очень важную проблему при использовании аппаратного кэша. Многие процессоры обладают кэшем данных первого уровня небольшого объема (размер кэша равен некоторой степени числа 2, более подробно см. в разделе 15.13). Адрес ячейки кэша вычисляется в MMU по следующей формуле:

```
cache location = address % cash size; // адрес ячейки кэша = адрес % размер кэша
```

Если в адресе присутствует отсылка на аппаратный кэш, сначала проверяется его адрес ячейки на предмет наличия данных. Если данные отсутствуют, на аппаратном уровне происходит сброс памяти в кэш, тем самым перезаписывается его прежнее содержимое.

Обычные распределители памяти (например, алгоритм Мак-Кьюзика—Кэрелса или методика близнецов) округляют размер выделяемой памяти до следующей по счету степени числа 2 и возвращают объекты этого размера. Более того, большинство объектов ядра обладают некоторыми важными, часто используемыми полями, расположенными ближе к их началу.

Эффект, обусловленный этими двумя факторами, ощутим. Представим, к примеру, реализацию, где индексный дескриптор имеет размер 300 байтов, 48 из которых являются часто востребуемыми. Ядро выделит для этого объекта 512-байтовый буфер, выравненный по 512-байтовой границе. Из них только 48 используются часто, то есть всего 9%.

В результате получается, что часть аппаратного кэша, близкого к 512-байтовой границе, заполнена важным содержимым, в то время как оставшаяся его часть используется редко. В приведенном примере индексные дескрипторы фактически нуждаются лишь в 9% кэша. Похожий расклад свойственен и другим объектам ядра. Такая аномалия в распределении буферных адресов приводит к неэффективному использованию аппаратного кэша, и, следовательно, низкой производительности операций с памятью.

Описанная проблема становится еще более заметной на машинах, где доступ к памяти осуществляется по нескольким главным шинам. Например, SPARCcenter 2000 [5] чередует передачу данных размером по 256 байтов по двум шинам. В случае последнего примера с индексными дескрипторами применительно к такой машине большинство операций доступа оккупирует шину 0, что в результате приведет к несбалансированному использованию шин.

12.10.3. Рабочая площадка распределителя

Рабочая (или опорная, footprint) площадка распределителя — это часть аппаратного кэша и *буфера ассоциативной трансляции* (translation lookaside

buffer, TLB), перезаписываемых самим распределителем. Более подробно о TLB см. в разделе 13.3.1. Распределители памяти, использующие карты ресурсов или алгоритмы близнецов, должны проверять сразу несколько объектов для нахождения подходящего буфера. Такие объекты обычно расположены в различных областях памяти, физически удаленных друг от друга. Следовательно, они чаще всего отсутствуют в кэше и TLB, что уменьшает производительность распределителя. Еще существеннее то, что при осуществлении доступа к памяти происходит перезапись активных вхождений кэша и TLB, что требует их повторного сброса из основной памяти.

Некоторые распределители, такие как Мак-Кьюзика—Кэрелса или зональный, обладают малой опорной площадкой, поскольку способны оценивать корректность области памяти путем простых вычислений и просто удаляют буфер из соответствующего списка свободных буферов. Для управления своим «следом» в кэше составной распределитель применяет такую же методику.

12.10.4. Структура и интерфейс

Составной распределитель является одним из вариантов зонального метода и организован как набор кэшей объектов. Каждый кэш содержит объекты одного типа. Следовательно, в системе существует один кэш для объектов vnode, один — для структур proc и т. д. Обычно ядро размещает объекты в соответствующих им кэшах и после производит их освобождение. Однако технология обладает механизмом, позволяющим предоставлять дополнительную память для кэша и возвращать обратно лишнюю.

Концептуально каждый кэш поделен на две составляющие (см. рис. 12.10) — *внешнюю* (front end) и *внутреннюю* (back end). Первая часть взаимодействует с клиентом памяти. Клиент получает построенные объекты из кэша и возвращает ему разрушенные объекты. Внутренняя часть взаимодействует с распределителем страничного уровня, занимаясь обменом «кусками» памяти при изменении ее конфигурации.

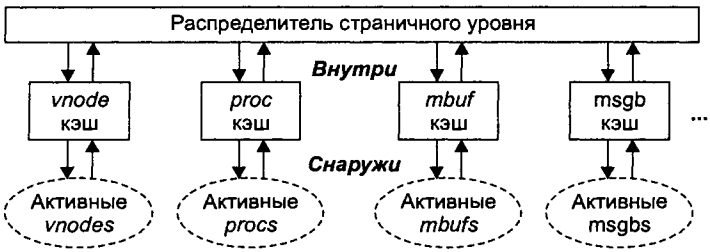


Рис. 12.10. Структура составного распределителя памяти

Подсистема ядра инициализирует кэш для управления объектами определенного типа с помощью функции

```
cachep = kmem_cache_create (name, size, align, ctor, dtor);
```

где name — символьная строка, описывающая объект, size — размер объекта в байтах, align — параметр выравнивания объектов, а ctor и dtor — указатели на функции построения и разрушения объектов соответственно. Функция возвращает указатель на кэш для данного объекта.

После этого ядро системы может заказывать объект в кэше посредством вызова

```
objp = kmem_cache_alloc (cachep, flags);
```

или освобождать его при помощи

```
kmem_cache_free (cachep, objp);
```

Эти команды не создают или уничтожают объект при повторном их использовании. Следовательно, ядро должно восстанавливать объект в *изначальном* состоянии при освобождении. Как уже описывалось в разделе 12.10.1, эти действия чаще всего производятся автоматически и не требуют проведения дополнительных операций.

Если в кэше не остается больше свободной памяти, происходит вызов функции kmem_cache_grow() для запроса области памяти от страничного распределителя и создания в ней объектов. Эта область состоит из нескольких последовательных страниц, составляющих в кэше монолитный участок. Она содержит достаточно памяти для поддержания нескольких копий объекта. Кэш использует небольшую часть области для управления памятью и разделяет свою оставшуюся часть на буферы, равные размеру объекта. После этого он инициализирует объекты путем вызова их конструкторов (операции, указанной в аргументе ctor вызова kmem_cache_create) и добавляет их в кэш.

Если распределителю страничного уровня необходимо вернуть себе часть памяти, для этой цели вызывается операция kmem_cache_reap(). Функция находит области памяти, в которых все объекты свободны, разрушает такие объекты (путем вызова операции-деструктора, указанной в аргументе dtor вызова kmem_cache_create) и затем производит удаление области из кэша.

12.10.5. Реализация алгоритма

Составной распределитель использует для объектов большого и маленького размера разные методики управления. К небольшим объектам относятся те из них, которые помещаются в одну страницу. Распределитель делит каждую область-«плиту» на три части: структуру kmem_slab, набор объектов и некоторое количество неиспользуемого пространства (см. рис. 12.11). Структура kmem_slab занимает 32 байта и располагается в конце этой области. Дополнительные четыре байта в конце каждой плиты служат для хранения указателя на список свободных объектов. Неиспользуемым пространством является некоторое количество памяти, остающееся незадействованным после создания максимально возможного количества объектов в области. Например, если

размер индексного дескриптора равен 300 байтам, область размером 4096 байтов может содержать 13 дескрипторов, 104 байта останутся неиспользованными (вспомним, что некоторое место отводится под структуру kmem_slab и указатели на списки свободных объектов). Это пространство разделяется на две части, о чем более подробно мы расскажем чуть позже.

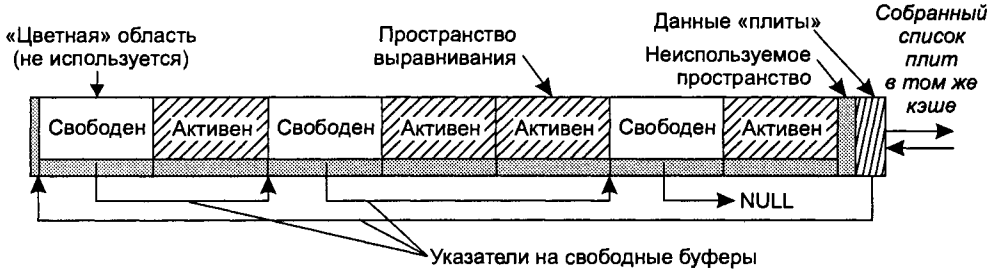


Рис. 12.11. Организация области для размещения объектов малой величины

Структура kmem_slab содержит счетчик используемых объектов, а также указатели на элемент двунаправленного связанного списка областей, расположенных в том же кэше, и указатель на первый свободный объект в области. Каждая область поддерживает собственный связанный список свободных буферов, информация о связи с которым хранится в четырехбайтовых полях, замыкающих каждый объект. Такое поле необходимо только для свободных объектов и должно различаться от самого объекта с целью невозможности его перезаписи.

Неиспользуемое пространство области делится на две части: «цветную» (slab coloring area) в начале и просто остаток непосредственно перед структурой kmem_slab. «Цветные» области получают различные размеры из соображений выравнивания. В примере, приведенном ранее, если индексный дескриптор требует 8-байтового выравнивания, он будет включать в себя 14 «цветных» областей различной длины (от 0 до 104 байтов при 8-байтовом инкременте). Это позволяет более удобно распределить стартовые смещения объектов класса, что приведет к сбалансированному и эффективному использованию аппаратных кэшей и шин памяти.

Ядро производит предоставление объекта посредством удаления первого элемента из списка свободных буферов и последующего увеличения счетчика используемых объектов. При освобождении объекта для идентификации области-плиты необходимо выполнить простейшую математическую операцию:

```
slab address = object address % slab size; // адрес плиты = адрес объекта %
размер плиты
```

Затем происходит возврат объекта в список свободных буферов области и уменьшение счетчика на единицу.

При значении счетчика используемых объектов, равном нулю, область является свободной или готовой к повторному использованию. Кэш соединяет все области в частично отсортированном двунаправленном связанном списке. В голове этого списка хранятся полностью активные области (в которых используются все объекты), в середине — частично используемые, а свободные области располагаются в его конце. Кэш также поддерживает указатель на первую область, имеющую свободный объект. Выделение памяти происходит из нее. Отсюда следует, что кэш не станет выделять объекты из полностью свободных областей до тех пор, пока не заполнятся все частично используемые области. Если страничному распределителю необходимо получить обратно часть памяти, то он произведет проверку конца списка и удалит свободные области.

Области больших размеров

Описанная реализация не позволяет эффективно использовать доступное пространство для объектов большой величины, занимающих сразу несколько страниц памяти. Для таких объектов выделение структур данных управления областью происходит из отдельного участка памяти (другого объекта кэша). Кроме структуры kmem_slab для таких объектов поддерживается еще одна, дополнительная структура kmem_bufctl. Она содержит связи со списком свободных объектов, указатель на kmem_slab и указатель на сам объект. Большая область также поддерживает таблицу хэширования для возможности обратного преобразования в структуру kmem_bufctl.

12.10.6. Анализ

Составной распределитель является прекрасно сконструированным, мощным средством выделения памяти. Он эффективно использует пространство, так как потенциальные потери ограничены структурой kmem_slab, полем связи объектов и неиспользованным участком, размер которого меньше одного объекта области. Большинство запросов обслуживается с максимальной быстротой путем удаления объектов из списка свободных буферов и изменения значения счетчика используемых объектов. Применение схемы «цветных» секций приводит к оптимальному использованию аппаратного кэша и шины памяти, что увеличивает общую производительность системы. Методика также имеет малую рабочую площадку в кэше, так как при обработке большинства запросов происходит доступ только к одной области.

Алгоритм сбора мусора является более простым по сравнению с зональным распределителем памяти, несмотря на то, что обе методики основаны на одних и тех же принципах. Затраты на сбор мусора минимальны, так как распределены по всем запросам: при обработке каждой операции изменяется значение счетчика используемых объектов. Операция возвращения памяти немного увеличивает нагрузку на систему, так как ей приходится производить сканирование различных кэшей с целью обнаружения свободных областей.

Максимально возможное влияние на производительность пропорционально общему количеству кэшей (но не количеству областей).

Среди недостатков составного распределителя следует указать на издержки управления памятью, связанные с тем, что для каждого типа объектов применяется отдельный кэш. Для обычных классов объектов, расположенных в кэше большого объема и часто используемых, издержки являются незначительными. Однако для малых, редко востребуемых объектов, потери часто являются абсолютно неприемлемыми. Вышеописанная проблема относится и к зональному распределителю памяти, используемому в ОС Mach, и решается путем поддержки набора буферов размером, равным некоторой степени числа 2, в которых размещаются нестандартные объекты, не нуждающиеся в собственном кэше.

Составной распределитель не должен отказываться от преимуществ дополнительных процессорных кэшей, наподобие используемых в Dynix. Это упоминается в работе [4] как одно из возможных дополнительных средств для будущих реализаций.

12.11. Заключение

При создании общецелевого распределителя памяти необходимо учитывать множество различных факторов: он должен быть быстрым, простым в использовании и уметь эффективно управлять памятью. В этой главе мы рассмотрели несколько технологий распределения памяти и проанализировали их преимущества и недостатки. Распределитель на основе карты ресурсов является единственным поддерживающим частичное освобождение объекта. Однако используемые в нем методики последовательного поиска приводят к снижению производительности, неприемлемому для большинства приложений. Распределитель Мак-Кьюзика–Кэрелса обладает простым интерфейсом с привлечением стандартных вызовов malloc() и free(). Вместе с тем в нем отсутствует механизм слияния буферов и возврата участков памяти, ранее запрошенных у распределителя страничного уровня. Методика близнецов, наоборот, производит постоянное слияние и деление буферов с целью удовлетворения изменяющихся потребностей к памяти. Производительность ее работы, как правило, низка, особенно при частом вызове операции объединения буферов. Зональный распределитель является весьма быстрым, но обладает недостаточно эффективными механизмами сбора мусора.

Dynix и составной распределитель вобрали в себя достоинства всех перечисленных выше методов. В Dynix применяется методика степеней двойки, сочетающаяся с использованием процессорного кэша и быстрым сбором мусора (на уровне слияния в страницу). Составной распределитель представляет собой модифицированный зональный алгоритм. Увеличение производительности достигается за счет повторного использования объектов и сбалансированного распределения адресов памяти. В этом методе применяется простой

алгоритм сбора мусора, не самым худшим образом влияющий на производительность. Как уже упоминалось ранее, добавление поддержки аппаратных кэшей в алгоритм позволило реализовать на его основе прекрасный распределитель памяти.

В таблице 12.1 показаны результаты ряда экспериментов [4] по сравнению составного распределителя с методикой, реализованной в системе SVR4 и алгоритмом Мак-Кьюзика—Кэрелса. Опыты также показали, что повторное использование объектов уменьшает время, необходимое на их размещение и инициализацию, в 1,3–5,1 раза, в зависимости от типа объекта. Это преимущество является приятным дополнением к улучшенным показателям времени выделения памяти, охарактеризованным таблицей.

Таблица 12.1. Измерения производительности популярных распределителей памяти

	SVR4	Мак-Кьюзика—Кэрелса	Составной распределитель
Среднее время, требуемое для alloc и free, мкс	9,4	4,1	3,8
Общая фрагментация (потери памяти)	46%	45%	14%
Производительность по тесту Kenbus (количество сценариев, выполненное за 1 мин)	199	205	233

Многие методики, рассмотренные в этой главе, годятся для новых реализаций распределителей памяти прикладного уровня. Однако требования к таким программам выдвигаются несколько особые, поэтому хороший распределитель памяти ядра может иметь плохую производительность на прикладном уровне и наоборот. Распределители прикладного уровня обычно управляют большими объемами (виртуальной) памяти, практически ничем не ограничиваемой для большинства приложений. Следовательно, для этого применения памяти слияние буферов и выравнивание адресов не являются настолько критичными моментами по сравнению со скоростью выделения и освобождения памяти. Для прикладного уровня также важны простота и стандартность предлагаемого интерфейса взаимодействия, поскольку таковой обычно используется совершенно отличающимися друг от друга приложениями, созданными разными программистами. Некоторые реализации распределителей памяти прикладного уровня описаны в [9].

12.12. Упражнения

1. Чем отличаются требования, предъявляемые к распределителю памяти ядра и распределителю памяти прикладного уровня?
2. Назовите максимальное количество вхождений карты ресурсов, используемое для обработки ресурса, имеющего `N` элементов.

3. Создайте программу, измеряющую использование памяти и производительность распределителя на основе карт ресурсов (при помощи специально сгенерированной последовательности запросов). Сравните с ее помощью варианты выделения участков: лучшего, худшего и первого подходящего.
4. Предложите свою реализацию функции free() для распределителя Мак-Кьюзика—Кэрелса.
5. Создайте процедуру scavenge(), выполняющую слияние свободных страниц в распределителе Мак-Кьюзика—Кэрелса и освобождение их для распределителя страничного уровня.
6. Реализуйте простейший алгоритм близнецов, управляющий 1024-байтовой областью памяти с минимальным размером участка, равным 16 байтам.
7. Определите проследовательность запросов, которые могут привести к наихудшему варианту поведения простого алгоритма близнецов.
8. Как отразятся на значении переменных N, A, L, G и *допуска* следующие события, происходящие в алгоритме отложенного слияния системы SVR4 (см. раздел 12.7):
   - буфер освобождается при значении допуска больше двух;
   - перераспределяется задержанный буфер;
   - выделяется не задержанный буфер (нет задержанных буферов);
   - буфер освобождается при допуске, равном 1, при этом ни один из свободных буферов не может быть подвергнут слиянию, так как все близнецы в данный момент используются;
   - буфер подвергнут слиянию со своим близнецом.
9. Можно ли преобразовать методики распределения памяти так, чтобы они поддерживали список свободных объектов для каждого процессора в случае использования в многопроцессорных системах (см. технологию Dynix)? Какие из алгоритмов нельзя адаптировать для этой методики и почему?
10. Почему в составном распределителе используются две различные реализации управления большими и малыми объектами?
11. Какой распределитель из описанных в этой главе позволяет клиенту освобождать часть выделенного ранее блока?
12. Какой из распределителей способен отклонить запрос на выделение даже в том случае, если ядро в данный момент обладает блоком памяти, размером достаточным для удовлетворения такого запроса?

12.13. Дополнительная литература

1. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
2. Barkley, R. E., and Lee, T. P., «A Lazy Buddy System Bound By Two Coalescing Delays per Class», Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, Dec. 1989, pp. 167–176.
3. Barkley, R. E., and Lee, T. P., «A Dynamic File System Inode Allocation and Reclaim Policy», Proceedings of the Winter 1990 USENIX Technical Conference, Jan. 1990, pp. 1–9.
4. Bonwick, J., «The Slab Allocator: An Object-Caching Kernel Memory Allocator», Proceedings of the Summer 1994 USENIX Technical Conference, Jun. 1994, pp. 87–98.
5. Cekleov, M., Frailong, J. M., and Sindhu, P., «Sun–4D Architecture», Revision 1.4, Sun Microsystems, 1992.
6. Digital Equipment Corporation, «Alpha Architecture Handbook», Digital Press, 1992. Digital Equipment Corporation, DEC OSF/1 Internals Overview — Student Workbook, 1993.
7. Knuth, D., «The Art of Computer Programming», Vol. 1, Fundamental Algorithms, Addison-Wesley, Reading, MA, 1973.
8. Korn, D. G., and Vo, K. P., «In Search of a Better Malloc», Proceedings of the Summer 1985 USENIX Technical Conference, Jun. 1985, pp. 489–505.
9. Lee, T. P., and Barkley, R. E., «A Watermark-Based Lazy Buddy System for Kernel Memory Allocation», Proceedings of the Summer 1989 USENIX Technical Conference, Jun. 1989, pp. 1–13.
10. McKenney, P. E., and Slingwine, J., «Efficient Kernel Memory Allocation on Shared-Memory Multiprocessors», Proceedings of the Winter 1993 USENIX Technical Conference, Jan. 1993, pp. 295–305.
11. McKusick, M. K., and Karels, M. J., «Design of a General-Purpose Memory Allocator for the 4.3BSD UNIX Kernel», Proceedings of the Summer 1988 USENIX Technical Conference, Jun. 1988, pp. 295–303.
12. Peterson, J. L., and Norman, T. A., «Buddy Systems», Communications of the ACM, Vol. 20, No. 6, Jun. 1977, pp. 421–431.
13. Sciver, J. V., and Rashid, R. F., «Zone Garbage Collection», Proceedings of the USENIX Mach Workshop, Oct. 1990, pp. 1–15.
14. Stephenson, C. J., «Fast Fits: New Methods for Dynamic Storage Allocation», Proceedings of the Ninth ACM Symposium on Operating Systems Principles, Vol. 17, No. 5, 1983, pp. 30–32.

Глава 13
Виртуальная память

13.1. Введение

Одной из главных функций операционной системы является эффективное управление ресурсами памяти. В каждой вычислительной машине имеется высокоскоростная память с произвольным доступом (или оперативная память, RAM), иногда называемая основной памятью, физической памятью, или просто памятью. Скорость доступа к оперативной памяти сравнима с несколькими тактами процессора. Программы могут напрямую обращаться к данным или участкам кода, находящимся в оперативной памяти компьютера. Микросхемы памяти RAM относительно дороги, а объем памяти всегда недостаточен. В системе используется большое количество разнообразных вторичных носителей информации для хранения данных, не помещающихся в основную память (обычно это жесткие диски или серверные машины в сети). Скорость обращения к таким устройствам на несколько порядков ниже по сравнению со скоростью доступа к оперативной памяти, а операции доступа требуют определенных действий от части операционной системы. Подсистема управления памятью ядра отвечает за распределение информации между оперативной памятью и вторичными устройствами хранения. Ядро взаимодействует напрямую с аппаратным компонентом, называемым *блоком управления памятью* (memory management unit, MMU), отвечающим за получение данных из оперативной памяти и помещение данных в нее.

Работа любой операционной системы без необходимости управления памятью значительно упрощается. В элементарном случае система будет поддерживать одновременное выполнение только одной программы, постоянно загруженной по фиксированному адресу в памяти. Это упрощает задачу связывания и загрузки, а также освобождает аппаратуру от необходимости какого-либо преобразования адресов. Любая адресация будет относиться только к физическим адресам, а программа будет иметь в своем распоряжении все ресурсы машины (конечно, используемые ею совместно с операционной системой). Данный метод является наиболее быстрым и эффективным способом выполнения единственной программы.

Такой вариант можно обнаружить в системах реального времени, основанных на небольших микропроцессорах (например, программа «прошита» в ПЗУ). Для систем общего применения недостатки этого метода очевидны. Во-первых, размер программы ограничен объемом памяти, следовательно, крайне ограничена возможность выполнения больших программ. Во-вторых, если в память загружена только одна программа, при необходимости ожидания ввода-вывода простаивает вся система. Если какая-либо система была оптимизирована для случая выполнения единственной небольшой программы, то, скорее всего, она будет совершенно непригодной для многозадачных сред.

Из вышесказанного следует, что ОС необходима некоторая форма управления памятью. Опишем вкратце список возможностей, ожидаемых от любой операционной системы:

- выполнение программ, имеющих размер больший, чем объем физической памяти. В идеальном варианте система должна уметь выполнять программы любого размера;
- выполнение частично загруженных программ, что уменьшает время их первоначальной загрузки;
- размещение в памяти одновременно более чем одной программы, что увеличивает коэффициент использования процессора;
- возможность выполнения перемещаемых в памяти программ, которые могут располагаться в любом участке памяти и перемещаться из одного месторасположения в другое в течение выполнения;
- возможность создания машинно-независимых кодов, то есть *априори* между программой и физической памятью не должно быть связи;
- освобождение программистов от обязанности выделения ресурсов памяти и управления ими;
- поддержка совместного использования ресурсов, например, если два процесса загружают одну и ту же программу, они должны иметь возможность использовать сообща одну и ту же копию программного кода.

Поставленные задачи практически реализуемы при использовании *виртуальной памяти* [9]. Она позволяет дать приложению иллюзию доступности большого объема памяти, в то время как реально компьютер может обладать лишь небольшим объемом оперативной памяти. Это требует определения абстракции *адресного пространства*, отличного от физического местонахождения памяти. Программа создает ссылки на коды и данные в своем адресном пространстве, такие адреса должны преобразовываться в адреса ячеек оперативной памяти. Передача информации в основную память для использования ее программой и выполнение трансляции адресов при каждом доступе к памяти вынуждает к совместным действиям как программную, так и аппаратную часть компьютера.

Виртуальная память требует некоторых издержек. Таблицы трансляции адресов и другие структуры данных, создаваемые для управления памятью, уменьшают количество физической памяти, доступное программам. Ко времени выполнения каждой инструкции прибавляются затраты на преобразование адресов, которые становятся значительной частью операции, если она требует дополнительного обращения к памяти. Если процесс пытается получить доступ к странице, не загруженной в основную память, система среагирует тем, что поместит эту страницу в оперативную память, что может потребовать относительно медленных операций дискового ввода-вывода. Все действия с памятью занимают значительную часть процессорного времени (примерно 10% на загруженных системах). Реальное количество используемой памяти уменьшается из-за *фрагментации*. Например, в страничных системах полезные данные занимают только часть страницы, остальное же пространство не используется. Все перечисленные факторы воздействуют на эффективность разработок, а это, в свою очередь, оказывает влияние на их производительность и функциональность.

13.1.1. Управление памятью в «каменном веке» вычислительной техники

Первые реализации системы UNIX (до седьмой версии) выполнялись на машине PDP-11 с 16-разрядной архитектурой и адресным пространством, равным 64 Кбайт. Некоторые модели PDP поддерживали отдельное пространство для размещения инструкций и данных, однако в совокупности адресное пространство любого процесса не могло превышать 128 Кбайт. Это ограничение привело к развитию различных технологий *программных оверлеев* (software overlay), как для прикладных программ, так и для самого ядра системы [7]. Оверлеи позволяют повторно использовать память путем перезаписи больше не нужного части адресного пространства программы. Например, после загрузки и начала функционирования ОС уже не требуется код инициализации системы. Занимаемое им пространство может быть выделено для других частей программы. Оверлейные схемы требуют от разработчика приложения указания определенных последовательностей действий, позволяющих их реализацию. Программист должен не только тщательно изучить особенности своего продукта, но и машины, на которой он будет работать. Программы, использующие оверлеи, плохо поддаются переносу на другие аппаратные платформы, так как оверлейная схема сильно зависит от структуры физической памяти компьютера. Даже простое добавление новой памяти в систему требует внесения изменения в такие программы.

Механизмы управления памятью в ранних версиях UNIX ограничивались возможностями *свопинга* (swapping) (рис. 13.1). Процессы загружались в оперативную память целиком друг за другом. В один момент времени

в физической памяти машины могло поместиться некоторое небольшое число процессов, а система являлась для них совместно используемым ресурсом. Если требовалось выполнить другой процесс, то один из существующих процессов должен быть выгружен из памяти. Такой процесс копировался в заранее определенную *область свопинга* (swapping partition), расположенную на диске. Некоторое *пространство свопинга* (swap space) выделялось для каждого процесса на стадии его создания, что гарантировало доступность области свопинга при необходимости ее использования.

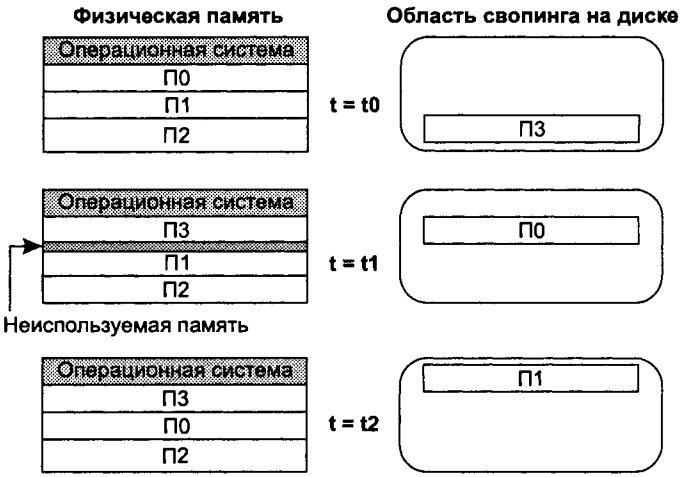


Рис. 13.1. Управление памятью на основе технологии свопинга

Технология *загрузки страниц по запросу* (demand paging) появилась в системе UNIX после создания VAX-11/780 в 1978 году. Эта машина обладала 32-разрядной архитектурой, 4-гигабайтовым адресным пространством и аппаратной поддержкой упомянутой технологии, то есть операционная система 3BSD стала первой реализацией UNIX, ее поддерживающей [1], [2]. А к середине 80-х годов все имеющиеся тогда версии UNIX уже обеспечивали загрузку страниц по запросу в качестве основной методики управления памятью, технология свопинга же отошла на второй план.

В системах загрузки страниц по запросу память и адресное пространство процесса поделены на страницы фиксированного размера, которые помещаются в память или выгружаются по мере необходимости. Страницу физической памяти часто называют *страничным фреймом* (page frame) или *физической страницей* (physical page). В один момент времени могут выполняться сразу несколько процессов, при этом в физической памяти располагается лишь некоторая часть страниц каждого из них (см. рис. 13.2). Каждая выполняемая программа считает себя единственной в системе. Программные адреса *виртуальны* и расчленяются машиной на номер страницы и смещение на этой странице. Аппаратная часть совместно с операционной системой произ-

водит преобразование номера виртуальной страницы в номер физического страничного фрейма и затем обращается к соответствующей ячейке памяти. Если требуемой страницы нет в оперативной памяти, ее необходимо туда загрузить. В идеальном случае ни одна из страниц не может быть помещена в память до тех пор, пока она не станет востребована (то есть пока на эту страницу не будет сделана ссылка). Большинство современных систем UNIX производят *упреждающее* помещение некоторого количества страниц в память (anticipatory paging), загружая те страницы, которые по мнению системы скоро будут затребованы.

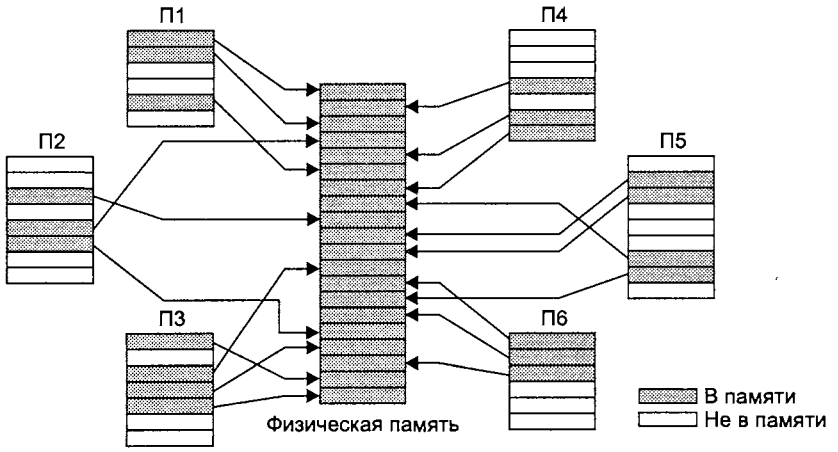


Рис. 13.2. В физической памяти компьютера обычно располагается лишь малое количество страниц каждого процесса

Схема загрузки страниц по запросу может использоваться как совместно с технологией свопинга, так и в качестве ее замены. Перечислим некоторые ее преимущества:

- размер программы ограничивается только максимальным объемом виртуальной памяти, который на 32-разрядных машинах равен 4 гигабайтам;
- первичная загрузка программы происходит быстро, поскольку для начала работы программы не требуется полностью помещать ее в память компьютера;
- одновременно в системе может быть загружено сразу несколько программ, так как в определенный момент времени лишь небольшая часть страниц каждой из них должна находиться в памяти;
- перемещение отдельных страниц памяти намного проще для системы, чем свопинг процессов или сегментов целиком.

Еще одним важным аспектом, о котором нужно обязательно рассказать, является *сегментация*. Технология сегментации позволяет поделить адресное пространство процесса на несколько сегментов. Любой адрес программы

содержит идентификатор сегмента (segment ID) и смещение от его начала. Каждый сегмент может обладать собственными установками защиты (чтение/запись/выполнение). Сегменты загружаются в оперативную память целиком, каждый из них описывается при помощи дескриптора, содержащего физический адрес, по которому загружен сегмент (базовый адрес), размер и сведения о защите. Аппаратная часть проверяет границы сегмента при каждой операции доступа, что исключает факты повреждения процессом соседнего сегмента. Загрузка и выгрузка (свопинг) может быть проведена на уровне отдельных сегментов, а не всей программы.

Сегментация в иных случаях используется совместно с разбивкой на страницы [3], что в результате дает гибкий гибридный механизм управления памятью. В таких системах сегменты не обязательно должны располагаться в памяти непрерывно. Каждый из них имеет собственную карту адресации, при помощи которой производится преобразование смещений сегмента в физические ячейки памяти. Такую методику поддерживает, к примеру, архитектура *Intel 80x86*[1].

Создатели программ обычно предполагают, что адресное пространство процесса содержит области текста, данных и стека, чему способствует принятая в сегментах адресация. Несмотря на то, что в большинстве версий UNIX все три области описаны явно, поддерживается и абстракция высшего уровня, построенная на основе виртуально непрерывных страниц памяти. Технология сегментации не прижилась в основных вариантах UNIX, поэтому мы не будем обсуждать ее более подробно на страницах книги.

13.2. Загрузка страниц по запросу

Страничная система отвечает за выделение адресных пространств всех процессов и управлением ими. Она пытается оптимизировать использование физических ресурсов, к которым причисляются оперативная память и вторичные устройства хранения данных, стараясь обеспечить желаемый уровень функциональности при минимуме потерь. В этой главе, а также в следующих двух главах проводится сравнение различных архитектур виртуальной памяти. Начнем их описание с рассказа о некоторых фундаментальных определениях, имеющихся во всех реализациях систем загрузки страниц по запросу.

13.2.1. Требования к функциональности

При рассмотрении преимуществ архитектуры загрузки страниц по запросу необходимо учитывать и уровень функциональности различных реализаций. Основной целью системы является поддержка выполнения процесса в виртуальном адресном пространстве и проведение преобразования виртуальных

[1] В этой книге термин 80x86 (или x86) используется для обозначения общих свойств архитектур процессоров Intel 80386, 80486 и Pentium.

адресов в физические незаметно для процессов. Реализация поставленной цели должна как можно меньше влиять на системные ресурсы. Большинство остальных требований к системе исходят из основной цели. С этой точки зрения мы можем разработать более подробный набор требований к системе.

- **Управление адресным пространством.** Ядро должно выделять адресное пространство процессу на этапе выполнения fork и освобождать его при вызове exit. Если процесс производит системный вызов exec, ядро освобождает его старое адресное пространство и выделяет новое, относящееся уже к новой программе. Другие важные операции в адресном пространстве подразумевают изменение областей данных или стека, а также добавление новых областей (например, разделяемой памяти).
- **Преобразование адресов.** При выполнении любой инструкции, производящей доступ к памяти, диспетчеру MMU необходимо подставлять вместо виртуальных адресов, генерируемых процессом, физические адреса оперативной памяти. В системах загрузки страниц по запросу страница представляет собой единицу памяти, к которой относятся операции размещения, защиты и преобразования адресов. Виртуальный адрес преобразуется в номер виртуальной страницы и смещение внутри этой страницы. Номер виртуальной страницы пересчитывается в физический номер страницы при помощи *карт трансляции адресов*. Если инструкция пытается получить доступ к странице, отсутствующей в физической памяти, в системе возникнет исключительное состояние *страничной ошибки* (page fault). Обработчик исключительных состояний ядра исправит эту ошибку путем помещения необходимой страницы в оперативную память.
- **Управление физической памятью.** Физическая память является наиболее важным ресурсом компьютера, контролируемым подсистемой управления памятью. Память необходима как ядру, так и прикладным процессам, такие запросы должны обрабатываться как можно быстрее. Суммарное адресное пространство всех активных процессов системы, как правило, превышает объем физической памяти, в которой может находиться лишь ограниченный набор данных. Из этого следует, что система использует оперативную память как кэш необходимых данных. Ядро должно оптимизировать коэффициент использования кэша, а также гарантировать корректность и актуальность находящихся в нем данных.
- **Защита памяти.** Процесс не может производить доступ к страницам памяти или их изменение произвольно. Ядро должно оберегать собственные коды и данные от порчи прикладными процессами. В противном случае программа может случайно (или намеренно) повредить ядро. Из соображений защиты кодов и данных ядра их также необходимо обезопасить от возможности чтения прикладными процессами. Процессы не должны иметь права доступа к страницам, относящимся к другим

процессам. Часть адресного пространства процесса бывает защищена даже от него самого. Например, область текста процесса чаще всего изолирована от записи, так как иначе процесс в состоянии повредить ее. Правила защиты системной памяти реализованы в ядре системы. Они используют доступные аппаратные механизмы. Если ядро обнаруживает попытку обращения к запрещенному адресу памяти, оно информирует об этом процесс посылкой сигнала нарушения сегментации (segmentation violation, сигнал SIGSEGV)[1].

- **Разделение памяти.** Характеристики процессов UNIX и их взаимодействие друг с другом показывают необходимость коллективного использования части их адресных пространств. Например, все процессы одной программы могут разделять между собой единственную копию текстовой области программы. Процессы вправе произвести определенный запрос на совместное использование области памяти вместе с другими взаимодействующими процессами. Коды стандартных библиотек можно разделять точно таким же образом. Приведенные примеры относятся к совместному использованию ресурсов на высоком уровне. Однако существует и вариант разделения на низком уровне, относящийся к отдельным страницам памяти. Например, после выполнения fork предок и потомок могут разделять между собой единственную копию страниц данных и стека до тех пор, пока процессы не попытаются внести в них изменения.

  Эти и другие формы коллективного использования памяти положительно влияют на производительность системы, так как уменьшают количество конфликтов в физической памяти и позволяют избавиться от копирования внутри памяти и операций дискового ввода-вывода, необходимых для поддержки нескольких дубликатов одних и тех же данных. Решение о применяемой методике разделения памяти и ее реализация выносятся на уровень подсистемы управления памятью.

- **Отслеживание загрузки системы.** Обычно страничная система справляется с потребностями активных процессов. Однако иногда система может оказаться перегруженной. Если это происходит, процессы не получают достаточного количества памяти для размещения своих активных страниц и в результате этого не могут двигаться дальше. Загруженность страничной системы зависит от общего числа и размеров адресного пространства активных процессов, а также от конфигурации памяти. Операционной системе требуется следить за страничной подсистемой с целью обнаружения таких ситуаций и при необходимости совершать некоторые действия по их разрешению. Система может отреагировать на перегрузку путем запрета выполнения новых процессов или деактивации некоторого количества выполняющихся процессов.

[1] В некоторых случаях ядро посылает другой сигнал — SIGBUS (ошибка шины).

- **Другие средства.** К другим функциям системы управления памятью можно причислить поддержку файлов карт памяти, разделяемых библиотек и выполнение программ, расположенных на удаленных узлах.

Архитектура управления памятью значительно влияет на общую производительность системы. В любом случае, ее реализация должна быть восприимчива к потребности в быстродействии и расширяемости. Переносимость также важна, так как позволяет системе выполняться на различных машинах. Подсистема памяти обязана функционировать незаметно для пользователя. Программист же должен иметь возможность создавать коды без знания внутреннего устройства памяти.

13.2.2. Виртуальное адресное пространство

К адресному пространству процесса относятся все (виртуальные) ячейки памяти, к которым может ссылаться или обращаться программа. В любой момент времени адресное пространство совместно с контекстом регистров процессора отражает текущее состояние программы. Если процесс загружает новую программу путем вызова exec, ядро строит для нее новое адресное пространство. В архитектурах выделения страниц по запросу пространство процесса делится на страницы фиксированного размера. Страницы могут содержать несколько различных типов информации:

- текст;
- инициализированные данные;
- неинициализированные данные;
- измененные данные;
- стек;
- кучу;
- разделяемую память;
- разделяемые библиотеки.

Перечисленные типы данных отличаются друг от друга в вопросах защиты, методики инициализации и совместного использования процессами. Текстовые данные обычно доступны только для чтения, в то время как прочие, включая стек и кучу, доступны в режиме чтения-записи. Защита совместно используемых страниц обычно устанавливается при первоначальном выделении области памяти.

Текстовые страницы обычно разделяются между всеми процессами, выполняемыми одной программой. Страницы в разделяемой области памяти используются всеми процессами, обладающими возможностью присоединения этой области к своему адресному пространству. Совместно используемая библиотека может содержать как страницы с текстами, так и данными. Текстовые страницы разделяются между всеми процессами, имеющими до-

ступ к библиотеке. Страницы данных библиотеки не разделяются, каждый процесс получает их копию (некоторые реализации систем могут позволять разделять такие страницы до тех пор, пока в них не будут вноситься изменения).

13.2.3. Первое обращение к странице

Процесс может начать выполнение программы даже в том случае, если ни одной ее страницы нет в физической памяти. При обращении к каждой нерезидентной странице генерируется страничная ошибка, обрабатываемая ядром путем выделения свободной страницы и ее инициализации с размещением необходимых данных.

Методика инициализации не одинакова при первом и последующих обращениях к странице. При первом обращении происходит чтение страниц текста и инициализируемых данных из выполняемого файла. Страницы неинициализируемых данных заполняются нулями; следовательно, глобальные инициализируемые переменные автоматически получают значение ноль. Разделяемые страницы библиотек инициализируются из библиотечных файлов. Область u и стек ядра устанавливаются при создании процесса путем копирования соответствующих страниц родительского процесса.

Если процесс выполнит программу, уже загруженную другим процессом или недавно выполненную, часть (а иногда и все) его текстовых страниц может остаться в оперативной памяти или на устройствах быстрого доступа, таких как область свопинга (обсуждаемая в разделе 13.2.4). В этом случае система способна предотвратить избыточные операции загрузки страниц из выполняемого файла. Более подробно об этом свойстве системы можно прочесть в разделах 13.4.4 и 14.6.

13.2.4. Область свопинга

Общее пространство всех выполняемых программ часто превышает объем оперативной памяти, в которой хранится только часть страниц каждого процесса. Если процессу необходима страница, не загруженная в память, ядро выделяет для нее место путем выделения новой страницы или замещения содержимого старой.

В идеале необходимо заменять только те страницы, которые больше не понадобятся, например, те из них, которые относятся к завершившимся процессам. На практике такое не всегда возможно, следовательно, ядро иногда отзывает страницу, которая может понадобиться в будущем. В этом случае ядру необходимо сохранить копию такой страницы на вторичном носителе. В системах UNIX для хранения таких временных страниц используется специальная *область свопинга*, которая занимает один или несколько разделов диска. При первоначальной настройке системы такие разделы оставляются неформатированными и резервируются для свопинга. Они не могут использоваться для хранения файловых систем.

На рис. 13.3 показана схема перемещения страниц между физической памятью и различными вторичными устройствами. При осуществлении доступа к странице, сохраненной в области свопинга, ядро обрабатывает страничную ошибку путем чтения необходимой страницы из этой области. Для этого ядру необходимо поддерживать *карту свопинга*, описывающую местонахождение всех страниц, находящихся в области свопинга. Если страница должна быть удалена из памяти повторно, ее сохранение в области свопинга происходит только в том случае, когда ее содержимое не совпадает с предыдущей копией. Это бывает тогда, когда страница является «*грязной*» (dirty), то есть была модифицирована после последнего чтения из области свопинга. Следовательно, ядро нуждается в средстве, позволяющем распознавать подобные страницы. Его реализация может оказаться простой задачей для некоторых систем, в которых вхождения таблицы страниц имеют аппаратно-поддерживаемый бит «грязная». В другом случае ядро получает такую информацию иными способами, о которых вы можете справиться в разделе 13.5.3.

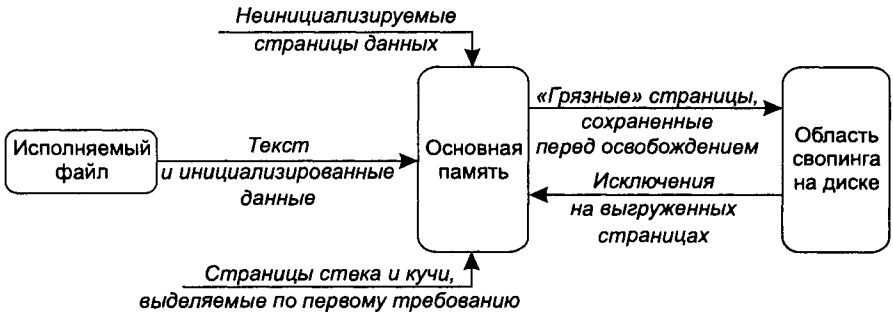


Рис. 13.3. Удаление страниц в физическую память машины и размещение их

Нет надобности хранить в области свопинга текстовые страницы, так как они могут быть восстановлены прямо из выполняемого файла. Однако в некоторых реализациях систем предусмотрено копирование текстовых страниц в область свопинга наравне со всеми остальными из-за соображений производительности. Карты свопинга обычно достаточно эффективны, ядро может находить страницу путем простого индексированного доступа к таблице, расположенной в памяти. Для обнаружения страницы в выполняемом файле ядру необходимо обратиться к файловой системе, которая проведет проверку индексного дескриптора и, возможно, некоторого количества блоков косвенной адресации (см. раздел 9.2.2). Эти действия занимают большой промежуток времени, поскольку для доступа к блокам косвенной связи требуются дополнительные дисковые операции. Однако помещение текстовых страниц в область свопинга также приводит к дополнительному дисковому вводу-выводу, который может происходить быстрее. Большинство современных систем не помещает текстовые страницы в область свопинга, читая их из файла при необходимости.

13.2.5. Карты преобразования адресов

Страничная система может использовать четыре различных типа карт трансляции адресов в целях реализации поддержки виртуальной памяти (рис. 13.4).

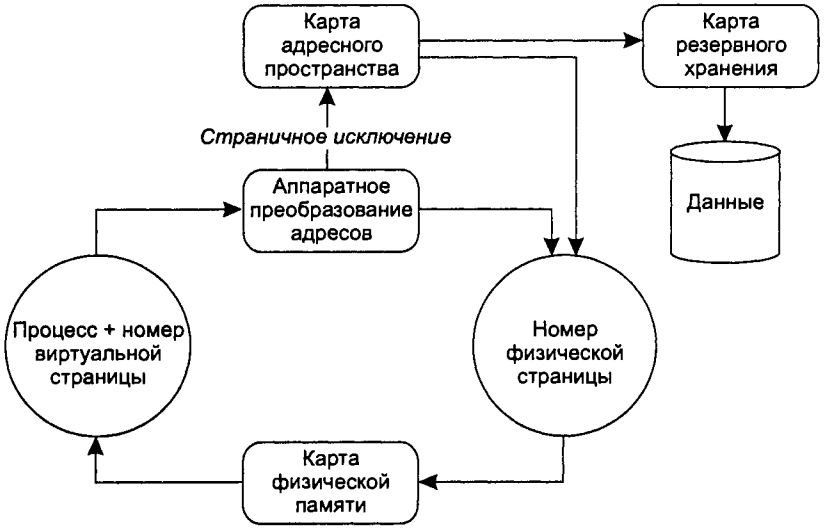


Рис. 13.4. Преобразование адресов

- **Аппаратное преобразование адресов.** Для выполнения каждой инструкции, осуществляющей доступ к памяти, аппаратной части необходимо транслировать виртуальный адрес программы в ячейку физической памяти. Любой компьютер обладает некоторым механизмом *аппаратного преобразования адресов*, следовательно, операционной системе не нужно участвовать в решении этой задачи. В разделе 13.3 проводится анализ трех примеров архитектур памяти, каждая из которых использует некоторую форму *буфера ассоциативной трансляции* (TLB) и *таблиц страниц*. Несмотря на то, что формат этих структур данных определяется аппаратной частью машины, за их конфигурирование и поддержание отвечает операционная система.

  Карты аппаратного преобразования адресов являются единственным типом структур данных, распознаваемых аппаратурой. Остальные карты, описываемые в этом разделе, используются только операционной системой.

- **Карта адресного пространства.** Если аппаратная часть не может преобразовать адрес, генерируется *страничная ошибка* (page fault). Это может произойти в том случае, если страница не находится в основной памяти машины или аппаратура не знает, как корректно произвести пересчет адреса. Решение проблемы возложено на обработчик ошибки

ядра, который либо помещает нужную страницу в основную память, либо использует действительное вхождение аппаратного преобразования адресов.

Карты, распознаваемые аппаратной частью машины, могут содержать неполную информацию об адресном пространстве процесса. Например, на MIPS R3000 аппаратура использует небольшой буфер TLB. Операционной системе в таких случаях необходимо поддерживать дополнительные карты, хранящие полное описание адресного пространства процессов.

- **Карта физической памяти.** Ядру часто нужно производить обратное преобразование и определять процесс, являющийся владельцем данной страницы в оперативной памяти, а также номер виртуальной страницы для нее. Например, при удалении активной страницы из памяти ядру необходимо обозначить недействительными любые преобразования, касающиеся ее. Для этого ядро находит соответствующее вхождение в таблице страниц и/или вхождение TLB. Если не выполнить такой процедуры, аппаратура будет считать, что страница по-прежнему находится в физической памяти. Таким образом, ядро поддерживает карту физической памяти для того, чтобы следить, какие данные находятся в каждой странице.
- **Карта внешней памяти.** Если обработчик ошибки не обнаруживает нужную страницу в оперативной памяти машины, он выделяет другую страницу и инициализирует ее либо путем заполнения нулями, либо чтением данных со вторичных носителей. В последнем случае страница должна быть получена из выполняемого файла, объектного файла разделяемой библиотеки или копии, находящейся в области свопинга. Такие объекты и составляют *внешнюю память* по отношению к страницам процесса. Ядро поддерживает карты для нахождения страниц, находящихся во внешней памяти.

13.2.6. Правила замещения страниц

При выделения места для новой страницы ядро должно удалить страницу, находящуюся в памяти в текущий момент времени. Правила замещения страниц служат для принятия решения о том, какую именно страницу следует удалить из памяти [5]. Идеальным кандидатом, конечно, является «мертвая» страница, которая больше не потребуется кому-либо (например, если она относится к завершенному процессу). В том случае, если таких страниц нет в памяти (или их количества недостаточно), ядро может выбрать правило либо *локального*, либо *глобального* замещения страниц. Правило локального замещения выделяет определенное количество страниц каждому процессу или группе взаимосвязанных процессов — если процессу необходима новая

страница, он должен заменить одну из своих собственных страниц. Если ядро выбирает другое правило, то оно может взять страницу любого процесса, используя глобальные критерии выбора.

Правила локального замещения необходимы в том случае, если важны гарантии выделения ресурсов определенным процессам. Например, системный администратор может выделить больший объем страниц наиболее важным процессам. С другой стороны, «глобальные» правила проще для реализации и являются наиболее подходящим вариантом для обычных систем разделения времени. В большинстве вариантов UNIX реализовано правило глобального замещения, но для каждого активного процесса резервируется некое минимальное количество резидентных страниц.

Для реализации «глобального» подхода ядру необходимо выбрать критерий, по которому принимается решение о страницах, хранимых в памяти. В идеале должны поддерживаться только те страницы, которые будут использованы в недалеком будущем. Этот набор страниц получил название *рабочего набора* (working set) процесса. Если заранее известно, что на те или иные страницы ожидается определенное количество ссылок в будущем, такой набор может быть точно определен (по крайней мере, теоретически). Однако на практике чаще всего дальнейшее поведение процесса неизвестно, поэтому необходимо эмпирическое изучение типичных процессов как руководство для реализации.

Практика показывает, что большинство процессов характеризуются *локальностью ссылок*, то есть существует тенденция процесса к сужению круга ссылок до небольшого набора страниц, который меняется очень медленно. Например, при выполнении функции все инструкции на странице (страницах) содержат эту функцию, через некоторое время процесс может начать выполнение другой функции, что изменит его рабочий набор. Сходным образом происходит использование данных: циклы, работающие с массивами и функциями, производящими некоторые операции над структурой, являются примером кода, показывающим локальность ссылок.

Практическим выводом из всех этих наблюдений является тот факт, что недавно затребованные страницы, скорее всего, подвергнутся доступу в ближайшем будущем повторно. Таким образом, хорошим вариантом аппроксимации для рабочего набора является набор недавно запрошенных страниц. Логика рассуждений приводит к неизбежности применения правила наименее частого использования (LRU) для замещения страниц — удаляются те страницы, к которым доступ производился наиболее давно. Подобные соглашения типичны для буферного кэша файловой системы, где схема доступа к файлам демонстрирует сходные тенденции. Однако в случае управления памятью правило LRU необходимо адаптировать в соответствии с практическими обстоятельствами, показанными в разделе 13.5.3.

И, наконец, ядру необходимо принимать решение об освобождении активных страниц. Здесь, например, можно разрешить просматривать страницы

для последующего удаления только в том случае, если активному процессу понадобится дополнительная память. Такой способ является неэффективным и отрицательно влияет на производительность системы. Лучшим выходом является поддержка набора свободных страниц и периодическое добавление всех страниц в этот набор, что позволяет сбалансировать загрузку страничной системы.

13.3. Аппаратные требования

Выполнение некоторых задач подсистемой управления памятью обуславливается возможностями аппаратной части. Такие задачи зависят от аппаратного компонента, называемого *блоком управления памятью* (Memory Management Unit, MMU), «расположенного» функционально между центральным процессором и основной памятью компьютера. Архитектура MMU сильно сказывается на устройстве подсистемы управления памятью в ядре ОС. Рассмотрим сначала работу устройства MMU в целом, а затем на трех специфических вариантах архитектур (x86, IBM RS/6000 и MIPS R3000) проанализируем, как влияют характеристики аппаратуры на построение ядра.

Основной задачей диспетчера MMU является преобразование виртуальных адресов. В большинстве систем таблицы трансляции адресов реализованы при помощи страничных таблиц, TLB или обоих методов. В этом разделе вы увидите описание страничных таблиц, о TLB мы расскажем далее. Обычно в системе поддерживается одна таблица страниц для адресов ядра и одна (или более) таблиц для описания адресного пространства каждого процесса. Таблица страниц представляет собой массив элементов, по одному на виртуальную страницу процесса. Индекс *элемента таблицы страниц* (page table entry, PTE) определяет описываемую им страницу. Например, PTE 3 текстовой области страницы в таблице описывает виртуальную страницу 3 текстовой области.

Элемент страничной таблицы, как правило, имеет разрядность 32 бита и поделен на несколько полей. Поля содержат физический номер фрейма страницы, информацию о защите, бит *корректности* (valid), бит *изменения* (modified) и (дополнительно) бит *ссылки* (referenced). Формат рассматриваемых таблиц зависит от аппаратуры, с другой стороны таблицы страниц представляют собой обычные структуры данных, расположенные в основной памяти. В любой момент времени в памяти находится несколько страничных таблиц. MMU использует только активные таблицы, то есть те, информация о расположении которых загружена в аппаратные регистры таблицы страниц. Обычно на однопроцессорной системе активными являются две таблицы, одна из которых предназначена для ядра, а вторая — для текущего выполняющегося процесса.

Диспетчер MMU расчленяет виртуальный адрес на номер виртуальной страницы и смещение от ее начала. Затем он находит элемент таблицы страниц для этой страницы, узнает из нее номер физической страницы и комбинирует его со смещением для вычисления физического адреса.

Преобразование адресов может завершиться неудачно по трем причинам:

- **вследствие ошибки границ** (bounds error). Адрес находится за пределами диапазона корректных адресов процесса. Не существует элемента в таблице страниц для указанной страницы;
- **из-за ошибки корректности** (validation error). Элемент таблицы страниц помечен как некорректный. Обычно это означает, что страница не находится в оперативной памяти. В некоторых ситуациях бит корректности является сброшенным даже в том случае, если страница корректна и загружена в память. Подробнее об этом см. в разделе 13.5.3;
- **в случае ошибки защиты** (protection error). Страница не обладает запрошенным типом доступа (например, при попытке записи на страницу, имеющую атрибут «только для чтения», или запроса прикладного приложения к страницам ядра).

В таких случаях устройство MMU устанавливает исключительное состояние и передает управление обработчику ядра[1]. Такое исключительное состояние называется *страничной ошибкой*. Обработчику, как правило, передается ошибочный виртуальный адрес и тип ошибки (корректности или защиты; несоответствие границ относится к ошибкам корректности). Обработчик может попытаться обслужить ошибку путем помещения нужной страницы в память или уведомления процесса, то есть посылки ему сигнала (обычно это SIGSEGV).

Если обработка ошибки завершена успешно, процессу (при последующем выполнении) необходимо заново вызвать функцию, ставшую причиной ошибки. Это требует от аппаратуры сохранения корректной информации, необходимой для повторного запуска ошибочной инструкции во избежание генерации страничной ошибки.

При каждой записи в страницу аппаратура устанавливает бит *изменения* в ее таблице PTE. Если операционная система обнаруживает, что бит изменения установлен, то она сохраняет страницу на вторичном носителе при уничтожении. Если аппаратура поддерживает бит *ссылки* в таблицах PTE, то установка этого бита происходит при каждом обращении к странице. Это позволяет операционной системе отслеживать использование резидентных страниц и обновлять те из них, которые являются неиспользуемыми.

---

[1] Некоторые архитектуры не поддерживают ошибки защиты. В таких системах ядро должно принудительно объявлять ошибки корректности путем отключения страницы, после чего производится определение соответствующего действия, основанного на нахождении страницы в памяти, и типе запрашиваемого доступа.

Если процесс обладает объемным виртуальным пространством, его таблица страниц становится слишком громоздкой. Например, если процесс имеет адресное пространство размером 4 Гбайт, а размер одной страницы равен 1024 байтам, то его таблица состоит из двух миллионов элементов, что составляет объем 8 Мбайт. Такие огромные таблицы нецелесообразно хранить в оперативной памяти. Более того, большая часть такого адресного пространства, скорее всего, так и не будет использована, поскольку обычно адресное пространство процесса включает в себя определенное количество участков (текста, данных, стека и т. д.), расположенных в различных точках этого пространства. Таким образом, системе необходим более компактный вариант описания адресного пространства.

Решение проблемы состоит в поддержке сегментированных таблиц страниц или делении на страницы самой таблицы. Первый вариант является наиболее подходящим в том случае, если система поддерживает сегментацию. Каждый сегмент процесса имеет собственную таблицу страниц, которая достаточно велика для хранения диапазона корректных адресов сегмента. Второй вариант подразумевает деление на страницы самой таблицы. Это значит, что для отображения таблиц страниц нижнего уровня используется дополнительная таблица высшего уровня. С такой многоуровневой иерархией таблиц необходимо размещать только те страницы таблицы нижнего уровня, которые отображают корректные адреса процесса. Наиболее распространенным вариантом является двухуровневая система таблиц, однако некоторые архитектуры, такие как SPARC Reference MMU [15], позволяют использовать три уровня страничных таблиц.

Таким образом, таблица страниц формирует связь между устройством MMU и ядром системы, которые в равной степени имеют право обращения, использования и изменения таблиц PTE. Аппаратная часть машины также обладает набором регистров MMU, указывающих на таблицы страниц. Диспетчер MMU отвечает за использование PTE для преобразования адресов, проверку битов *корректности* и защиты процесса, а также установку битов *ссылки* и *изменения*. Ядро должно производить настройку таблиц страниц, наполнять PTE корректными данными, а также устанавливать регистры MMU для указания на таблицы. Такие регистры обычно нужно сбрасывать при каждом *переключении контекста*.

13.3.1. Кэши MMU

Применение таблиц страниц недостаточно для эффективного преобразования адресов. При выполнении каждой инструкции необходимо производить несколько обращений к памяти: сначала для разрешение виртуального адреса в значение счетчика команд, затем для выборки инструкции, а также по два обращения для каждого ее операнда в памяти. Если таблица страниц сама по себе разбита на страницы или является частью страницы более высокого

уровня, количество обращений к памяти еще более учащается. Так как каждая такая операция требует, по крайней мере, одного такта процессора, такое количество обращений в состоянии увеличить время выполнения операции до неприемлемого показателя.

Обозначенная проблема имеет два способа решения. Первым способом является добавление высокоскоростного кэша, в котором будет производиться поиск перед обращением к памяти. Компьютеры могут поддерживать отдельные кэши данных и инструкций, либо единый кэш для хранения информации обоих типов. Операция получения данных из кэша происходит намного быстрее по сравнению с обращением к оперативной памяти. На многих машинах (особенно на устаревших вариантах) кэш адресовался в физической памяти. Он полностью обслуживался аппаратурой и был прозрачен для программного обеспечения. Обращение к кэшу производилось после преобразования адресов, что снижало его преимущества. Многие современные архитектуры, например Hewlett-Packard PA-RISC [12], используют виртуально адресуемые кэши, что позволяет осуществлять поиск в кэше параллельно операции преобразования адресов. Такой подход значительно увеличивает производительность, однако рождает некоторое количество проблем, связанных с корректностью кэша, которые должны преодолеваться на уровне ядра (см. раздел 15.13).

Второй подход, уменьшающий количество обращений к памяти, заключается в применении микросхем преобразования кэша, называемых *буфером ассоциативной трансляции* (translation lookaside buffer, TLB). Буфер TLB является ассоциативным кэшем недавних преобразований адресов. Элементы TLB сходны по формату с вхождениями таблицы страниц и содержат информацию о преобразовании адресов и защите. Они также могут содержать тег, идентифицирующий процесс, к которому эти адреса относятся. Кэш считается ассоциативным из-за того, что поиск в нем является основанным на содержимом (а не на индексе), совпадение номера виртуальной страницы ищется во всех вхождениях TLB.

MMU управляет большинством операций TLB. Если он не может обнаружить данные для преобразования в буфере TLB, то дальнейший поиск производится в программных картах адресов (например, в таблицах страниц), после чего полученные данные загружаются в элемент TLB. Операционной системе необходимо работать совместно с MMU и в некоторых других ситуациях. Если ядро изменяет элемент таблицы страниц, такая модификация не фиксируется автоматически в копии вхождения TLB, находящейся в кэше. Ядру следует принудительно очистить некорректное вхождение TLB, после чего устройство MMU сможет загрузить его из памяти при следующем обращении к странице. Например, ядро может защитить страницу от записи в ответ на пользовательский запрос (системный вызов `mprotect`). После этого оно должно удалить устаревший элемент буфера TLB, указывающий на эту страницу, так как в противоположном случае процесс будет в состоянии про-

извести запись в страницу (вследствие того, что предыдущие установки разрешали это).

Методику взаимодействия ядра системы и буфера TLB определяет аппаратная часть компьютера. Аппаратура может поддерживать определенные инструкции, позволяющие загружать элементы TLB или помечать их как некорректные, либо подобные функции являются «побочным продуктом» некоторых инструкций. В некоторых системах (таких как MIPS R3000) аппаратура использует только буферы TLB, а любые таблицы страниц и другие карты памяти полностью поддерживаются ядром. В таких системах операционная система участвует в удалении каждого вхождения буфера TLB и должна загружать в него корректные данные о преобразовании.

Несмотря на то, что различные варианты TLB должны поддерживать одинаковые основные возможности, способы реализации этих средств могут отличаться друг от друга. Размер виртуальной и физической страниц, поддерживаемые типы защиты и формат карт преобразования адресов регламентируются архитектурой MMU. На машинах, обладающих аппаратной поддержкой таблиц страниц, блок MMU определяет иерархию таблиц, а также регистры, указывающие на такие таблицы. Архитектура MMU также отвечает за распределение функций между самим блоком MMU и ядром системы, а также определяет роль ядра в управлении адресами и кэшами преобразований. После рассмотрения работы диспетчера MMU в общем обратимся к трем различным аппаратным архитектурам, исходя из степени их влияния на реализацию подсистемы управления памятью в UNIX.

13.3.2. Intel 80x86

Архитектура Intel 80x86 является одной из основных платформ для систем UNIX, основанных на System V. Архитектура поддерживает адресное пространство размером 4 Гбайт (с 32-разрядной адресацией), размер одной страницы составляет 4096 байтов, имеется поддержка сегментации и разбиения на страницы [10]. На рис. 13.5 показаны стадии преобразования адресов в Intel 80x86. Каждый виртуальный адрес состоит из 16-разрядного селектора сегмента и 32-разрядного смещения. На уровне сегментации виртуальный адрес преобразуется в 32-разрядный *линейный адрес*, который позже транслируется в физический адрес на страничном уровне. Разделение на страницы может быть отключено путем очистки старшего бита в управляющем регистре под названием `CR0`. В этом случае линейный адрес ячейки памяти *равняется* ее физическому адресу.

Адресное пространство процесса может содержать до 8192 сегментов. Каждый сегмент описывается дескриптором сегмента, в котором находятся данные о его базовом адресе, размере и защите. Каждый процесс обладает собственной *локальной таблицей дескрипторов* (local descriptor base, LDT), в которой

каждый сегмент описан одним элементом. В системе существует и *глобальная таблица дескрипторов* (global descriptor table), в которой хранятся элементы, используемые для кодов, данных и сегментов стека ядра, а также хранения некоторых специальных объектов, включающих таблицы LDT процессов. При преобразовании виртуального адреса устройство MMU использует селектор сегмента для идентификации корректного дескриптора сегмента либо из GDT, либо из текущей таблицы LDT (в зависимости от бита селектора). Затем MMU должен убедиться в том, что смещение меньше, чем размер сегмента, и добавить его к базовому адресу сегмента для получения линейного адреса.

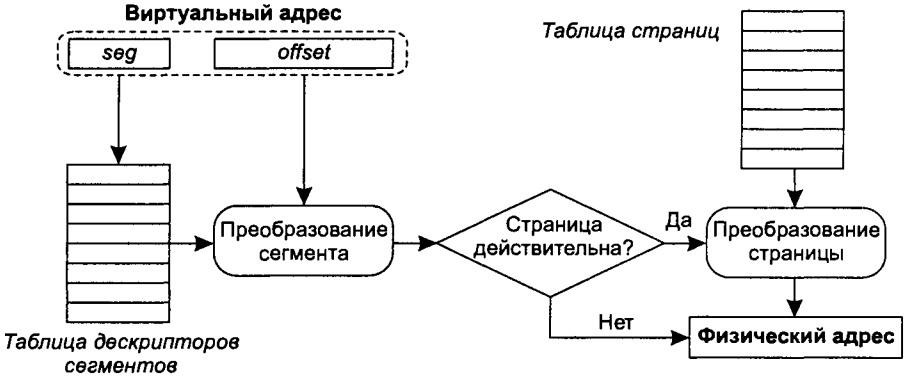


Рис. 13.5. Преобразование адресов в Intel 80x86

Реализации UNIX на платформе x86 используют сегментацию только для целей защиты памяти, элементов ядра и переключения контекстов [14]. Деление на сегменты скрыто от прикладных процессов, которые видят обычное адресное пространство. Для этого ядро устанавливает для всех сегментов прикладного процесса базовый адрес 0. Сегменты имеют большой размер, исключая верхнюю границу виртуальной памяти, резервируемой для кодов и данных ядра. Сегменты кодов доступны только для чтения, данные имеют атрибут защиты «чтение-запись», однако и те и другие ссылаются на одни и те же ячейки. Каждая таблица LDT включает ссылки на несколько специальных сегментов — *сегмент шлюза вызовов* (call gate segment) для элементов системного вызова и *сегмент состояния задачи* (task state segment, TSS) для сброса контекста регистров при переключении контекста.

Архитектура x86 использует двухуровневую схему таблиц страниц (см. рис. 13.6). Размер одной страницы, равный 4 Кбайт, позволяет процессу обладать миллионом страниц. В x86 поддерживается множество мелких таблиц для каждого процесса в отдельности (вместо огромной единой таблицы для всех). Каждая таблица имеет размер, равный одной странице, и, следовательно, может хранить до 1024 элементов PTE. Таблица содержит информацию о последовательной области размером 4 Мбайт, выровненной по 4-мегабайто-

вой границе. Таким образом, процесс может иметь до 1024 таблиц страниц. Однако большинство процессов обладает намного меньшим адресным пространством и использует лишь несколько страничных таблиц.

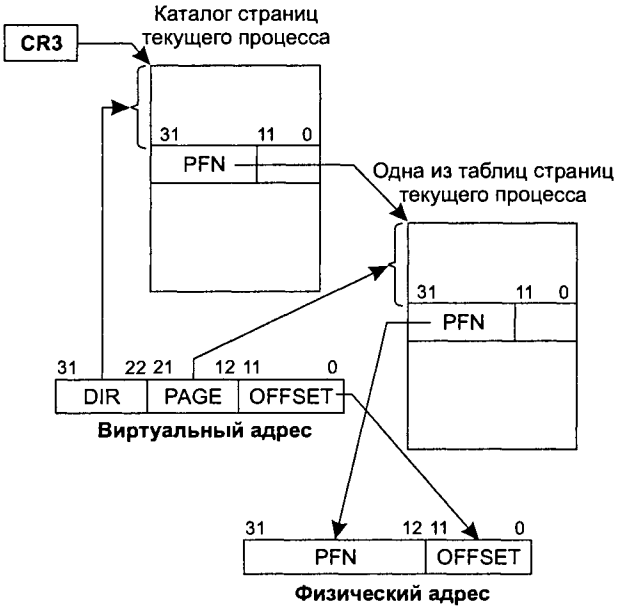


Рис. 13.6. Преобразование адресов в Intel x86

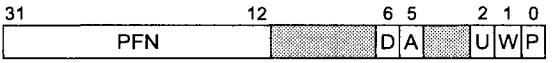
Каждый процесс также имеет *каталог страниц* (page directory), содержащий элементы PTE, указывающие на таблицы страниц. Под каталог страниц выделяется одна страница, и он может быть использован для хранения до 1024 элементов PTE, по одному для каждой таблицы. Каталог является таблицей страниц первого уровня, а таблицы страниц сами по себе составляют второй уровень. В дальнейшем мы будем придерживаться терминов «*каталог страниц*» для описания таблиц первого уровня и «*таблица страниц*» – для второго уровня.

Управляющий регистр CR3 содержит номер физической страницы текущего каталога страниц в старших 20 разрядах. Он также известен под именем *базового регистра каталога страниц* (Page Directory Base Register или PDBR). В 80x86 виртуальные адреса разбиты на три части. Старшие 10 разрядов содержат поле DIR, являющееся индексом к каталогу страниц. Это поле комбинируется со значением CR3 при получении физического адреса элемента каталога страниц для искомой таблицы страниц. Следующие 10 разрядов содержат поле PAGE, в котором хранится номер виртуальной страницы, зависящий от начала области. Он используется как индекс к таблице страниц для получения необходимого PTE, в свою очередь содержащего номер физической страницы. Младшие 12 разрядов виртуального адреса определяют смещение

на странице, которое комбинируется с номером физической страницы для получения физического адреса.

Каждый элемент таблицы страниц содержит физический номер страницы, поле защиты, а также биты *корректности* (valid), *ссылок* (referenced) и *изменения* (modified) (рис. 13.7). Поле защиты имеет два бита, один из которых указывает на доступность только для чтения (при значении 0) или возможности чтения-записи (при значении 1), второй бит указывает на страницу пользователя (сброшен) или супервизора (установлен). Если процесс выполняется в режиме ядра, все страницы доступны в режиме чтения-записи (то есть защита от записи игнорируется). В прикладном режиме страницы супервизора недоступны независимо от установок защиты, которые имеют значение только для страниц прикладных процессов. Так как полями защиты обладают не только таблицы страниц, но и каталог страниц, запрет доступа должен быть произведен в обоих вхождениях.

Поддержка бита ссылок упрощает ядру задачу отслеживания использования страниц. Регистр CR3 необходимо сбрасывать при каждом переключении контекста, так как он будет указывать на каталог страниц уже нового процесса.



PFN	Номер страничного фрейма
D	«Грязная»
A	Доступна (по ссылке)
U	Пользователь (0) / супервизор (1)
W	Чтение (0) / запись (1)
P	Присутствует (действительна)

Рис. 13.7. Элемент таблицы страниц в архитектуре Intel 80x86

Буфер TLB в архитектуре x86 не может быть доступен ядру напрямую. Однако он сбрасывается целиком автоматически при записи регистра PDBR, либо принудительно инструкцией move или косвенно в результате переключения контекста. Ядро UNIX сбрасывает TLB при изменении корректности элемента таблицы страниц (например, при повторном использовании страницы).

Архитектура x86 поддерживает четыре привилегированных уровня *защищенности*, из которых UNIX использует только два. Ядро выполняется на привилегированном уровне. Он позволяет выполнять привилегированные инструкции (например, изменяющие регистры MMU) и обращаться к любым сегментам и страницам (прикладных приложений и управляющих программ). Пользовательские коды выполняются на наименее привилегированном уровне, который ограничивает их только обычными инструкциями и обращениями только к страницам прикладных процессов в собственном сегменте. *Шлюз вызовов* (call gate) позволяет пользователю производить вызовы процедур с отличным (более высоким) уровнем привилегий.

13.3.3. IBM RS/6000

IBM RS/6000 [4] является машиной на основе архитектуры *RISC* (reduced instruction set computer, *компьютер с сокращенным набором команд*), работающей под управлением операционной системы IBM AIX, продолжившей линию System V. Архитектура памяти этой машины имеет два интересных свойства — использование единого системного адресного пространства и применение *инвертированной таблицы страниц* (inverted page table) для преобразования адресов. Еще одной системой, обладающей такими же качествами, является Hewlett-Packard PA-RISC [12].

Проблема обычных таблиц страниц, индексируемых по виртуальным адресам, заключается в том, что их размер пропорционален виртуальному адресному пространству, которое может быть огромным у некоторых процессов. Многие современные архитектуры поддерживают 64-разрядную адресацию, что еще более увеличивает возможный объем адресного пространства. Следовательно, таблицы страниц могут превысить размер имеющейся физической памяти. Одним из решений, принятым в таких системах, как x86, является поддержка иерархических таблиц страниц. Однако даже в этом случае для больших процессов ядру необходимо уметь разбивать на страницы сами таблицы страниц.

Использование инвертированных таблиц страниц является способом ограничения физической памяти, необходимой для поддержания информации о преобразовании адресов. Такие таблицы имеют по одному элементу на каждую страницу в физической памяти и отображают физические номера страниц в их виртуальные адреса. Так как объем физической памяти намного меньше по сравнению с общим объемом виртуальной памяти всех процессов системы, инвертированная таблица является весьма компактной. Однако диспетчеру MMU необходимо транслировать виртуальные адреса, что нельзя сделать напрямую из инвертированной таблицы. Следовательно, система должна поддерживать другие структуры данных для преобразования виртуального адреса в физический. Об этом мы и поговорим сейчас.

В архитектуре RS/6000 реализованы два типа виртуальных адресов. Она обладает единым системным виртуальным адресным пространством с адресами размером 52 разряда. Общий размер такого пространства равен 2^{52}, или примерно 4×10^{15} байтов. Каждый процесс использует 32-разрядные адреса, карты адресного пространства каждого процесса отображают часть системного адресного пространства, как это показано на рис. 13.8. Виртуальные и физические страницы имеют размер 4096 байтов, что совпадает с размером дискового блока, принятым по умолчанию. 32-разрядный виртуальный адрес процесса делится на три части — 4-разрядный идентификатор сегмента, 16-разрядный индекс страницы и 12-разрядное смещение на странице. Таким образом, адресное пространство составляет 16 сегментов размером 256 Мбайт.

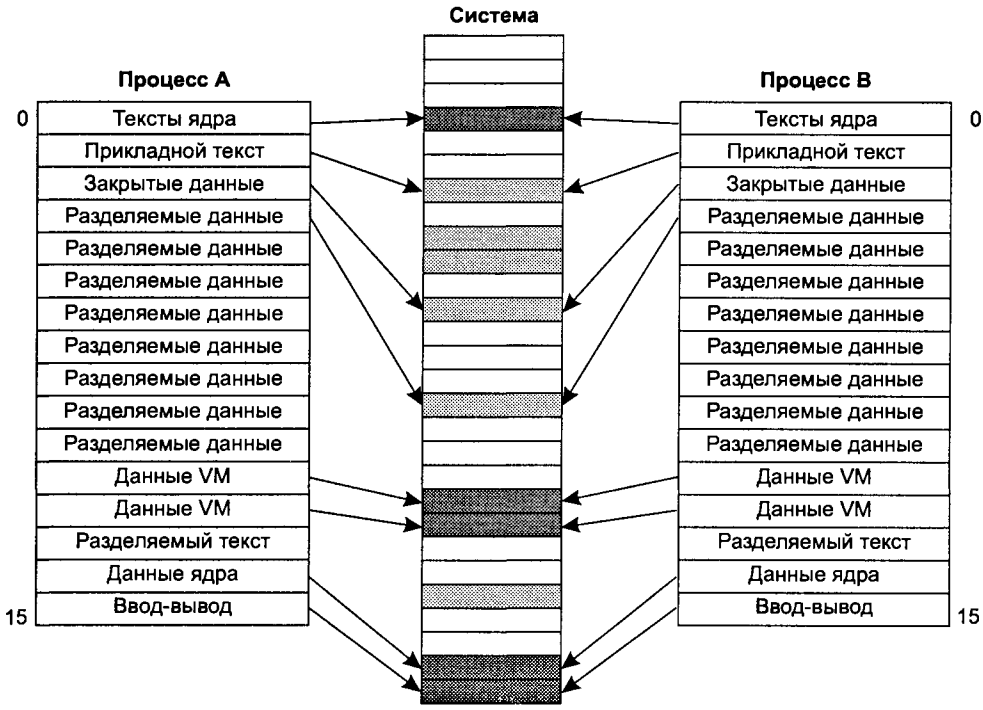


Рис. 13.8. Адресное пространство в RS/6000

Компьютер RS/6000 имеет 16 регистров сегмента, загружаемых вместе с дескрипторами сегмента каждого процесса. Каждому сегменту отводится определенная роль. Сегмент 1 хранит тексты программы. Сегмент 2 используется для закрытых данных процесса и хранит пользовательские данные, кучу, стек, а также стек ядра и область u процесса. Сегменты 3–10 являются совместно используемыми и применяются для разделяемой памяти и отображения на файлы (см. раздел 14.2). Сегмент 13 содержит совместно используемые тексты, например, загруженные из разделяемой памяти. Остальные сегменты используются только ядром. Нулевой сегмент хранит тексты ядра, сегменты 11 и 12 содержат структуры управления памятью, а в сегменте 14 находятся структуры данных ядра. Сегмент 15 зарезервирован для адресов ввода-вывода.

На рис. 13.9 показана схема преобразования в системный виртуальный адрес. Идентификатор сегмента определяет регистр сегмента, имеющий размер 32 байта. Он содержит 24-разрядный индекс сегмента, формирующий старшие 24 разряда в системном виртуальном адресе. Они комбинируются с 16-разрядным индексом виртуальной страницы из виртуального адреса процесса для формирования номера виртуальной страницы в системном адресном пространстве. Позже они преобразуются для получения номера физической страницы.

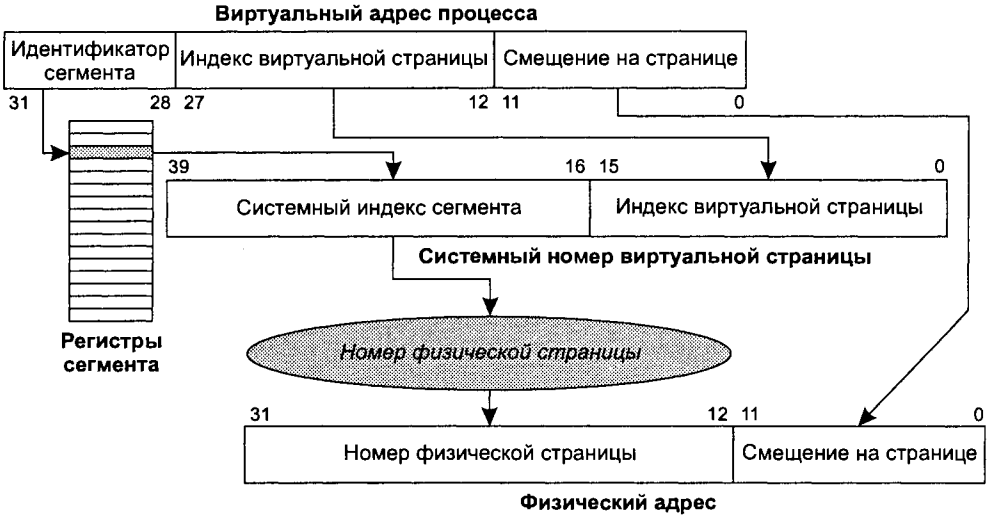


Рис. 13.9. Преобразование адресов в архитектуре RS/6000, стадия 1

Как уже рассказывалось ранее, в RS/6000 нет прямых карт преобразования виртуальных адресов в физические. Вместо этого архитектура поддерживает инвертируемую таблицу страниц под названием *таблицы страничных фреймов страниц* (page frame table, PFT), в которых один элемент описывает один физический диапазон. Система задействует технику хэширования для преобразования виртуальных адресов, как это показано на рис. 13.10. Структура данных под названием *hash anchor table, HAT*) содержит информацию, используемую для преобразования номера системной виртуальной страницы в хэш-значение, указывающее на связанный список элементов PFT. Каждый элемент PFT включает в себя следующие поля:

- номер виртуальной страницы, которую он отображает;
- указатель на следующий элемент в цепочке;
- флаги (корректности, ссылок и изменения);
- информацию о защите и блокировке.

В RS/6000 таблица HAT применяется для нахождения необходимого номера виртуальной страницы по хэш-значению. Индекс элемента PFT эквивалентен номеру физической страницы, размер которого равен 20 битам. Он комбинируется с 12-разрядным смещением страницы виртуального адреса процесса для получения физического адреса.

Процедура преобразования является весьма медленной, поэтому нежелательно проводить эти действия при каждом обращении к памяти. Архитектура RS/6000 поддерживает две отдельных таблицы TLB — 32-элементную таблицу инструкций и 128-элементную таблицу данных. В обычном режиме работы системы эти буферы отвечают за большинство операций преобразования адресов, операция поиска вызывается только в том случае, если необходи-

мые адреса отсутствуют в TLB. Кроме этого, RS/6000 обладает отдельными кэшами для хранения данных и инструкций. Кэш данных имеет размер 32 или 64 Кбайт, кэш команд — 8 или 32 Кбайт, в зависимости от модели системы [6]. Оба кэша адресуются виртуально, следовательно, если в них имеются необходимые данные, преобразование адресов производить не нужно. Виртуально адресуемые кэши требуют участия ядра системы в решении некоторых проблем целостности информации. Более подробно об этом читайте в разделе 15.13.

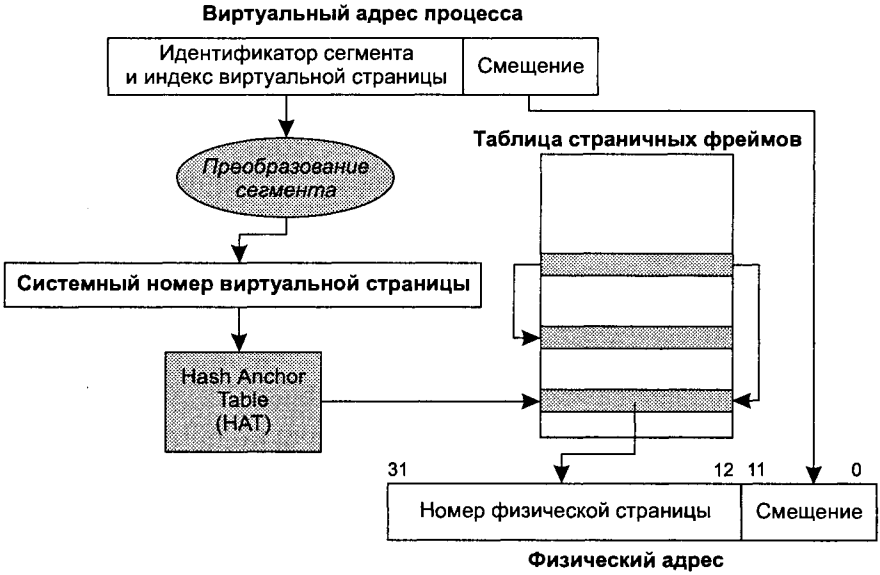


Рис. 13.10. Преобразование адресов в архитектуре RS/6000, стадия 2

13.3.4. MIPS R3000

MIPS R3000 является системой на базе архитектуры RISC и работает под управлением SVR4 UNIX или ULTRIX компании Digital Equipment Corporation (системы, основанной на 4.2BSD). Такая машина обладает нестандартной архитектурой MMU [11], что заключается в отсутствии аппаратной поддержки таблиц страниц. Единственным вариантом преобразования адресов на уровне аппаратуры являются операции, определенные в микросхеме буфера TLB.

Построение задач управления памятью и интерфейс между аппаратурой и ядром сильно зависят от архитектурных особенностей машины. Например, в Intel 80x86 структура элементов TLB непрозрачна для ядра. Единственной допустимой операцией над ней является объявление некорректным отдельного элемента (по его виртуальному адресу) или всей таблицы TLB сразу. Однако в архитектуре MIPS формат и содержимое элементов таблицы TLB является для ядра открытым. Архитектура MIPS позволяет производить чтение, изменение и загрузку определенных элементов таблицы.

Виртуальное адресное пространство поделено на четыре сегмента (см. рис. 13.11). Первые 2 Гбайт занимает сегмент kuseg, используемый в качестве прикладного адресного пространства. Остальные сегменты доступны только в режиме ядра. Сегменты kseg0 и kseg1 отображают напрямую первые 512 Мбайт физической памяти, следовательно, они не требуют буферов TLB. Сегмент kseg0 использует кэш данных/команд, в отличие от kseg1, который их не использует. Верхний гигабайт памяти передан отображаемому и кэшируемому сегменту kseg2. Адреса в kseg2 могут отображать любую ячейку физической памяти.

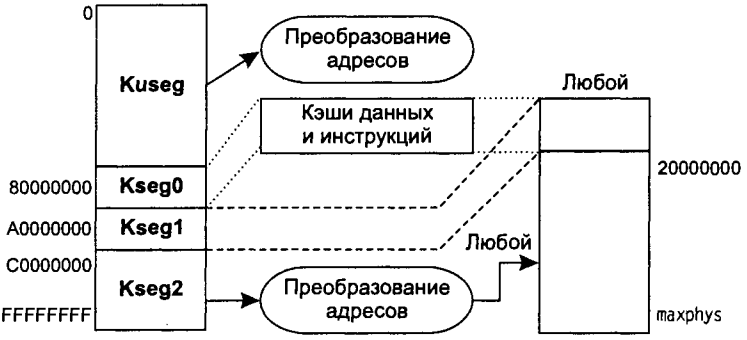


Рис. 13.11. Виртуальное адресное пространство в MIPS R3000

На рис. 13.12 показаны регистры устройства MMU и формат элемента TLB. Размер страницы в архитектуре MIPS постоянен и равняется 4 килобайтам, следовательно, виртуальный адрес разделяется на 20-разрядный номер страницы и 12-разрядное смещение. Буфер TLB содержит 64 элемента, каждый из которых имеет размер 64 байта. Регистры entryhi и entrylo имеют точно такой же формат, как и старшие и младшие 32 бита элемента TLB соответственно. Они используются для чтения и записи в элемент TLB. Поля номера виртуальной страницы VPN (virtual page number) и номера физического фрейма PFN (physical frame number) позволяют производить преобразование виртуальных номеров страниц в физические. Поле PID служит в роли тега, ассоциируя каждый элемент TLB с процессом. Оно имеет размер 6 битов и может принимать значения от 0 до 63, что, однако, не является традиционными идентификаторами процессов. Каждому процессу, обладающему активными элементами TLB, присваивается идентификатор tlbpid в диапазоне от 0 до 63. Ядро устанавливает поле PID в регистре entryhi в значение tlbpid текущего процесса. Аппаратура сравнивает его с соответствующим полем в элементах TLB и отменяет преобразование в случае их несовпадения. Это позволяет буферу TLB содержать элементы с одним и тем же номером виртуальной страницы, относящиеся к различным процессам, без потенциальных конфликтов.

При установке бита N (no cache) страница не помещается в кэш данных или команд. Бит G (global) указывает на необходимость игнорирования PID для такой страницы. Если бит V (valid) сброшен, страница считается некор-

ректной, если сброшен B (dirty) — элемент защищен от записи. Заметим, что система не поддерживает битов ссылки и изменения.

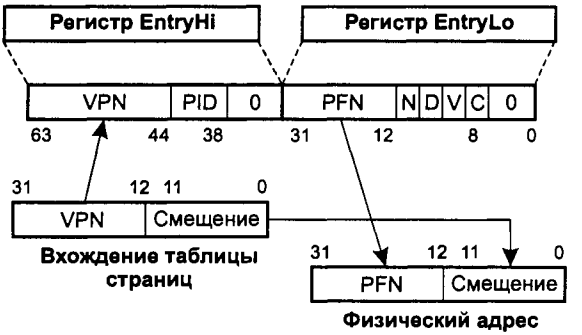


Рис. 13.12. Преобразование адресов в MIPS R3000

При преобразовании адресов из сегментов kuseg или kseg2 номер виртуальной страницы сравнивается со всеми элементами TLB. Если совпадение будет обнаружено (со сброшенным битом G), то система далее сравнит PID элемента с текущим значением tlbpid, хранящимся в регистре entryhi. При совпадении значений (или в случае установленного бита G), а также установке бита V, поле PID элемента будет указывать на корректный номер физической страницы. В противоположном случае возникнет ошибка TLBmiss. Бит B необходимо устанавливать для операции записи. Если это не так, в системе возникнет ошибка TLBmod.

На аппаратном уровне не поддерживаются каких-либо средства (например, таблицы страниц памяти), при помощи которых можно выполнить обработку ошибок, поэтому это необходимо делать на уровне ядра системы. Ядро проверяет собственные отображения памяти и либо обнаруживает в них правильное преобразование, либо посылает процессу сигнал. В первом случае необходимо загрузить корректный элемент TLB, после чего заново выполнить команду, повлекшую сбой. Аппаратная часть не требует от ядра операционной системы поддержки карт памяти в виде таблиц и не ограничивает формат их элементов. Однако на практике реализации UNIX на архитектуре MIPS используют таблицы страниц для организации базовой структуры управления памятью. Формат регистра entrylo имеет обычную форму PTE. Младшие 8 битов этого регистра не задействуются на аппаратном уровне, поэтому ядро может использовать их для любых целей.

Отсутствие поддержки битов ссылок и изменения повлияло на разработку ядра системы. Ядру необходимо знать, какие из страниц памяти были изменены, так как последние должны быть сохранены перед повторным использованием. Разработчики решили эту проблему путем установки атрибута защиты от записи всех чистых страниц (сброса бита D). При первой попытке записи в такие страницы возникает исключительная ситуация TLBmod. Обработчик производит установку бита B в буфере TLB и устанавливает соответствую-

щие биты в программном элементе PTE, помечая страницу как «грязную». Информация о ссылках на страницу также получается косвенными методами (см. раздел 13.5.3).

Особенность описываемой архитектуры состоит в большом количестве исключительных состояний, так как каждое отсутствие данных в TLB обрабатывается программно. Необходимость отслеживания изменений страниц и ссылок на них приводит к увеличению ошибочных состояний. С другой стороны, простота архитектуры управления памятью дает высокую скорость преобразования адресов в том случае, если необходимая информация присутствует в кэше. Более того, применение высокоскоростных процессоров уменьшает потери времени на обработку исключительных ситуаций. И, наконец, архитектура поддерживает неотображаемую область памяти kseg0, используемую для статических данных и текстов ядра, что позволяет увеличить скорость работы системы в режиме ядра, поскольку в этом случае не нужно производить преобразование адресов. Поддержка этой области памяти также уменьшает размер буфера TLB, который необходим только для адресов памяти прикладных процессов и некоторых динамически выделяемых структур данных ядра.

13.4. Пример реализации: система 4.3BSD

Мы разобрались с основными концепциями методики загрузки страниц по запросу, а также оценили влияние характеристик аппаратуры на их практическую реализацию. С целью закрепления материала рассмотрим реализацию управления памяти в ОС 4.3BSD как практический пример. Первой системой UNIX, имевшей поддержку виртуальной памяти, была 3BSD. В последующих версиях системы архитектура памяти постоянно улучшалась. ОС 4.3BSD стала последней версией системы, основанной на принятой модели памяти. В 4.4BSD была реализована поддержка новой архитектуры памяти на основе ОС Mach (о ней мы расскажем в разделе 15.8). Подробно об архитектуре памяти в 4.3BSD можно прочесть в книге [13][1]. В этом разделе мы лишь кратко задержимся на ее основных возможностях, рассмотрим преимущества и недостатки архитектуры, а также попытаемся сформулировать некоторые соображения, положенные в основу появления новых, более сложных технологий управления памятью, описываемых в следующих главах.

Изначальной аппаратной платформой для реализации системы BSD была машина VAX-11, однако впоследствии ОС была перенесена и на другие архитектуры. Характеристики аппаратуры сильно влияют на алгоритмы ядра, в частности на реализацию функций низкого уровня, работающих с таблицами страниц памяти и буфером преобразования. Задача переноса подсистемы

[1] В 1996 году те же авторы-разработчики выпустили «продолжение» — The Design and Implementation of the 4.4BSD Operating System, Addison-Wesley Longman, Inc. — *Прим. ред.*

управления памятью BSD на другие платформы была весьма сложной, так как практически все компоненты системы зависели от особенностей аппаратной части машины. В результате некоторые реализации BSD программно эмулируют архитектуру памяти машины VAX, в том числе ее адресное пространство и формат элементов таблицы памяти. Мы не будем подробно углубляться в архитектуру памяти VAX, поскольку на сегодняшний день эта машина является морально устаревшей. Однако при описании системы BSD мы остановимся на ее некоторых важнейших средствах.

Система 4.3BSD использует лишь малое количество базовых структур данных: карта ядра (core map) описывает физическую память, таблицы страниц служат для описания виртуальной памяти и карты дисков применяются для областей свопинга. В системе также поддерживаются карты ресурсов, которые применяются для управления выделением таких ресурсов, как таблицы страниц и пространство для свопинга. Некоторая важная информация хранится в структуре proc и области u каждого процесса.

13.4.1. Физическая память

Физическая память может рассматриваться как последовательный массив байтов в диапазоне от 0 до n, где n равняется общему объему памяти системы. Логически память делится на страницы, их размер зависит от аппаратной архитектуры конкретной машины. Память может быть разбита на три раздела, как это показано на рис. 13.13. Первым разделом является *область, не разбиваемая на страницы* (nonpaged pool), она содержит коды ядра и часть его данных, размещаемых статически (или во время загрузки системы). Так как исключительное состояние страницы ядра может привести к блокировке процесса ядра в критической точке выполнения, большинство реализаций UNIX требуют, чтобы страницы ядра не обрабатывались на уровне страниц. Высшая часть физической памяти зарезервирована для хранения сообщений об ошибках, создаваемых при крахе системы. Между этими двумя разделами расположена *страничная область* (paged pool), занимающая почти весь объем физической памяти машины. В ней находятся все страницы, относящиеся к прикладным процессам, а также динамически выделяемые страницы ядра. Последние помечаются системой как не подлежащие страничной обработке, хотя они и размещены в страничной области.

Рис. 13.13. Физическая память

Такие физические страницы получили название страничных фреймов. Фрейм хранит содержимое страницы процесса. Страница, вложенная во фрейм, может быть в любое время перенесена в другую страницу, следовательно, системе необходимо отслеживать информацию о содержимом каждого фрейма, что реализовано при помощи карты ядра. *Карта ядра* (core map) представляет собой массив элементов структур cmap, по одному на каждый фрейм в страничной области[1]. Карта ядра сама по себе является структурой данных ядра, размещается при загрузке системы и хранится в не-страничной области памяти резидентно. Элемент карты ядра содержит следующую информацию о фрейме:

- **имя** (name). Имя или идентификатор страницы, находящейся во фрейме. Пространство имен страниц процесса описывается идентификатором владельца процесса, типом информации (данные или стек), а также номером виртуальной страницы для страниц области. Текстовые страницы могут быть совместно использованы несколькими процессами, следовательно, их владельцами являются текстовые структуры программ. Элемент карты ядра хранит индекс к таблице процесса или таблице текстов. Имя вида <тип, владелец, номер виртуальной страницы> (<type, owner, virtual page number>) позволяет ядру производить обратное преобразование адресов, то есть обнаружение элементов PTE, относящихся к странице фрейма (см. рис. 13.14);
- **список свободных страниц** (free list). Указатели на предыдущий и следующий элементы, соединяющие свободные страницы с их списком. Список ведется в порядке «наименее частого использования» (см. раздел 13.5.3) и используется процедурами отображения памяти для выделения и освобождения физической памяти;
- **кэш текстовых страниц** (text page cache). Имена страниц имеют значение только на время существования процесса, являющегося их владельцем. Такой вариант является приемлемым для страниц стека и данных, поскольку подобные страницы становятся совершенно ненужными после завершения работы процесса. Однако в случае текстовых страниц существует ненулевая вероятность использования их другим процессом через какой-то промежуток времени. Если процессу необходимо повторно запустить некий код, который по-прежнему находится в памяти, становится очевидной нужность повторного использования страниц вместо чтения их с диска. Для идентификации таких страниц уже после завер-

[1] На самом деле элемент структуры cmap охватывает сразу кластер фреймов. Кластеры являются логической страницей, составленной из (определенного) числа физических страниц. Такой подход позволяет увеличить производительность путем грануляции (степень детализации – характеристика программы, определяемая размером неделимых параллельно выполняемых фрагментов. — *Прим. ред.*) операций и уменьшения размера таких структур данных, как карта ядра.

шения работы их владельца (или владельцев) в элементе карты ядра хранятся данные о размещении их на диске (номер устройства и блока). Текстовые страницы также хешируются (на основе номеров устройства и блока) с целью их быстрой последующей локализации. Страница может быть загружена как из выполняемого файла, так и из области свопинга;

- **синхронизация** (synchronization). Набор флагов для синхронизации доступа к странице. Обычно страница блокируется при перемещении данных с диска или на диск.

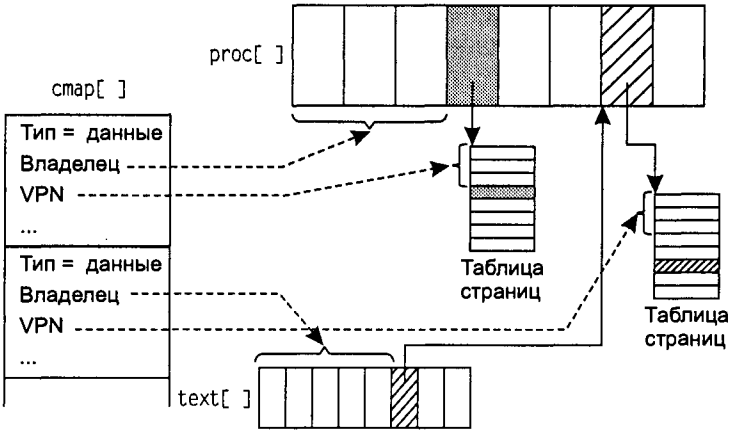


Рис. 13.14. Преобразование физического адреса в виртуальный

13.4.2. Адресное пространство

Подсистема виртуальной памяти BSD использует модель адресного пространства 32-разрядной машины VAX-11, поддерживающей страницы размером 512 байтов. Адресное пространство VAX объемом 4 Гбайт поделено на четыре области равных размеров. Первый гигабайт называется *программной* (program) областью и обозначается P0. Здесь содержатся тексты и данные процессов. За ним следует область *управления* (control) P1, в которой хранится стек прикладных процессов, области u и стек ядра. Следующая область (S0) называется *системной* (system) и содержит данные и тексты ядра. Последний участок памяти является зарезервированным для будущего применения и не поддерживается аппаратной частью VAX. Схема организации памяти позволяет увеличивать каждую область без потерь адресного пространства.

На аппаратном уровне VAX управляет использованием таблиц страниц для разрешения виртуальных адресов. Таблицы адресов служат одновременно нескольким целям. Ядро задействует их как для описания адресного пространства процесса (в структуре proc хранится лишь краткая информация, включающая в себя расположение и размер таблиц), так и для хранения дан-

ных о действиях по инициализации страниц (см. раздел 13.4.3). Ядро изменяет часть битов PTE, не трогаемых аппаратной частью.

Отображение данных и текстов ядра реализовано при помощи единственной *системной таблицы страниц*. Однако каждый процесс обладает двумя таблицами, одна из которых отображает область P0 (текст и данные), а вторая — область P1 (стек, область u и т. д.). Системная таблица целиком находится в физической памяти (располагаясь непрерывно). Каждая таблица прикладных процессов размещается в системной виртуальной памяти и отображается набором элементов PTE в разделе Userprmap системной таблицы страниц. Для описания свободных участков Userprmap ядро использует карту ресурсов, составленную из пар <base, size>. Выделение элементов из этой карты происходит на основе методики «первого наиболее подходящего». При увеличении нагрузки карта становится сильно фрагментированной. Это приводит к тому, что процесс может не обнаружить элементы PTE величины, достаточной для отображения таблиц страниц. В этом случае ядро запускает задачу swapper для выгрузки из памяти этого процесса с целью освобождения некоторого количества места в разделе Userprmap.

Можно ли использовать таблицы страниц совместно? В частности, если два процесса выполняют одну и ту же программу, вправе ли они использовать единственную таблицу, отображающую область текстов? Теоретически это осуществимо, и многие системы UNIX позволяют такое разделение. Однако в ОС BSD имеются определенные проблемы, препятствующие реализации такой возможности. Каждый процесс должен иметь таблицу для отображения области P1. Такая таблица целиком находится в системном виртуальном адресном пространстве и, следовательно, должна описываться непрерывным набором *системных элементов PTE* в Userprmap. По той причине, что область данных не разделяема между процессами, только часть таблицы страниц P0 может быть использована совместно. Так как каждый процесс обладает собственным набором вхождений Userprmap, элементы PTE, отображающие текстовую область, должны указывать на один и тот же набор страниц. Это, в свою очередь, означает, что начало области данных таблицы страниц должно быть связано с новой страницей и описываться новым элементом PTE в Userprmap. Отсюда вытекает потребность ограничения областей данных размером 64 Кбайт.

Такое условие приводит к несовместимости с точки зрения пользователя. С целью предупреждения подобной ситуации в системе BSD есть соглашение, чтобы каждый процесс обладал собственной таблицей текста. Если несколько процессов будут разделять текстовую область памяти, это потребует синхронизации элементов таблицы страниц. Например, если один из процессов поместит страницу в память, то изменение элемента PTE для этой страницы должно быть сделано всеми процессами, разделяющими участок памяти. На рис. 13.15 показана схема обнаружения ядром всех таблиц страниц, отображающих определенный участок текстов.

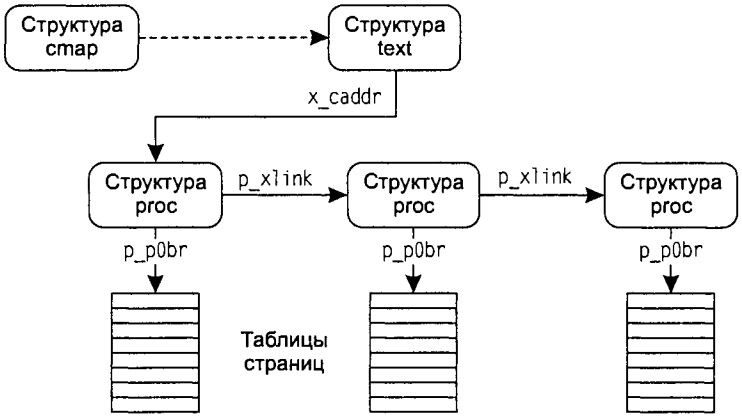


Рис. 13.15. Несколько отображений одной и той же страницы текстов

13.4.3. Где может располагаться страница памяти

В любой момент времени конкретная страница памяти процесса может находиться в одном из следующих состояний:

- **резидентная** (resident). Страница загружена в основную память. Элемент таблицы страниц содержит физический номер страницы фрейма;
- **заполненная по необходимости** (fill-on-demand). Процесс пока не произвел ссылку на такую страницу. Она будет помещена в память при первом обращении к ней. Существуют два типа страниц, заполняемых при необходимости:
  - **заполненная текстом** (fill-from-text). При первом обращении в страницу считываются тексты и статические данные из исполняемого файла;
  - **заполненная нулями** (zero-fill). Страница используется для хранения неинициализируемых данных, кучи и стека и заполняется нулями при необходимости;
- **выгруженная** (outswapped). К этому типу относятся страницы, ранее считанные в память и выгруженные в область свопинга с целью освобождения памяти для размещения других страниц. Такие страницы могут быть восстановлены из области свопинга.

Ядру нужно поддерживать данные обо всех нерезидентных страницах для того, чтобы иметь возможность возвратить их в физическую память по требованию. Для выгруженных страниц ядро сохраняет информацию об их размещении на устройствах свопинга. В случае страниц, заполняемых нулями, ядру достаточно уметь определять их тип. Для страниц, заполняемых текстом, ядру необходимо знать их расположение в файловой системе. Такие

данные можно получить путем вызова системных процедур, считывающих массив дисковых блоков индексного дескриптора. Однако такой метод является не совсем эффективным, так как для обнаружения нужной страницы часто требуется обращение к другим («косвенным») блокам файла.

Решение, реализованное в 4.3BSD, основано на том, что все биты таблицы страниц, кроме битов защиты и корректности, не проверяются аппаратурой до тех пор, пока бит корректности не будет установлен. Так как у всех нерезидентных страниц бит корректности сброшен, поля могут быть заполнены данными, позволяющими отслеживать такие страницы. На рис. 13.16, *а* показан формат элемента таблицы страниц, обуславливаемый архитектурой машины VAX-11. Для обнаружения всех элементов, заполняемых по необходимости, система 4.3BSD использует 25-й бит, не используемый аппаратной составляющей.

Для обычных вхождений таблицы страниц бит заполнения по необходимости сброшен (см. рис. 13.16, *б*). Если он оказывается установленным, это указывает на то, что страница является заполняемой по необходимости (при этом бит корректности должен быть сброшен), а элемент таблицы страниц — специальным с отличающимся набором полей (см. рис. 13.16, *в*). Вместо *номера страничного фрейма* и бита *изменения* такое вхождение хранит *номер блока файловой системы* и бит, указывающий на тип страницы (при установке — заполняемой текстом, при сбросе — заполняемой нулями). Для страниц, заполняемых текстом, номер устройства запрашивается из текстовой структуры программы.

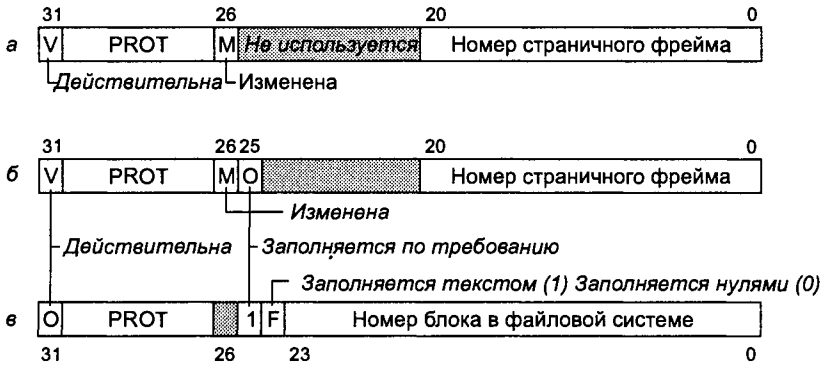


Рис. 13.16. Формат элемента таблицы страниц в системе 4.3BSD: *а* — формат вхождения VAX-11; *б* — обычное вхождение; *в* — вхождение для страницы, заполняемой по необходимости

Интерпретация выгруженных страниц неодинакова. Элементы PTE таких страниц имеют биты *корректности* и *заполнения по необходимости* сброшенными, значение *номера страничного фрейма* равняется нулю. Для управления выгруженными страницами ядро поддерживает отдельные карты свопинга (см. более подробное описание в следующем разделе).

13.4.4. Пространство свопинга

Пространство свопинга необходимо по двум причинам. Во-первых, в случае выгрузки процесса из памяти целиком все его страницы должны быть сохранены на диск. Во-вторых, иногда требуется выгружать из памяти отдельные страницы процессов, для которых нужно некоторое место для их временного хранения. Для свопинга резервируется один или более логических дисков или разделов. Такие разделы являются неформатированными, то есть не содержат файловой системы. Система 4BSD позволяет располагать область свопинга на нескольких дисках, что увеличивает производительность страничной подсистемы. Такие разделы представляются в виде единой области свопинга. Это обеспечивает примерно одинаковую нагрузку на все разделы, составляющие область свопинга. Место страницы в пространстве свопинга указывается в виде номера псевдоустройства, представляющего логическую область свопинга, и смещения внутри этого устройства. Эти данные преобразуются в соответствующую им физическую область и смещение на низком уровне.

Строго говоря, пространство свопинга необходимо для размещения тех страниц, которые не обязательно хранить в памяти. Однако, следование такому правилу может привести к выгрузке иного процесса из памяти, то есть большинство страниц этого процесса окажется в пространстве свопинга на большой промежуток времени. Если это произойдет при выполнении программы, то можно ожидать ее «зависания» или неожиданного завершения. Для предупреждения таких ситуаций в системе 4.3BSD проводится менее агрессивная политика свопинга. При старте процесса ядро выделяет все необходимое для его данных и стека пространство. Область свопинга обычно представляет собой области большой величины, что позволяет разместить в ней некоторое количество данных. Однако если объем сбрасываемого участка вырастает за пределы области, то для его размещения необходимо сначала выделить дополнительное пространство свопинга. Такой подход гарантирует, что пространство свопинга может быть исчерпано только в определенных ситуациях, то есть при создании или расширении области памяти.

Нет нужды выгружать из памяти текстовые страницы (или неизменяемые страницы с данными), так как они могут быть получены из исполняемых файлов. Это — одна из проблем реализации системы BSD, причиной которой является хранение данных о размещении блока файла в *элементе PTE, заполняемом по требованию*. После помещения страницы в память эта информация заменяется номером страничного фрейма. В результате запрос страницы из файла требует пересчета ее месторасположения и, возможно, обращения к одному или нескольким блокам косвенной связи. Эти операции являются слишком затратными, поэтому в BSD текстовые страницы, как правило, сохраняются в пространстве свопинга. Если несколько процессов выполняют одну и ту же программу, в области свопинга достаточно размес-

тить лишь одну копию страницы. Так как размер текстовой области обычно является фиксированным, для нее разумно выделить один непрерывный участок пространства. Это позволит прочесть за одну дисковую операцию сразу несколько смежных страниц.

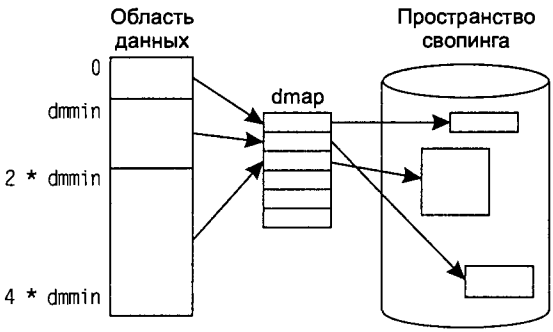


Рис. 13.17. Запись информации о пространстве свопинга в структуре dmap

Ядро хранит данные о выделении пространства свопинга в структурах dmap (по одной на каждый участок). Первая выделяемая порция имеет размер, равный dmmin (обычно это 16 Кбайт). Каждая последующая область в два раза больше предыдущей. Максимально допустимый размер области описывается переменной dmmax и составляет от 0,5 до 2 Мбайт. Все остальные области имеют длину, равную dmmax. Структура dmap представляет собой массив фиксированного размера (рис. 13.17), каждый элемент которого содержит начальный адрес области на устройстве свопинга. Индекс элемента (вместе со значениями dmmin и dmmax) преобразуется в длину области. Постоянный размер массива четко ограничивает максимально возможную протяженность областей данных или стека. Карты областей данных и стека хранятся в области u. Текстовые области отображаются немного по-другому, так как их размеры являются фиксированными. Независимо от количества процессов, использующих их, на диске хранится только одна копия текстовой области, следовательно, карта свопинга является частью текстовой структуры. Пространство свопинга выделяется для текстовых страниц одинаковыми порциями, соответственно значению dmtext (обычно 512 Кбайт). Последняя область может быть заполнена лишь частично.

13.5. Операции работы с памятью 4.3BSD

Рассмотрим алгоритмы нескольких наиболее важных операций управления памятью в системе 4.3BSD — создание процесса, обработку страничной ошибки, замену страницы памяти и свопинг.

13.5.1. Создание процесса

Системный вызов fork создает новый процесс (потомок), адресное пространство которого является дубликатом пространства предка. При этом имеют место несколько операций, связанных с управлением объектами в памяти.

- **Пространство свопинга.** Первой стадией создания процесса является выделение пространства свопинга для областей данных и стека потомка. Объем выделяемого пространства эквивалентен тому объему, которым в данный момент располагает родительский процесс. Если операция выделения пространства свопинга завершается неудачно, вызов fork возвращает ошибку.
- **Таблицы страниц.** Ядру необходимо выделить элементы PTE в области Userptmap для отображения таблиц страниц создаваемого процесса. Если операция терпит неудачу, система выгрузит другой процесс в область свопинга для освобождения места в Userptmap. Системные элементы PTE инициализируются путем выделения физических страниц из списка свободных страниц памяти.
- **Область u.** Новая область u для вновь порожденного процесса выделяется и инициализируется путем копирования области u его родителя. Для разрешения прямого доступа к полям области u потомка ядро отображает их в специальной карте под названием Forkmap, располагаемой в системном пространстве.
- **Область текстов.** Процесс-потомок добавляется в список процессов, совместно использующих текстовые области, принадлежащие его предку. Элементы таблицы страниц текстовой области копируются от родительского процесса.
- **Данные и стек.** Данные и стек необходимо копировать по одной странице за один прием. Для тех из них, которые заполняются по требованию, достаточно скопировать только элементы PTE. Остальные страницы формируются путем выделения физической памяти и дублирования их от родительского процесса. Если страницы процесса-предка оказались выгруженными в область свопинга, их необходимо считать и затем сделать дубликат. Все вновь скопированные страницы помечаются как измененные, следовательно, они могут быть выгружены в область свопинга до повторного использования.

Операция fork является весьма трудоемкой. Основную нагрузку несет копирование страниц, совершаемое на последнем этапе работы вызова. Однако дублирование областей данных и стека целиком представляется избыточным, так как большинство процессов обычно завершают работу или вызывают exec для загрузки новой программы сразу после завершения работы fork, что означает уничтожение адресного пространства одного из процессов.

Существует две методики, позволяющие уменьшить значение описанной проблемы. Первая из них называется *копированием при записи* (copy-on-write).

Эта технология была адаптирована в System V UNIX. Предок и потомок используют одни и те же страницы данных и стека, имеющие атрибут «только для чтения». Если один из процессов попытается внести изменения в какую-либо страницу, система сгенерирует страничную ошибку. Обработчик ошибки в ответ создаст новую копию изменяемой страницы, изменит ее атрибут защиты на «чтение-запись», а также внесет изменения в соответствующие элементы PTE предка и потомка. В этом случае будут продублированы только те страницы памяти, которые необходимо изменить любому из двух процессов, что уменьшает затраты на создание нового процесса.

Реализация методики копирования при записи требует поддержания счетчика ссылок для каждой страницы памяти. Это стало одной из причин, из-за которой методика не стала использоваться в BSD UNIX. Вместо нее в системе реализован новый вызов vfork («виртуальный fork»), предлагающий альтернативный вариант решения проблемы.

Вызов vfork используется в тех случаях, когда после обычного fork предполагается выполнение exit или exec. Именно поэтому он не производит копирование адресного пространства. Вместо этого процесс-предок передает собственное адресное пространство потомку и затем переходит в режим сна до тех пор, пока не будет вызвана функция exec или exit. Позднее ядро будит родительский процесс, который «отбирает» свое бывшее адресное пространство у потомка. Для этого процесса создаются только собственная область u и структура proc. Передача адресного пространства реализована путем простого копирования регистров таблицы страниц от родителя к потомку. При этом не трогаются даже таблицы страниц. При выполнении vfork необходимо всего лишь изменить элементы PTE, отображающие область u процесса.

Системный вызов vfork является не слишком затратным для системы и работает намного быстрее копирования при записи. Его основным недостатком является тот факт, что процесс-потомок обладает правом изменения размера или содержимого адресного пространства родителя. Основная нагрузка по предупреждению возможных трудностей лежит на программисте, который должен сам следить за правильным использованием вызова vfork.

13.5.2. Обработка страничной ошибки

Существуют два типа страничных ошибок: ошибка корректности (validation fault) и ошибка защиты (protection fault). Ошибки корректности обычно возникают в тех случаях, если для страницы нет необходимого элемента PTE (*ошибка границ*) или такой элемент помечен как недействительный. Ошибки защиты возвращаются тогда, когда запрашиваемый тип доступа не разрешен для этой страницы. Если пользователь пытается обратиться к странице, элемент PTE которой одновременно некорректен и недоступен из-за наложенного на страницу ограничения прав доступа, это выльется в ошибку защиты (следует понимать, что система не обращает внимания на некорректность страницы, если доступ к ней пресекается).

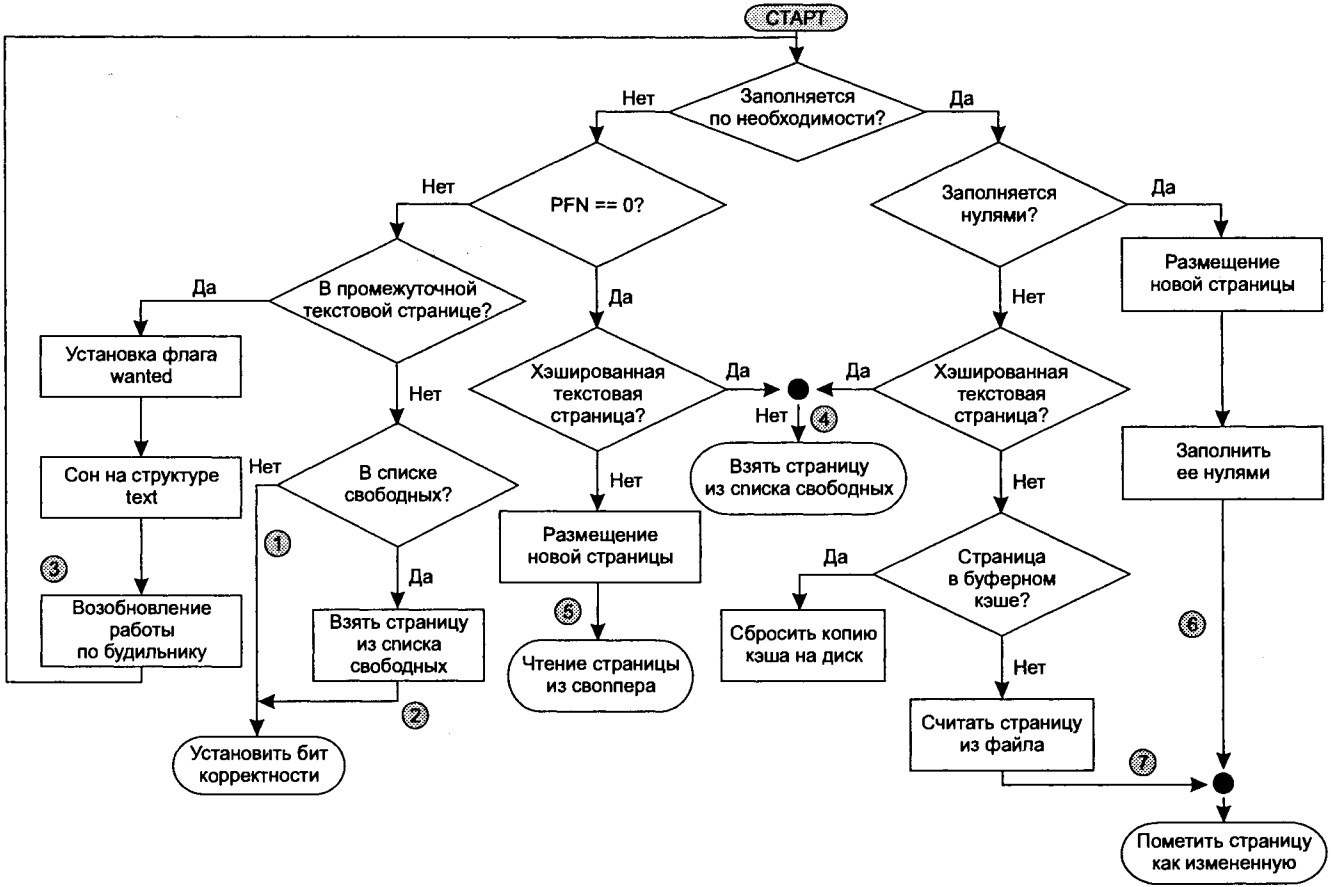


Рис. 13.18. Алгоритм процедуры pagein()

В случае возникновения любой из перечисленных ошибок система сохраняет информацию о состоянии, достаточную для повторного выполнения инструкции, а затем передает управление процедуре обработки ошибок ядра. В случае ошибки нарушения границ процесс, как правило, завершается путем передачи сигнала, кроме переполнения прикладного стека (при этом ядро вызывает процедуру, автоматически увеличивающую размер стека). Ошибки защиты практически всегда приводят к отправке сигнала процессу. Если система поддерживает технологию копирования при записи, то она должна уметь отслеживать такие ситуации и обрабатывать ошибку посредством создания новой копии страницы, доступной для записи. В остальных случаях для обработки ошибки вызывается процедура pagein().

Процедуре pagein() передается виртуальный адрес страницы (ставшей причиной ошибки), который получается из PTE. Если страница является резидентной (то есть элемент PTE, не заполняемый по необходимости, а номер страничного фрейма не равен нулю), то запрашивается и элемент структуры cmap для этой страницы. Обе переменные вместе содержат всю необходимую информацию о состоянии страницы и достаточны для функционирования процедуры pagein(). Алгоритм ее работы представлен на рис. 13.18. Существуют семь сценариев работы процедуры.

1. Элемент PTE может быть помечен как некорректный с целью имитации бита ссылки (см. раздел 13.5.3). В этом случае, если страница резидентна, а элемент cmap не маркирован как свободный, процедура просто установит бит корректности и завершит выполнение.
2. Страница является резидентной и при этом находится в списке свободных страниц. Этот случай сходен с вариантом 1, отличаясь от него только тем, что элемент cmap помечен как свободный. Процедура сбрасывает бит корректности и удаляет данный элемент cmap из списка свободных страниц.
3. Для текстовых страниц возможна ситуация, когда другой процесс пытается начать чтение страницы. Такая ситуация может сложиться, если два процесса, использующие совместно один и тот же участок текста, пытаются одновременно прочесть одну и ту же страницу. Последний процесс обнаружит, что номер страничного фрейма запрашиваемой страницы не равен нулю, при этом элемент карты ядра помечен как *блокированный* (locked) и находящийся в *стадии передачи* (in-transit). В этом случае процедура pagein() установит флаг wanted и заблокирует второй процесс, который будет «спать» по адресу (см. раздел 7.2.3) текстовой структуры страницы. После снятия блокировки страницы первым процессом (вслед за завершением чтения ее содержимого), им будет разбужен первый процесс. Так как второй процесс не всегда возобновляет работу сразу после пробуждения, он не обладает информацией о том, что страница уже находится в памяти, и выполняет ее поиск повторно.

4. Текстовые страницы могут находиться в памяти даже в том случае, если элементы PTE не владеют информацией об их номерах страничного фрейма. Такое случается в результате недавнего завершения выполнения другого процесса. Такие страницы находятся обращением к соответствующей таблице хэширования, где пара <устройство, номер блока> работает в качестве ключа. Если страница будет обнаружена, ее можно удалить из списка свободных страниц и использовать повторно.

   В оставшихся случаях страница не находится в памяти машины. После ее локализации процедуре `pagein()` необходимо сначала затребовать страницу из списка свободных страниц и затем произвести чтение данных тем или иным способом. Возможные варианты представлены ниже.

5. Страница расположена на устройстве свопинга. Бит заполнения по требованию сброшен, а номер страничного фрейма равен нулю, случай 4 неприменим. Для обнаружения страницы в области свопинга применяются карты свопинга, содержимое страницы читается в память из устройства свопинга.

6. Обработка страниц, заполняемых нулями, заключается в заполнении новой выделенной страницы нулями.

7. Если страница заполняется из текста и при этом не найдена в таблице хэширования (случай 4), то она будет считана из исполняемого файла. При этом загрузка производится напрямую из физической страницы, минуя буферный кэш. Однако это может привести к некорректности, если дисковая копия страницы окажется устаревшей. Для предупреждения такой ситуации ядро сначала проверит буферный кэш, и если обнаружит в нем нужную страницу, то сбросит ее дубликат на диск до того как считать данные для процесса. Такое решение требует двукратного обращения к диску и вследствие этого является неэффективным, однако продолжает поддерживаться системой. Наилучшим вариантом представляется прямое копирование страницы из буферного кэша в случае ее обнаружения в нем.

Для 5 и 6 новая страница памяти помечается как измененная с той целью, чтобы ее можно было сохранить на устройстве свопинга перед повторным использованием.

13.5.3. Список свободных страниц

Ядру часто приходится помещать новую страницу в основную память машины. Если памяти становится недостаточно, необходимо удалять некоторые страницы из нее. В системе 4.3BSD при выборе страниц для удаления из памяти срабатывает правило глобальной замены.

Очевидно, что наилучшими кандидатами на повторное использования являются те страницы, которые больше никому не потребуются, например, страницы стека и данных, принадлежащие завершившимся процессам. Именно на них нужно заострить внимание в первую очередь (и только после их исчерпания системе приходится обращать свой взор на потенциально необходимые страницы). Если в системе нет ненужных страниц, принцип локальности предписывает использовать страницы в порядке LRU. Недавно использовавшиеся страницы с наибольшей вероятностью могут потребоваться снова в отличие от тех, обращение к которым происходило относительно давно.

Однако не совсем целесообразно производить поиск наиболее подходящего кандидата каждый раз, когда процессу необходима новая страница памяти. Лучшим решением представляется поддержка некоего списка страниц, доступных для повторного применения. Если кому-то необходимо получить страницу памяти, то ее можно взять из такого списка. Список должен в любой момент времени содержать достаточное количество страниц и сортироваться так, чтобы в его голове всегда находились наиболее подходящие кандидаты. В идеале все ненужные страницы должны находиться в начале списка, после чего в порядке LRU можно расположить те страницы, для которых возможность повторного обращения не равна нулю. Такой список поддерживается ядром системы BSD. Минимальный и максимальный размер списка ограничиваются значениями переменных `minfree` и `maxfree`. Текущий размер списка описывается переменной `freemem`.

Строгое соответствие списка сортировке в порядке LRU является весьма актуальной проблемой. Достижение такой релевантности требует проведения упорядочивания списка всякий раз при каждой ссылке на страницу, то есть при выполнении всех команд, выполняющих обращение к прикладному адресному пространству. Это является слишком затратной операцией для системы, поэтому разработчики 4.3BSD нашли разумный компромисс. Они заменили правило последнего, недавно использовавшегося, на противоположное — *последнего, давно использовавшегося* [2]. Страница может считаться свободной, если к ней в недавнем времени не производилось обращений.

Данное правило может быть реализовано путем выполнения двойного обращения к каждой странице. На первом этапе происходит сброс бита *ссылки* в элементе PTE страницы. На втором этапе производится проверка бита ссылки. Если он по-прежнему оказывается сброшенным, страница считается подходящей для освобождения, так как за время контрольного замера никто не осуществлял доступа к ней. Такой алгоритм получил название *часов с двумя стрелками* (two-handed clock). Таблица `cmap` интерпретируется как циклическая (то есть ее конец соединен с началом), в ней имеются два указателя («стрелки») на фиксированном расстоянии друг от друга (то есть отстоящие через определенное количество элементов `cmap` (рис. 13.19)). Указатели инкрементируются синхронно. При прохождении первого указателя над элементом происходит сброс бита *ссылки*, а вторая «стрелка» произ-

водит проверку этого бита. Если бит окажется все так же в значении 0, это означает, что за время перемещения скользящего окна не произошло ни одного обращения к странице. Из чего делается вывод о том, что страница может быть освобождена. Если страница окажется «грязной» (заполненной данными), то перед ее освобождением следует произвести сброс информации на устройство свопинга.

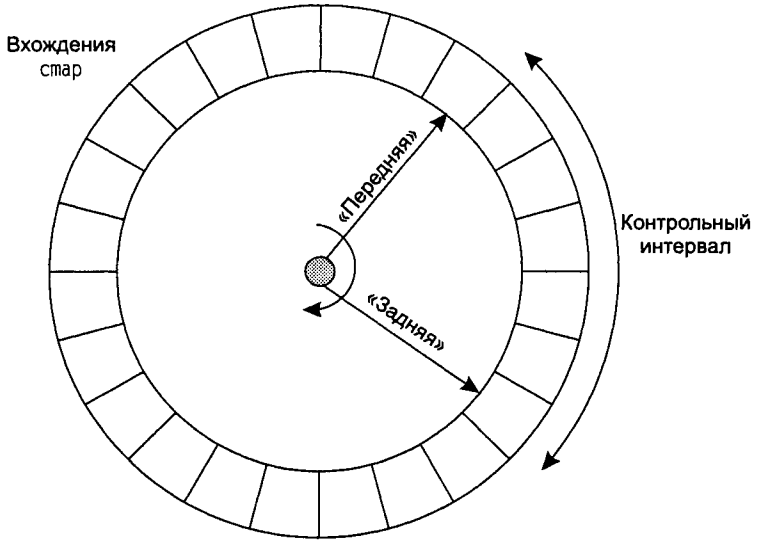


Рис. 13.19. Технология часов с двумя стрелками

Некоторые архитектуры, такие как VAX-11 или MIPS R3000, не поддерживают на аппаратном уровне бит *ссылки.* Этот недостаток можно преодолеть путем программной имитации бита. Для этого первая «стрелка» производит сброс бита корректности в PTE, что приводит к страничной ошибке при обращении к странице. Обработчик ошибки распознает такую ситуацию (см. случай 1 в предыдущем разделе) и устанавливает бит корректности обратно. Таким образом, если вторая «стрелка» обнаружит бит корректности сброшенным, это будет означать, что обращения к странице не было, и такая страница является претендентом на освобождение. Данное решение приводит к дополнительным перегрузкам системы из-за необходимости обработки страничных ошибок, создаваемых исключительно с целью отслеживания информации об обращениях на страницу.

За замену страниц отвечает отдельный процесс под названием pagedaemon (всегда являющийся процессом 2). Он позволяет сбрасывать страницы в область свопинга без блокировки процессов. Более того, процесс pagedaemon имеет право сброса страниц, относящихся к другим процессам, поэтому ему требуется перед началом операции отобразить такие страницы в своем адресном пространстве. Страницы, сбрасываемые в область свопинга в обход

буферного кэша, используют специализированный набор заголовков буфера свопинга. Операции записи производятся асинхронно, поэтому процесс pagedaemon вправе в оставшееся время выполнять проверку других страниц. По окончании операции записи процедура завершения переносит такие страницы в список очищенных, из которого они могут быть возвращены в список свободных страниц процедурой под названием cleanup().

13.5.4. Свопинг

Несмотря на то, что производительность работы страничной системы, как правило, высока, при сильной загрузке она может уменьшиться. Основная проблема, именуемая *захламлением* (thrashing), возникает в том случае, если в системе недостаточно памяти для рабочих наборов активных процессов. Это может произойти, если в системе слишком много активных процессов или их страничная конфигурация чересчур произвольна (вследствие чего их рабочие наборы имеют большие размеры). При перегрузках растет количество страничных ошибок. Если результатом обращения к странице становится ошибка, то при ее обработке такая страница заменит собой другие страницы, являющиеся частью рабочего набора активного процесса, что еще больше усугубит описанную проблему. В такой ситуации система будет тратить много времени на обработку страничных ошибок, замедляя тем самым дальнейшее выполнение процессов.

Одним из вариантов выхода из проблемной ситуации является уменьшение общего количества активных процессов в целях контроля загруженности системы. Деактивированные процессы не могут стать выполняемыми. Очевидно, что память, используемая такими процессами, должна быть по возможности освобождена путем копирования страниц в область свопинга. Такая операция получила название «выгрузка процесса». При спаде загрузки системы деактивированные процессы могут быть опять загружены в оперативную память машины.

Задача отслеживания загрузки системы лежит на специализированном процессе под названием swapper. Именно он производит выгрузку и загрузку процессов. При изначальной инициализации системы ядро создает процесс с идентификатором PID 0, который затем вызывает sched() — основную функцию процесса swapper. Таким образом, swapper становится процессом «0». Обычно он дремлет, просыпаясь периодически для проверки состояния системы и проведения некоторых необходимых действий. Демон swapper выгружает процесс из памяти в следующих случаях:

- фрагментация Userptmap. Если секция Userptmap таблицы PTE переполнена или слишком фрагментирована, процесс не в силах выделить элементы PTE для отображения собственных таблиц страниц. Такая ситуация может сложиться при вызове fork, exec или увеличении областей памяти. При ее возникновении swapper выгрузит один из процессов для освобождения места в секции Userptmap;

- уменьшение объема свободной памяти. Переменная freemem содержит количество кластеров в списке свободных страниц. Если значение переменной в течение некоторого периода времени близко к нижней границе, это означает, что страничная система перегружена. Для снятия «стресса» загружается процесс swapper;
- неактивные процессы. Если какой-либо процесс продолжает оставаться неактивным в течение длительного временного промежутка (больше 20 секунд), то он, скорее всего, не будет пытаться изменить в дальнейшем свое состояние и, следовательно, может быть выгружен из памяти. Например, пользователь способен покинуть рабочее место, забыв произвести выход из системы, оставив неактивным командный интерпретатор. Несмотря на то, что его резидентные страницы через некоторое время все равно будут перенесены в область свопинга, выгрузка процесса целиком является более целесообразной, так как позволит освободить такие важные ресурсы системы, как вхождения Userptmap.

По каким критериям swapper производит выбор процесса для выгрузки из памяти? Идеальным кандидатом для этого является процесс, находящийся в режиме сна более 20 секунд. Если в системе нет такого процесса, то swapper находит четыре наиболее «крупных» процесса и выгружает тот из них, который находится резидентно в памяти дольше других. Оставшиеся три процесса могут быть выгружены, если системе понадобится свободная дополнительная память.

При выгрузке процесса демон swapper осуществляет следующие действия:

- выделяет пространство свопинга для области u, стека ядра и таблиц страниц;
- открепляет процесс от своей текстовой области. Если при этом не остается процессов, использующих область, текст, как правило, также выгружается в пространство свопинга;
- сохраняет резидентные страницы данных и стека в области свопинга, затем туда же копируются таблицы страниц и, наконец, область u и стек ядра;
- освобождает системные элементы PTE в секции Userptmap, отображающие таблицы страниц выгружаемого процесса;
- записывает местонахождение в области свопинга области u и структуры proc.

Если один или несколько процессов были выгружены из памяти, процесс swapper периодически начнет проверять систему на предмет возможности возврата их в основную память. Время проведения обратных операций зависит от количества свободной памяти и наличия места в секции Userptmap. Если в системе были выгружены в область свопинга несколько процессов, демон swapper присваивает каждому из них *приоритет загрузки* (swapin

priority), высчитываемый исходя из «размера» процесса, значения *фактора любезности* (см. раздел 5.4.1), времени нахождения в области свопинга и длительности спячки. Для загрузки в память выбирается процесс с наивысшим приоритетом из всех.

Процедура загрузки процесса в память является примерно противоположной по отношению к его выгрузке в область свопинга. Процесс подключается к области текста, выделяются элементы PTE в секции Userptmap для отображения его таблиц страниц. Для области u, стека ядра и таблиц страниц (их содержимое считывается из области свопинга) выделяется физическая память. После этого освобождается пространство свопинга, отведенное под все эти области. Затем процесс помечается как выполняемый и добавляется в очередь планирования. Страницы данных и стека могут быть получены из области свопинга уже после начала работы процесса, если это окажется необходимо.

13.6. Анализ

Подсистема управления памятью в BSD предоставляет широкие возможности при небольшом количестве базисных элементов. Единственным аппаратным требованием для системы является поддержка страниц, заполняемых по необходимости, так как BSD не пользуется сегментацией. Однако система обладает и некоторыми слабостями и недоработками, о которых не следует забывать.

- Отсутствие поддержки выполнения удаленных программ (по сети). Причиной этому является неготовность файловой системы BSD к осуществлению доступа к удаленным файлам. Если файловая система обладает такой возможностью, соответствующее расширение подсистемы управления памятью является несложной задачей.
- Отсутствие режима разделения памяти (кроме текстовых областей, доступных «только для чтения»). В частности, система BSD не имеет средств разделения памяти, похожих на существующие в System V.
- Вызов vfork не является полной заменой fork. Отсутствие копирования при записи отрицательно влияет на производительность приложений, часто использующих fork. Как пример таких приложений можно привести демоны и другие серверные программы, создающие процессы посредством fork при обработке каждого входящего запроса.
- Каждый процесс должен иметь собственную копию таблицы страниц для совместно используемой текстовой области. Это приводит не только к излишним затратам памяти на их размещение, но и к потере времени на синхронизацию таблиц при добавлении изменений в PTE, сделанных другим процессом, использующим ту же текстовую область.

- Отсутствует поддержка файлов карт памяти. Более подробно об этом средстве можно прочесть в разделе 14.2.
- Не поддержаны совместно используемые библиотеки.
- Не решена проблема отладки одной программы, выполняемой несколькими процессами. Если отладчик устанавливает в такой программе контрольную точку, то он изменяет этим соответствующую текстовую страницу. Эти изменения видны всей группе процессов, что может привести к неожиданным результатам. Для предупреждения возможных конфликтов система не позволяет устанавливать контрольные точки в текстах программ, используемых одновременно несколькими процессами, и запрещает новым процессам выполнять программы, находящиеся в стадии отладки. Очевидно, что такое решение проблемы является совершенно неудовлетворительным.
- Реализация системы BSD резервирует для каждого процесса достаточное пространство свопинга, позволяющее выгрузить любую страницу из адресного пространства процесса. Такое правило гарантирует, что нехватка пространства свопинга процесса может наблюдаться только в том случае, если его объем необходимо увеличить (или в результате вызова fork или exec). Проблема не может возникнуть во время обычной работы процесса. Описанное решение требует поддержки области свопинга большой величины. С другой стороны, пространство свопинга системы ограничивает размер выполняемой программы.
- Отсутствует поддержка пространства свопинга для удаленных узлов, которая может быть необходима в некоторых случаях, например, на бездисковых станциях.
- Структура системы оптимизирована для архитектуры VAX и сильно зависит от нее. Это делает систему неподходящей для выполнения на большом количестве машин, на которые уже были перенесены другие варианты UNIX. Более того, код BSD буквально пронизан конструкциями, зависящими от конкретной аппаратной архитектуры, что увеличивает затраты по переносу системы на другие платформы.
- Исходные коды системы не являются модульными, что усложняет задачу добавления новых возможностей в ОС или изменения отдельных компонентов или правил. Например, хранение номера блока файловой системы в некорректных (то есть заполняемых по необходимости) элементах PTE не дает возможности сделать задачи преобразования адресов и выделения страниц самостоятельными.

Несмотря на перечисленные минусы системы, структура 4.3BSD является основой для построения современных архитектур памяти, реализованных в таких ОС, как SVR4, 4.4BSD или Mach. О них мы расскажем в следующих главах. В перечисленных системах сохранилась поддержка многих методов

BSD, но была изменена базовая структура с целью предоставления большего количества возможностей и избавления от многих ограничений, присущих подсистеме памяти BSD.

Архитектура 4.3BSD наиболее подходила для систем 80-х годов, которые, как правило, обладали медленными процессорами и малым объемом оперативной памяти, но имели диски относительно большого объема. Именно поэтому алгоритмы системы были оптимизированы для уменьшения использования памяти за счет дополнительных операций дискового ввода-вывода. Однако в 90-е годы типичные системы обладали большим объемом памяти и быстрыми процессорами, но при этом объемы их дисков были относительно малы. Большинство файлов пользователей стало храниться в отдельных файл-серверах. Модель памяти 4.3BSD не пригодна для таких машин. В следующей версии, 4.4BSD, была представлена новая архитектура управления памятью, основанная на концепциях ОС Mach. Подробнее о ней можно прочесть в разделе 15.8.

13.7. Упражнения

1. Каким требованиям из списка, представленного в разделе 13.2.1, может удовлетворять система, использующая свопинг как основной механизм управления памятью?
2. Опишите преимущества технологии загрузки страниц по запросу по сравнению с методикой сегментации.
3. Почему системы UNIX используют упреждающий страничный обмен? Назовите недостатки этой технологии.
4. Какие преимущества и недостатки свойственны копированию текстовых страниц в область свопинга?
5. Представьте, что исполняемая программа находится на удаленном узле. Как вы думаете, будет ли лучше произвести копирование ее образа в локальную область свопинга перед выполнением программы?
6. Аппаратная часть машины и операционная система взаимодействуют друг с другом при преобразовании виртуальных адресов. Как распределены обязанности между ними? Чем отличается ответ на этот вопрос при рассмотрении конкретных аппаратных архитектур, описанных в разделе 13.3?
7. Сравните преимущества и недостатки правила глобального замещения страниц по сравнению с «локальным» правилом.
8. Что должен предпринять разработчик ПО для уменьшения рабочего набора своего приложения?
9. Назовите преимущества инвертированных таблиц страниц.

10. Почему в MIPS R3000 возникает большое количество страничных ошибок? Перечислите преимущества этой архитектуры, позволяющие сгладить недостаток по обработке дополнительных страничных ошибок.
11. Представьте, что в системе 4.3BSD один из процессов становится причиной страничной ошибки для одновременно нерезидентной и защищенной страницы (процесс не обладает выбранным типом доступа). Что в этом случае в первую очередь проверит обработчик ошибок? Какие действия он предпримет?
12. Почему карты ядра управляют только теми страницами, которые находятся в страничном пуле (paged pool)?
13. Что понимается под значением термина «имя страницы»? Всегда ли страница имеет одно и то же имя? Чем отличаются друг от друга пространства имен страниц в 4.3BSD?
14. Каков минимальный объем пространства свопинга в системе 4.3BSD. Какими преимуществами обладает использование области свопинга «гигантских» размеров?
15. Какие факторы ограничивают максимально доступное виртуальное адресное пространство процесса? Почему для процесса важно экономно использовать выделенную виртуальную память?
16. Считаете ли вы, что размещение области свопинга сразу на нескольких дисках является правильным? Аргументируйте свою точку зрения.
17. Почему точное следование правилу LRU не совсем подходит для реализации задачи замены страниц?
18. Первые реализации системы BSD ([2], [13]) использовали алгоритм «часов с одной стрелкой», сбрасывая бит ссылки на первом проходе и выгружая из памяти страницы, чей бит ссылки оказался по-прежнему сброшен, при следующем просмотре. Чем этот алгоритм хуже по сравнению с технологией часов с двумя стрелками?

13.8. Дополнительная литература

1. Babaoglu, O., Joy, W. N., and Porcar, J., «Design and Implementation of the Berkeley Virtual Memory Extensions to the UNIX Operating Systems», Technical Report, CS Division, EECS Department, University of California, Berkeley, CA, Dec. 1979.
2. Babaoglu, 0., and Joy, W. N., «Converting a Swap-Based System to Do Paging in a Architecture Lacking Page-Referenced Bits», Proceedings of the Eight ACM Symposium on Operating Systems Principles, Dec. 1981, pp. 78–86.

3. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
4. Bakoglu, H. B., Grohoski, G. F., and Montoye, R. K., «The IBM RISC system/6000 Processor: Hardware Overview», IBM Journal of Research and Development, Vol. 34, Jan. 1990.
5. Belady, L. A., «A Study of Replacement Algorithms for Virtual Storage Systems», IBM Systems Journal, Vol. 5, No. 2, 1966, pp. 78–101.
6. Chakravarty, D., «Power RISC System/600 — Concepts, Facilities, and Architecture», McGraw-Hill, 1994.
7. Collinson, P., «Virtual Memory», SunExpert Magazine, Apr. 1991, pp. 28–34.
8. Digital Equipment Corporation, «VAX Architecture Handbook», Digital Press, 1980.
9. Denning, P. J., «Virtual Memory», Computing Surveys, Vol. 2, No. 3, Sep. 1970, pp. 153–189.
10. Intel Corporation, «80386 Programmer's Reference Manual», 1986.
11. Kane, G., «Mips RISC Architecture», Prentice-Hall, Englewood Cliffs, NJ, 1988.
12. Lee, R. B., «Precision Archilecture», IEEE Computer, Vol. 21, No. 1, Jan. 1989, pp. 78–91.
13. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3 BSD UNIX Operating System», Addison-Wesley, Reading, MA, 1989.
14. Robboy, D., «A UNIX Port to the 80386», UNIX Papers for UNIX Developers and Power Users, The Waite Group, 1987, pp. 400–426.
15. SPARC International, «SPARC Architecture Manual Version 8», 1991.

Глава 14
Архитектура VM системы SVR4

14.1. Предпосылки появления новой технологии

Архитектура управления памятью VM (Virtual Memory) была представлена корпорацией Sun Microsystems в своей системе SunOS 4.0. Предыдущие версии этой системы вобрали в себя принципы модели управления памятью BSD и вследствие этого обладали всеми недостатками, описанными в предыдущей главе. В частности, от системы SunOS требовалась поддержка разделения памяти, совместно используемых библиотек и файлов, отображаемых в памяти, что нельзя было реализовать без внесения существенных изменений в архитектуру BSD. Более того, система SunOS создавалась для работы на различных аппаратных платформах (Motorola 680x0, Intel 386 и систем SPARC, выпускаемых компанией Sun). Именно поэтому ей требовалась легко переносимая архитектура памяти. Технология VM получила широкое распространение. Позднее объединенная команда разработчиков из AT&T и Sun Microsystems, трудившаяся над созданием SVR4 UNIX, стала базироваться именно на этой технологии, отказавшись от сегментной архитектуры (regions) SVR3.

Центральной частью архитектуры VM является понятие отображения файлов (file mapping). Этот термин используется для описания двух отличных друг от друга сущностей. С одной стороны, отображение файлов является мощным средством для пользователей, позволяющим им проецировать файл на часть адресного пространства и в дальнейшем использовать простые команды доступа к памяти для осуществления чтения и записи файла. Во-вторых, оно также годится для использования в качестве основной схемы организации ядра, которое может видеть все свое адресное пространство в виде набора проекций на объекты различного типа, такие как файлы. В архитектуре SVR4 нашли применение оба приведенных выше аспекта абстракции

отображения файлов[1]. Перед тем как начать описание архитектуры VM, остановимся на таком понятии, как файлы, отображаемые в памяти, и рассмотрим их область применения и степень важности.

14.2. Файлы, отображаемые в памяти

Традиционной методикой работы с файлами в UNIX является их открытие посредством системного вызова open и вызов функций read, write, seek и lseek для осуществления последовательного или произвольного ввода-вывода. Такая методика работы с файлами является неэффективной, так как требует отдельного системного вызова (или даже двух — для произвольных операций) при каждой необходимости чтения-записи. Более того, если несколько процессов осуществляют доступ к одному и тому же файлу, каждый из них копирует данные этого файла в свое адресное пространство, что приводит к излишнему расходованию памяти. На рис. 14.1 схематично очерчена ситуация чтения двумя процессами одной и той же страницы файла. При этом необходимо произвести чтение с диска для копирования страницы в буферный кэш, а также операции копирования данных из буфера в адресное пространство процесса (количество таких операций равняется количеству процессов, желающих прочесть файл). Более того, в результате проведенных действий в памяти оказываются три копии страницы: одна находится в буферном кэше и по одной размещается в адресном пространстве каждого процесса. Ко всему прочему, следует заметить, что для чтения информации каждому процессу нужно отдельно вызвать read (и дополнительно lseek, если предполагается произвольный доступ).

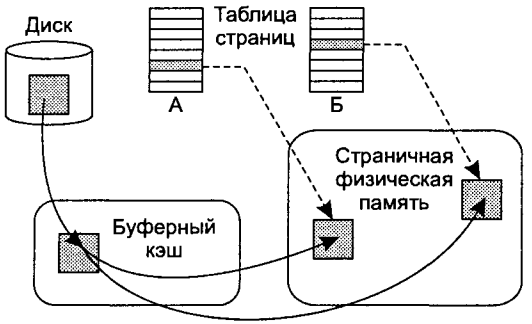


Рис. 14.1. Чтение двумя процессами одной и той же страницы файла в традиционных вариантах UNIX

[1] Эти два аспекта являются независимыми друг от друга. Например, система HP-UX 9.x поддерживает отображение файлов на прикладном уровне, имея при этом традиционную организацию ядра. В противоположность ей AIX 3.1 использует карты памяти как базовую стратегию ввода-вывода, но не передает ее на прикладной уровень (нет поддержки вызова mmap).

Рассмотрим альтернативный вариант, при котором процессы отображают ту же страницу в своем адресном пространстве (рис. 14.2). Ядро создает такое отображение путем простого обновления некоторых структур управления памятью. Если процесс А пытается обратиться к странице, генерируется страничная ошибка. Ядро исправляет ее путем считывания страницы в память и модификации таблицы страниц так, чтобы в ней появилось указание на эту страницу. Если впоследствии процесс Б захочет обратиться к странице (что снова вызовет ошибку), раз она уже находится в памяти, то обработка ситуации будет заключаться в изменении ядром таблицы страниц процесса Б.

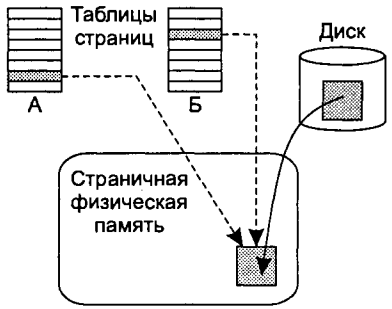


Рис. 14.2. Отображение двумя процессами одной и той же страницы файла в их собственных адресных пространствах

Пример показывает очевидные преимущества обращения к файлам путем отображения их в памяти. Общие затраты на две операции чтения заключаются в единственном обращении к диску. После установки отображения для чтения или записи данных больше не потребуется вызывать системные функции. В памяти машины будет храниться только одна копия страницы, что позволяет сохранить свободным объем, равный двум страницам, и отказаться от двух операций копирования данных в памяти. Сокращение числа действий с физической памятью приводит и к уменьшению страничных операций.

Что происходит в том случае, если процесс производит запись в отображенную страницу? Процесс может установить два типа отображения файлов: *разделяемое* (shared) и *закрытое* (private). В первом случае все изменения производятся в самом отображаемом объекте. Ядро передает изменения непосредственно разделяемой копии страницы и сбрасывает их на диск в файле при удалении страницы. Если отображение является закрытым, то все изменения будут отражены только в закрытой копии страницы, к которой они относятся. При этом операции записи не задевают исходный объект, следовательно, ядро не сбрасывает измененные данные в файл при удалении страницы из памяти.

Важно упомянуть, что закрытые отображения не защищены от возможности внесения изменений другими процессами, разделяющими отображение файла. Процесс получает закрытую копию страницы только в том случае,

если пытается модифицировать ее. При этом он видит все обновления, внесенные другими процессами за промежуток времени между настройкой отображения и попыткой записи данных на страницу.

Ввод-вывод файлов, отображаемых в памяти, является мощным механизмом системы, позволяющий осуществлять эффективный доступ к файлам. Однако эта технология не может полностью заменить традиционные вызовы read и write. Одним из важных моментов является неделимость ввода-вывода. При передаче данных системные вызовы read и write блокируют индексный дескриптор файла, что гарантирует неделимость этих операций. Файлы, отображаемые в памяти, доступны при помощи обычных программных команд, что дает возможность считывать или записывать за один раз, по крайней мере, одно слово. Такому типу доступа не подходит ни один из классических вариантов семантики блокировки файлов и задача синхронизации полностью ложится на взаимодействующие процессы.

Еще одним важным отличием новой технологии является степень видимости изменений. Если несколько процессов разделяют между собой отображение файла, то все изменения в странице, произведенные одним из них, сразу же становятся видны остальным взаимодействующим процессам.[1] Такой подход совсем не похож на традиционную модель, где процесс мог видеть изменения только после повторного вызова read. При использовании отображения файлов процесс будет незамедлительно получать обновленное содержимое страницы при каждом обращении к ней (а не только после первоначального создания образов файлов).

С другой стороны, после появления технологии отображения разработчикам приложений прибавилось хлопот по принятию решения, стоит ли использовать отображаемые файлы. Вышеперечисленные свойства и стали главным камнем преткновения. Разработчики не сразу воспользовались преимуществами технологии. Например, для системы 4.3BSD расписан интерфейс вызова mmap, но при этом не имеется его реализации. Об интерфейсе вызова mmap будет рассказано в следующем разделе.

14.2.1. Системный вызов mmap

Для отображения файла в память необходимо сначала произвести его открытие при помощи open и затем вызвать mmap. Системный вызов mmap имеет следующий синтаксис:

```
paddr = mmap (addr, len, prot, flags, fd, offset);
```

[1] Понятие «сразу же» применимо к однопроцессорным системам, где гарантируется *строгая* модель консистентности данных. На многопроцессорных системах она не может быть обеспечена только из-за того, что соответствующая разница в моментах времени доступа требует при передаче сигнала многократного превышения скорости света, и приходится ограничивать последовательность событий чтения/записи. — *Прим. ред.*

Результатом работы функции становится создание области с образом[1] размером [offset, offset+len) файла fd в адресном пространстве [paddr, paddr+len) процесса. Параметр flag указывает на тип отображения и может иметь значения MAP_SHARED и MAP_PRIVATE. Переменная prot устанавливается как любая комбинация из следующих возможных значений: PROT_READ, PROT_WRITE и PROT_EXECUTE. Некоторые системы, не поддерживающие привилегии выполнения, приравнивают флаги PROT_EXECUTE и PROT_READ.

Согласованное значение paddr выбирается системой. Оно не может быть равным нулю, а образ не вправе накладываться на уже существующие отображения. Вызов mmap игнорирует параметр addr, если не установлен флаг MAP_FIXED[2]. В этом случае значение paddr должно совпадать с addr. Если значение addr окажется неподходящим (например, не находится в диапазоне корректных адресов процесса), работа вызова завершится с ошибкой. Применение флага MAP_FIXED является нежелательным, так как созданная программа становится непереносимой на другие платформы.

Системный вызов mmap работает с целыми страницами памяти. Это означает, что параметр offset должен быть выровнен по величине страницы. При указании флага MAP_FIXED этому требованию должен соответствовать и параметр addr. Если значение len не совпадает с целым числом страниц, система сама произведет округление до следующего целого числа.

Отображение будет поддерживаться системой до тех пор, пока не будет выполнен вызов

```
munmap (addr, len);
```

либо выполнено замещение адресного диапазона другим файлом при помощи вызова mmap с флагом MAP_RENAME. Для изменения признака защиты страницы используется системный вызов

```
mprotect(addr, len, prot);
```

14.3. Основы архитектуры VM

Архитектура VM [5] базируется на определении *объекта памяти* (memory object), являющегося программным представлением связи между областью памяти и резервным носителем. В системе используются несколько различных видов таких носителей, в том числе пространство свопинга, локальные или удаленные файлы и буферы фреймов. Система VM видит такие объекты

[1] В этой книге используются стандартные обозначения диапазонов: квадратная скобка означает включение граничного значения в массив, круглая — исключение значения.

[2] В описании семантики вызова mmap рекомендовано использование addr, если не установлен флаг MAP_FIXED.

одинаково и выполняет над ними один и тот же набор операций: например, выбирает страницу из такого носителя или, наоборот, сбрасывает страницу на резервный носитель. В противоположность системе VM внутри носителя упомянутые операции могут быть реализованы по-разному. Следовательно, система определяет общий интерфейс, а каждый носитель — его собственную реализацию.

ПРИМЕЧАНИЕ

В этой главе термин «объект» употребляется для обозначения двух понятий. *Объект памяти* представляет образ, в то время как *объект данных* является одним из носителей, например, файлом. Правильное значение термина в большинстве случаев вытекает из контекста повествования, но в неочевидных случаях мы будем использовать полные названия (объект памяти или объект данных).

Архитектура VM является объектно-ориентированной. Основные концепции ООП (применительно к системам UNIX) были представлены в разделе 8.6.2. Используя принятую в концепции объектно-ориентированного подхода терминологию, общий интерфейс к объекту памяти представляет собой *абстрактный базовый класс* [4]. Каждый тип объектов памяти (относящийся к каждому типу носителя) является порожденным классом (или классом-наследником) базового класса. Каждое отдельное отображение является *экземпляром* (или *объектом*) соответствующего порожденного класса.

Адресное пространство процесса содержит множество образов различных объектов данных. При этом корректными адресами объекта считаются только те, которые отображены. Объект предоставляет постоянное место хранения для страниц, отображенных на него. Отображение делает объект доступным для процесса путем прямой адресации. Отображаемый объект сам по себе ничего не «осознает» и никак не изменяется при создании образа.

Пространство имен объектов памяти, а также механизмы доступа к их данным, предоставляются файловой системой. Взаимодействие VM с файловой системой осуществляется на *уровне vnode*. Между объектами памяти и объектами vnode существует взаимосвязь по типу «много к одному». Каждый именованный объект памяти ассоциируется с уникальным объектом vnode, при этом один и тот же объект vnode вправе быть ассоциирован с несколькими объектами памяти. Некоторые объекты памяти, например, прикладные стеки, не могут быть ассоциированы с файлами и не могут иметь имен. Такие объекты представлены *анонимными* объектами (anonymous).

Физическая память выступает как кэш данных отображаемых объектов. Ядро пытается хранить большинство используемых страниц в основной памяти машины, минимизируя тем самым общее количество страничных операций.

Память разбита на страницы, которые являются минимальными единицами для операций размещения, защиты, преобразования адресов и отображения. Адресное пространство является обычным массивом страниц, при этом отдельные страницы относятся именно к нему, а не к объектам данных. На более высоком уровне при помощи фундаментального понятия страницы могут быть реализованы такие абстракции, как области памяти (regions) и т. д.

Архитектура VM является независимой от системы UNIX. Все принятые в этой системе определения, такие как области текстов, данных и стека, находятся над уровнем базовой подсистемы VM. Такой подход позволяет применять коды VM в операционных системах, не совместимых с UNIX. Разработчики VM сделали архитектуру простой для переноса на различные аппаратные платформы, поместив все зависимые от аппаратуры коды на отдельный уровень аппаратного преобразования адресов (hardware address translation layer, HAT), который доступен через четко определенный интерфейс.

Ядро по возможности применяет технологию копирования при записи с целью уменьшения числа операций перемещения данных внутри основной памяти, а также минимизации копий страниц в памяти. Применение этой технологии становится необходимым, если процессы обладают закрытыми отображениями объектов, так как в этом случае любые изменения не влияют ни на исходный объект, ни на остальные процессы, разделяющие страницу памяти.

14.4. Основные понятия

Архитектура VM [7] базируется на пяти фундаментальных абстракциях для описания подсистемы памяти:

- **страница** (page) — структура page;
- **адресное пространство** (address space) — структура as;
- **сегмент** (segment) — структура seg;
- **аппаратное преобразование адресов** (hardware address translation layer) — структура hat;
- **анонимная страница** (anonymous page) — структура an.

Эти понятия имеют объектно-ориентированный интерфейс доступа друг к другу, а также к остальной части ядра. Основные концепции объектно-ориентированного подхода см. в разделе 8.6.2. Система VM работает в тесном взаимодействии с файловой системой на уровне vnode (см. раздел 8.6), а также с устройствами свопинга на уровне свопинга. Основные взаимосвязи между этими уровнями показаны на рис. 14.3.

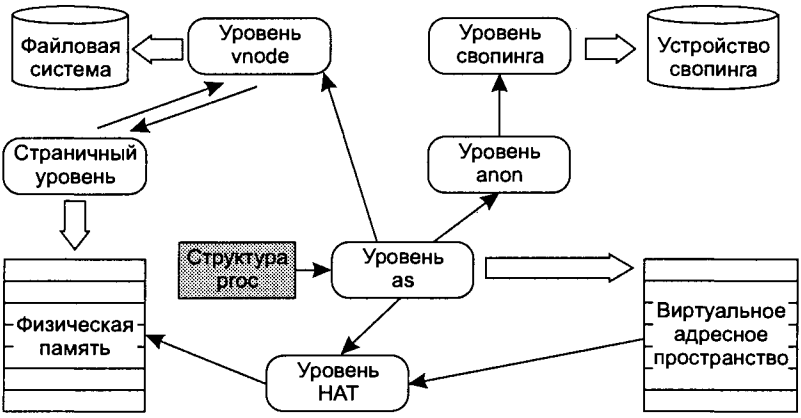


Рис. 14.3. Архитектура VM операционной системы SVR4

14.4.1. Физическая память

Как и в системе 4.3BSD, физическая память делится на страничную и нестраничную области. Страничная область описывается массивом структур page, каждая из которых представляет одну логическую страницу (кластер или аппаратные страницы). Структура page (рис. 14.4) сильно отличается от структуры cmap, поддерживаемой системой BSD. Так как физическая память представляет собой кэш страниц объектов памяти, в ней необходимо хранить информацию об управлении кэшем. В этой структуре также имеется информация для механизма преобразования адресов.

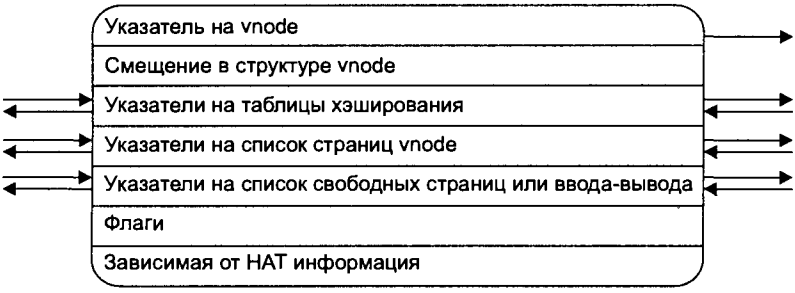


Рис. 14.4. Структура page

Каждая страница памяти сопоставлена объекту памяти, а каждый такой объект представлен при помощи vnode. Следовательно, имя физической страницы определяется как пара <vnode, offset>, то есть смещением страницы в объекте, представленном vnode. Это позволяет каждой странице обладать собственным уникальным именем даже в том случае, если она используется несколькими процессами. Смещение и указатель на vnode хранятся в структуре page.

Каждая страница фиксируется в нескольких двунаправленных связанных списках, для поддержки такой организации в структуре page предусмотрены три пары указателей. Для быстрого обнаружения все страницы хэшируются на основе vnode и смещения. Каждая структура page связана с одной из цепочек хэша. Любой объект vnode также поддерживает список всех своих страниц, находящихся в данный момент в физической памяти (с помощью второй пары указателей структуры page). Этот список используется процедурами, работающими одновременно со всеми страницами файла, находящимися в памяти. Например, при удалении файла ядру необходимо сбросить флаг корректности всех страниц этого файла, расположенных в физической памяти. Последняя пара указателей предназначена для указания на факт нахождения структуры в списке свободных страниц, либо в списке страниц, ожидающих записи на диск. Структура page не может в один момент времени быть в обоих списках.

В структуре page поддерживается счетчик ссылок, отображающий количество процессов, совместно использующих страницу на основе техники копирования при записи. Для синхронизации используется набор флагов (locked, wanted, in-transit). В структуре также хранятся копии битов *изменения* и *ссылки* (по информации HAT), а также некоторые поля, зависящие от HAT, служащие для обнаружения всех преобразований для этой страницы (см. раздел 14.4.5).

Со структурой page применяется интерфейс низкого уровня, позволяющий осуществлять такие процедуры, как поиск страницы (на основе смещения и vnode), перенос или удаление из структур хэширования и списка свободных страниц, а также синхронизацию доступа.

14.4.2. Адресное пространство

На рис. 14.5 показаны структуры данных, описывающие виртуальное адресное пространство процесса. Само адресное пространство (структура as) является главной абстракцией каждого процесса, обеспечивающей высокоуровневый интерфейс доступа к пространству процесса. Указатель на структуру as содержится в структуре proc каждого процесса. Структура as хранит заголовок связанного списка образов процесса, каждый из которых, в свою очередь, описывается структурой seg. Образ представляет собой не перекрывающие друг друга диапазоны адресов, разбитые на страницы, сортируемые по базовым адресам. Структура hat также является частью as. Структура as содержит рекомендацию (hint) — данные о последнем сегменте, вызвавшем страничную ошибку, а также другую полезную информацию, например, флаги синхронизации и размеры адресного пространства и резидентного набора.

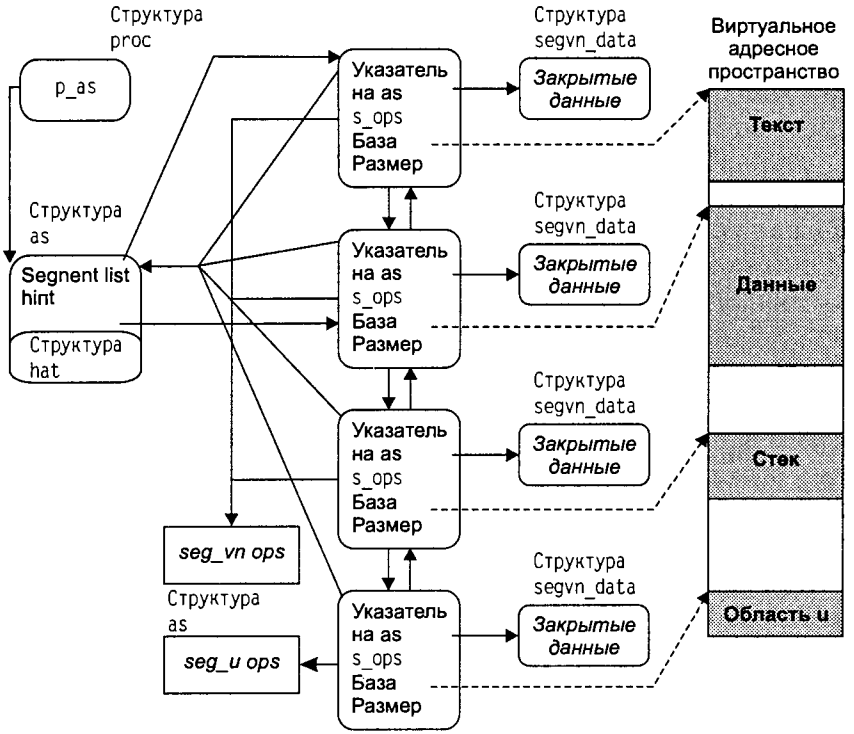


Рис. 14.5. Схема адресного пространства процесса

На *уровне as* поддерживается два базовых набора операций. Первый из них состоит из операций, производящих действия со всем адресным пространством, в том числе:

- as_alloc() — используется fork и exec для выделения нового адресного пространства;
- as_free() — вызывается fork или exit для освобождения адресного пространства;
- as_dup() — используется fork для дублирования адресного пространства.

Второй набор функций имеет дело с некоторыми наборами страниц внутри as и включает в себя следующие операции:

- as_map() и as_unmap() — применяется для определения объектов в структуре as и удаления из нее (вызывается mmap, munmap и некоторыми другими процедурами);
- as_setprot()и as_checkprot() — вызывается mprotect для установки и проверки защиты частей структуры as;
- as_fault() — начальная точка обработки страничных ошибок;
- as_faulta() — используется при упреждающем страничном обмене (fault ahead).

Реализация многих из перечисленных функций заключается в обнаружении образа и последующем вызове низкоуровневых процедур интерфейса отображения, описываемых в следующем разделе.

14.4.3. Отображение адресов

Адресное пространство является набором объектов памяти, представляющих собой результаты отображения между носителями и областями адресов процесса. Каждый такой образ называется *сегментом* (структура seg). Однако такой термин может показаться не совсем верным, поскольку описываемые сегменты не имеют никакого отношения к аппаратно распознаваемым сегментам сегментированных архитектур памяти. В технологии VM сегмент является объектом памяти, представляющим собой неразрывный диапазон виртуальных адресов процесса, сопоставленный байтовой области объекта данных и имеющий один и тот же тип отображения (закрытый или разделяемый).

Все сегменты используют одинаковый интерфейс взаимодействия с остальной частью подсистемы VM. В терминах объектно-ориентированного подхода такой интерфейс определяет *абстрактный базовый класс*. Существуют несколько различных типов сегментов, каждый из которых является *наследником* базового класса. Система VM также обладает набором общих функций для выделения и освобождения сегментов, присоединения их к адресному пространству и обратного действия.

Структура seg содержит открытые (или независимые от типа сегмента) поля, в которые входят базовый адрес и размер отображаемого адресного диапазона и указатель на структуру as, к которой диапазон относится. Все сегменты адресного пространства учитываются в двунаправленном связанном списке, сортируемом по базовым адресам (сегменты не могут перекрывать друг друга). Структура as указывает на первый сегмент. В каждой структуре seg хранятся указатели на предыдущий и последующий элементы списка.

Структура seg также обладает указателем на вектор seg_ops, являющийся набором *виртуальных функций*, определяющих независимый от типа интерфейс к классу сегмента. Каждый специфический порожденный класс (то есть каждый тип сегмента) должен иметь собственные реализации операций этого вектора. Структура seg содержит указатель (s_data) на зависимую от типа структуру данных, в которой хранятся закрытые данные сегмента. Эта структура не видна остальной части ядра и используется только зависимыми от конкретного типа сегмента функциями, реализующими операции над ним.

Вектор seg_ops содержит следующие операции:

- dup — дублирует образ;
- fault и faulta — используются при обработке страничных ошибок, возникающих в данном сегменте;
- setprot и checkprot — устанавливает или проверяет атрибуты защиты сегмента;

- unmap — отсоединяет сегмент и освобождает все его ресурсы;
- swapout — вызывается демоном swapper для выгрузки сегмента из памяти;
- sync — используется для сброса всех страниц в соответствующий сегменту объект данных.

Каждый сегмент также обладает процедурой создания create. Несмотря на то, что такая процедура является зависимой от определенного типа сегмента, доступ в ней не производится через вектор seg_ops по причине необходимости ее вызова до создания сегмента и, следовательно, до инициализации вектора seg_ops. Ядро имеет сведения об именах и синтаксисе вызова всех процедур создания сегмента (входные аргументы процедур могут отличаться друг от друга для разных типов сегментов) и само вызывает нужную из них при необходимости.

Следует упомянуть о разнице между виртуальной функцией и ее специфическими версиями для определенных порожденных классов. Например, функция faulta является виртуальной и определяет одну из общих операций над сегментом, но не существует какой-либо функции ядра под названием faulta(). Каждый класс-наследник (или тип сегментов) обладает собственной реализацией этой функции. Например, для сегмента типа seg_vn используется функция segvn_faulta().

14.4.4. Анонимные страницы

Анонимной страницей называется страница, не имеющая постоянного адреса хранения. Такие страницы создаются при первом изменении процессом страниц отображением их на объект типа MAP_PRIVATE. Система VM должна произвести создание закрытой копии такой страницы для процесса с тем, чтобы изменения не смогли повлиять на лежащий в ее основе объект. Все последующие обращения к странице должны переадресовываться к ее копии (не затрагивая при этом оригинальную страницу). Анонимные страницы могут уничтожаться при завершении работы процесса или при отказе от отображения. В то же время они могут при необходимости быть скопированы в область свопинга.

Анонимные страницы используются всеми сегментами, поддерживающими закрытые отображения. Например, инициализированные страницы данных, изначально указывающие на исполняемый файл, могут стать анонимными при первой попытке внесения изменений в них. За резервное копирование анонимных страниц отвечает *уровень свопинга.*

Существует еще одно понятие, связанное с анонимными страницами, — *анонимный объект.* В системе имеется единственный анонимный объект. Он представлен указателем vnode, равным NULL (в некоторых реализациях систем это файл /dev/zero) и является источником всех страниц, заполняемых нулями. Области неинициализированных данных и стека процесса являются ото-

бражением типа MAP_PRIVATE для анонимного объекта, в то время как совместно используемые области являются отображаемыми на него (MAP_SHARED).

При первом обращении к странице, имеющей ссылку на анонимный объект, такая страница становится анонимной независимо от того, является ли отображение закрытым или разделяемым. Это случается потому, что анонимный объект не обладает возможностью резервного копирования своих страниц, задача сброса их на устройство свопинга должна решаться ядром.

Анонимная страница представляется структурой anon, невидимой для остальных компонентов системы VM и управляемой при помощи процедурного интерфейса. Так как анонимная страница может быть совместно используемой, структура anon содержит счетчик ссылок. Сегмент, обладающий анонимными страницами, имеет ссылку на структуры anon таких страниц. Если отображение является закрытым, в каждом сегменте хранится отдельная ссылка на анонимную страницу. Если отображение разделяемое, то сегменты используют один и тот же счетчик. О совместном использовании анонимных страниц мы расскажем чуть позже, в разделе 14.7.4.

Процедурный интерфейс к остальной части системы VM экспортируется на уровне анонимности и содержит следующие функции:

- anon_dup() — дублирует ссылки на набор анонимных страниц. Это действие увеличивает на единицу значение счетчика ссылок каждой структуры anon, входящей в набор;
- anon_free() — освобождает ссылки на набор анонимных страниц, уменьшая значение счетчика ссылок каждой структуры anon, принадлежащей набору. Если при этом значение счетчика становится равным нулю, страница удаляется и освобождается ее структура anon;
- anon_private() — создает закрытую копию страницы и ассоциирует с ней новую структуру anon;
- anon_zero() — создает страницу, заполненную нулями, и ассоциирует с ней новую структуру anon;
- anon_getpage() — разрешает исключительное состояние анонимной страницы, производя ее чтение из устройства свопинга (при необходимости).

14.4.5. Аппаратное преобразование адресов

В системе VM все аппаратно зависимые коды собраны в отдельный модуль, называемый *уровнем HAT* (hardware address translation, аппаратное преобразование адресов), доступ к которому осуществляется при помощи четко определенного процедурного интерфейса. Уровень HAT полностью отвечает за преобразование адресов. Он должен устанавливать и поддерживать отображения, необходимые для MMU, такие как для таблиц страниц и буферов пре-

образования. Он является интерфейсом между ядром системы и устройством MMU, скрывая внутри себя детали архитектуры памяти конкретной машины от остальной части ядра.

Основная структура уровня HAT называется hat. Она является частью структуры as каждого процесса. Несмотря на то, что такое определение относится к взаимодействию между адресным пространством и набором аппаратных отображений типа «один к одному», уровень HAT является невидимым для *уровня as* и других частей системы VM. Этот уровень доступен через процедурный интерфейс, содержащий три типа функций:

- операции над самим уровнем HAT, такие как:
  - hat_alloc() и hat_free() — применяются для выделения и освобождения структур hat;
  - hat_dup() — дублирует преобразования при выполнении fork;
  - hat_swapin() и hat_swapout() — используются для обновления и освобождения информации HAT при загрузке процесса из области свопинга или выгрузки в нее;
- операции над некоторым диапазоном страниц процесса. Если такие страницы разделяются между другими процессами, преобразования на стороне последних не изменятся вследствие проведения следующих операций:
  - hat_chgprot() — изменяет атрибуты защиты страниц;
  - hat_unload() — используется для выгрузки или сброса бита корректности таблиц преобразований и сброса соответствующих вхождений TLB;
  - hat_memload() и hat_devload() — загружает преобразование одной страницы. Последняя функция используется seg_dev для загрузки преобразований страниц устройств;
- операции над всеми картами преобразованиями данной страницы. Страница может совместно использоваться несколькими процессами, каждый из которых обладает собственным преобразованием для нее. Операции этого типа включают в себя:
  - hat_pageunload() — выгружает все преобразования страницы. Это приводит к таким операциям, как сброс бита корректности PTE и вхождения TLB этой страницы;
  - hat_pagesync() — обновляет биты *изменения* (modified) и *ссылки* (referenced) во всех имеющихся структурах преобразований страницы, при этом используются переменные структуры page.

Вся информация, обрабатываемая на уровне HAT, является избыточной. Она может быть удалена или перестроена на основе данных, доступных не аппаратно-независимом уровне. Интерфейс не знает, что эти данные поддерживаются на уровне HAT и как долго это продолжается. На уровне HAT мож-

но отменить все преобразования в любой момент времени: если возникает исключительное состояние по определенному адресу, аппаратно-независимый уровень просто запросит уровень HAT повторно осуществить преобразование. Конечно, перестройка информации HAT является весьма затратной процедурой, поэтому сам уровень HAT старается по возможности избежать ее.

Структура hat сильно зависит от конкретной аппаратной архитектуры. Она может содержать указатели на таблицы страниц и другую информацию. Для поддержки таких операций, как hat_pageunload(), уровень HAT должен уметь обрабатывать все преобразования страницы, в том числе относящиеся к другим процессам, совместно использующими страницу. Для этого на уровне HAT все преобразования разделяемой страницы собираются в единую последовательность в связанном списке. Указатель на такой список хранится в зависящем от HAT поле структуры page (см. рис. 14.6).

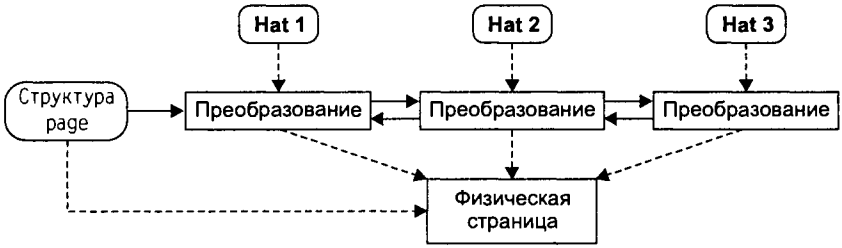


Рис. 14.6. Обнаружение всех преобразований одной страницы

Порт ссылок SVR4 поддерживается архитектурой Intel 80x86 [1]. Для нее на уровне HAT используется структура данных под названием *последовательность отображения* (mapping chunk), которая применяется для отслеживания всех образов физической страницы. Каждое активное вхождение таблицы страниц имеет соответствующее *вхождение последовательности отображения*. Так как неактивные преобразования не располагают такими вхождениями, размер последовательности меньше, чем таблицы страниц. Каждая физическая страница имеет связанный список вхождений последовательности отображения, по одному на каждое активное преобразование страницы. Указатель на этот список хранится в структуре page.

14.5. Драйверы сегментов

Существует несколько типов сегментов. Набор процедур и закрытых данных, реализующих каждый типа сегмента, называется *драйвером сегмента*. Изначальная версия VM поддерживает четыре типа сегментов:

- seg_vn. Отображают обычные файлы и анонимные объекты;
- seg_map. Внутренние отображения ядра на обычные файлы;

- seg_dev. Отображения на символьные устройства (буферы фреймов и т. д.);
- seg_kmem. Отображения, необходимые ядру системы.

В более поздних вариантах системы был добавлен драйвер seg_u для отображения области u, а также seg_objs для отображения объектов ядра в пространстве процесса. Некоторые специфические реализации включают в себя другие драйверы, например, seg_kp для многонитевых систем. Наиболее часто используемыми драйверами являются seg_vn и seg_map.

Драйвер сегмента должен иметь все функции, определенные в интерфейсе сегментов (см. раздел 14.4.3). Разработчик драйвера может объединять соседние сегменты одинакового типа или разбивать сегмент на несколько более мелких сегментов, если это повышает эффективность работы. Более того, хотя сегмент, как правило, обладает одинаковыми атрибутами защиты всех своих страниц, драйверы сегментов позволяют изменять защиту каждой страницы отдельно.

14.5.1. Драйвер seg_vn

Сегмент vnode (также известный под именем seg_vn) связывает адреса прикладных процессов с обычными файлами и анонимным объектом. Последний представлен указателем vnode, равным нулю (или файлом /dev/zero), и отображает области, заполненные нулями, такие как неинициализированные данные (bss) и пользовательский стек. Для заполнения таких страниц нулями используется исключительное состояние. Области текста и инициализируемых данных связываются с исполняемым файлом при помощи драйвера seg_vn. Дополнительные сегменты типа seg_vn могут быть созданы для обработки разделяемой памяти и файлов, принудительно отображенных системным вызовом mmap.

На рис. 14.7 показаны структуры данных, ассоциированные с сегментами vnode. Каждый такой сегмент поддерживает закрытую структуру данных для хранения дополнительной информации, используемой драйвером. Структура содержит следующие данные:

- установки *текущей* и *максимальной степени* защиты страниц сегмента;
- *тип* отображения (разделяемое или закрытое);
- указатель на vnode отображаемого файла. Дает возможность доступа ко всем операциям vnode над файлом [6];
- *смещение* от начала сегмента файла;
- указатель на *анонимную карту отображения* (для измененных страниц закрытых отображений, см. раздел 14.7.2);
- указатель на *массив элементов защиты страниц*, если страницы сегмента имеют неодинаковые атрибуты защиты.

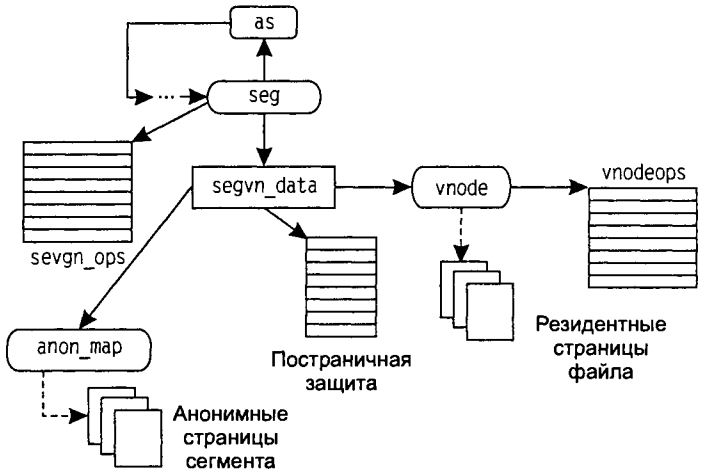


Рис. 14.7. Структуры данных, ассоциируемые с сегментом vnode

Максимальный уровень защиты устанавливается обычно при первичной инициализации отображения, в зависимости от потребности в безопасности и режима открытия файла. Например, если файл открыт в режиме «только для чтения», он не может отображаться как MAP_PRIVATE с атрибутом PROT_WRITE, даже если пользователь обладает правом на запись в этот файл. *Текущие установки защиты*, как правило, такие же, что и в случае максимально упроченной защиты. Позже они могут быть изменены при помощи вызова mprotect, но при этом уровень защиты не может стать выше.

Вызов mprotect может работать с любым диапазоном страниц, следовательно, некоторые страницы сегмента в состоянии иметь установки защиты, отличающиеся от установок сегмента. Информация об этом хранится в массиве элементов установок защиты. Он также позволяет производить блокировку отдельных страниц в памяти (через системный вызов mlock). Ответственность за соблюдение требований по поддержанию уровней защиты, не превосходящих наивысший, возлагается на ядро системы.

Возможность постраничной установки атрибутов защиты не является уникальной для seg_vn. Этот механизм разрешен к использованию сегментами любого типа, если они поддерживают вызовы mmap и mprotect.

14.5.2. Драйвер seg_map

В UNIX поддерживаются три вида доступа к файлам: загрузка страниц исполняемых файлов в память по необходимости, прямой доступ к отображаемым (nmap) файлам или применение вызовов read и write по отношению к открытым файлам. Первые два метода сходны друг с другом, так как после загрузки в память области текстов и данных исполняемого файла последующий доступ к ним мало чем отличается от обращений к отображаемым (nmap)

файлам. В обоих случаях для доступа к данным ядро использует подсистему памяти и исключительные состояния.

Проведение границы между вызовами read и write и доступом к образам (созданным при помощи mmap) может стать причиной некорректных ситуаций. В традиционных системах вызов read читает данные с диска в буферный кэш и из него в адресное пространство процесса. Если другой процесс вызовет mmap по отношению к тому же файлу, в памяти машины будут существовать две копии данных: одна — в буферном кэше, а вторая будет являться физической страницей, отображенной в адресном пространстве второго процесса. Если оба процесса произведут изменения в файле, ожидать можно всего.

Для предупреждения проблемы в системе VM произведена унификация всех трех методов доступа. Если процесс производит вызов read по отношению к открытому файлу, ядро сначала отобразит необходимые страницы файла в их собственные виртуальные адресные пространства при помощи драйвера seg_map, и только затем скопирует данные в адресное пространство процесса. Драйвер seg_map управляет вверенным ему адресным виртуальным пространством как кэшем; таким образом, в памяти находятся только те страницы, к которым обращения происходили относительно недавно. Тем самым система VM может играть роль буферного кэша, который становится излишним. Использование драйвера seg_map также позволяет полностью синхронизировать все виды доступа к файлам.

В системе имеется единственный драйвер seg_map. Он относится к ядру и создается при инициализации системы. Драйвер поддерживает две дополнительные функции: segmap_getmap() — для отображения части vnode в виртуальное адресное пространство и segmap_release() — для сброса такого отображения и записи данных обратно на диск (если они были изменены). Задача этих функций схожа с традиционными функциями буферного кэша bread() и brelse()/bwrite() и более подробно объясняется в разделе 14.8. Драйвер seg_map является оптимизированной для VM версией драйвера vnode и предоставляет возможность быстрого отображения файлов для ядра.

14.5.3. Драйвер seg_dev

Драйвер seg_dev отображает символьные устройства, реализующие интерфейс mmap. Обычно он применяется для работы с буферами фреймов, физической памятью, виртуальной памятью ядра и шинами памяти. Поддерживает только разделяемые отображения.

14.5.4. Драйвер seg_kmem

Этот драйвер отображает участки адресного пространства ядра, к которым относятся области текстов, данных bss и динамически выделяемой памяти ядра. Такие отображения не являются страничными, преобразование адресов

для них постоянно до тех пор, пока ядро не отменит отображение объекта (например, освободит динамически выделяемую память).

14.5.5. Драйвер seg_kp

Драйвер seg_kp отвечает за структуры нитей, стека ядра и легковесных процессов в многонитевых реализациях систем, таких как Solaris 2.x (см. раздел 3.6). Такие структуры могут находиться в областях памяти, поддерживающих или не поддерживающих свопинг. Драйвер seg_kp также выделяет *«красные»* зоны (red zones), чтобы обезопасить ядро от переполнения стека. «Красной» зоной называется защищенная от изменений страница, располагаемая в конце стека. Любая попытка записи в эту страницу приведет к исключительному состоянию, благодаря чему соседние страницы оберегаются от повреждения.

14.6. Уровень свопинга

Анонимный уровень (anon level) управляет анонимными страницами и обрабатывает исключительные состояния, вызванные ошибками доступа к принадлежащим ему страницам. На этом уровне должна поддерживаться информация, необходимая для локализации страницы. Если страница находится в памяти, в структуре anon появится указатель на ее структуру page. Если страница выгружается в область свопинга, ее необходимо считать с резервного носителя, управляемого уже уровнем свопинга. Взаимосвязи между структурой anon и страницами, выгруженными в область свопинга, управляются процедурой swap_xlate().

В системе может присутствовать несколько устройств свопинга. Каждое из них, как правило, является разделом на локальном диске, но не исключается использование удаленных дисков или даже файлов. Устройство свопинга может подключаться или отключаться динамически посредством системного вызова

```
swapctl (int cmd, void *arg);
```

где cmd принимает значения SC_ADD или SC_REMOVE[1], а arg является указателем на структуру swapres. Эта структура содержит полное имя к файлу свопинга (для локального раздела свопинга это может быть специальный файл устройства), а также местонахождение и размер области свопинга внутри этого файла.

Для каждого устройства свопинга ядро системы поддерживает структуру swapinfo, добавляемую в связанный список (см. рис 14.8). Оно также выделяет массив структур anon с количеством элементов, равным количеству страниц устройства. Структура swapinfo содержит указатель vnode и начальное

[1] Существует еще две команды системного вызова, используемые для административных целей, — SC_LIST и SC_GETNSWP.

смещение области свопинга, а также указатели на начало и конец массива anon. Свободные структуры anon в массиве взаимосвязаны. Указатель на такой связанный список хранится в структуре swapinfo. Изначально ее элементы свободны.

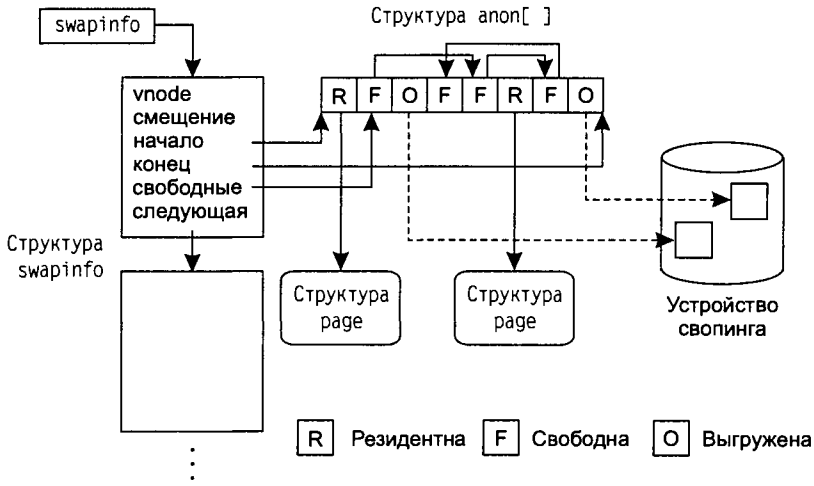


Рис. 14.8. Структуры данных уровня свопинга

Выделение и резервирование пространства свопинга производится для каждого сегмента в отдельности. При создании ядром сегмента, потенциально требующего пространство свопинга (обычно это закрытые отображения, доступные для записи), для этой цели сразу же резервируется необходимое пространство (обычно его объем равен размеру сегмента). Уровень свопинга отслеживает общее доступное пространство свопинга и резервирует необходимое число страниц из пула. При этом не происходит установки каких-то специфических страниц свопинга, задача состоит лишь в обеспечении наличия зарезервированного пространства, чтобы при необходимости им можно было воспользоваться.

Правила резервирования являются консервативными. Они требуют от процесса всегда резервировать область хранения для всей его анонимной памяти, даже если процесс никогда не воспользуется свопингом. Если система задействуется для выполнения больших приложений, то для их работы потребуются устройства свопинга больших объемов. С другой стороны, консервативный подход гарантирует, что ошибки из-за недостаточности памяти (например, при выполнении вызовов exec или mmap) обработаются синхронно. После настройки адресного пространства процессом ему всегда будет доступно необходимое пространство свопинга (до тех пор, пока не потребуется увеличить его объем).

Сегментам пространство свопинга предоставляется постранично, даже если создаются новые анонимные страницы. Пространство может выделяться

только в рамках ранее сделанного резервирования. Выделение свободной страницы свопинга осуществляется при помощи процедуры swap_alloc(), которая также ассоциирует ее с анонимной страницей через структуру anon. Процедура старается распределить нагрузку устройств свопинга, меняя устройство после размещения нескольких страниц.

В системе SVR4 расположение структуры anon в массиве anon соответствует позиции страницы свопинга в соответствующем устройстве свопинга. Процедура swap_alloc() возвращает указатель на структуру anon. Этот указатель служит *именем* анонимной страницы и может быть использован при локализации страницы в области свопинга.

Массив anon и его связи с устройством свопинга являются внутренней информацией уровня свопинга. Этот уровень работает с процедурным интерфейсом. Для выделения и освобождения структуры anon используются процедуры swap_alloc() и swap_free(); для получения vnode и смещения страницы свопинга, связанной со структурой anon, применяется процедура swap_xlate(). Затем для получения страницы из области свопинга выполняется операция VOP_GETGAGE.

Сегменты, использующие анонимную память, ссылаются на структуру anon косвенно: они поддерживают *массив ссылок*, содержащий указатели на такие структуры. Количество элементов массива равно количеству страниц в сегменте. Если страница не является анонимной (то есть обрабатывается драйвером сегмента), соответствующий его элемент имеет значение NULL. Это позволяет производить операции сразу над несколькими страницами, часть из которых могут быть анонимными.

Устройства свопинга могут быть добавлены или удалены динамически во время функционирования системы. Добавление устройства увеличивает максимально доступное пространство свопинга, удаление, наоборот, уменьшает этот объем. При удалении устройства все страницы, хранящиеся в нем, копируются на другое устройство. Для таких страниц должны быть выделены новые структуры anon. Операция удаления способна повлечь страничную ошибку, если на остальных устройствах свопинга недостаточно места для переноса страниц.

Однако при этом сегменты, имеющие страницы свопинга, продолжают обладать ссылками на устаревшие структуры anon. Не существует методики переназначения этих ссылок. Поэтому старые структуры anon содержат поле, в котором хранится указатель на новую структуру, действительно отображающую страницу. В структуре anon также находится счетчик ссылок, а также указатель на структуру page (если страница резидентна) или на следующий элемент (если структура anon свободна).

14.7. Операции системы VM

Мы рассмотрели структуры данных и интерфейсы, являющиеся наиважнейшими понятиями системы VM. Остановимся теперь на описании взаимодействия этих компонентов, которое и является средством управления памятью.

14.7.1. Создание нового отображения

Новые области отображаются в адресном пространстве либо при выполнении exec, либо процессом mmap. Системные вызовы exec и mmap находят vnode отображаемого файла и загружают определенную для него функцию VOP_MAP[1]. Эта функция производит проверку специфических аргументов файловой системы. Затем она проверяет, не осуществляет ли процесс других отображений в заданном диапазоне адресов, если это так, то для отмены отображения вызывается as_unmap(). Последним шагом является вызов as_map() для организации отображения файла в адресное пространство. Операция as_map() размещает структуру seg и вызывает соответствующую процедуру create для инициализации сегмента.

Системный вызов mmap проверяет права доступа, которые не должны превосходить уровень атрибутов доступа, установленных при открытии файла. Это означает, что процесс не может производить изменения файла через разделяемое отображение, если файл был открыт в режиме «только для чтения». Наивысшие права доступа записываются в структуре seg. Функции, их изменяющие, должны проверять соответствующее поле.

Системный вызов exec создает закрытые отображения области текстов, данных и стека[2]. Они соединяют тексты программ и инициализируемые данные с исполняемым файлом, данные bss и стек — с анонимным объектом. Вызов exec может также определить дополнительные отображения для областей совместно используемых библиотек. Текстовые области отображаются с атрибутами доступа PROT_READ и PROT_EXECUTE, таким образом попытка записи в такие страницы приведет к возникновению исключения. Области данных доступны для записи, но при этом еще не измененные страницы инициализированных данных могут совместно использоваться всеми процессами, выполняющими программу.

Функции as_unmap() и as_free() удаляют отображения. Функция as_unmap(), вызываемая процедурой munmap, освобождает диапазоны адресов, которые могут содержать один и более целых сегментов или их фрагментов. Функция as_free(), вызываемая exit, освобождает адресное пространство целиком. Обе операции выполняют обращение к сегментам в порядке очередности и вызов процедур сегментного уровня, производящих удаление отображения страниц.

14.7.2. Обработка анонимных страниц

Анонимные страницы могут быть созданы в двух случаях:

- при осуществлении первой записи на страницу, имеющую атрибут MAP_PRIVATE закрытого отображения на файл или анонимный объект.

---

[1] Для поддержки системы VM в вектор vnodeops были добавлены операции VOP_MAP, VOP_GETPAGE и VOP_PUTPAGE.

[2] Для файлов в формате ELF вызов exec отображает интерпретатор, указанный в заголовке программы. В свою очередь, интерпретатор отображает необходимую программу.

Такие страницы могут содержать тексты, данные, стек и другие принудительно созданные закрытые отображения;

- при первом обращении в разделяемой странице. Этот случай более подробно обсуждается в разделе 14.7.6.

Если создается закрытое отображение, аппаратные средства преобразования адреса объявляют такие страницы доступными лишь для чтения. Это происходит потому, что внесение изменений в такие страницы может привести к изменению лежащего в их основе объекта. Следовательно, ядру системы необходимо перехватить первую попытку записи в такую страницу и отключить отображение на файл. Будет также удалена и информация о резервном копировании страницы, поэтому ядру нужно выделить для нее место на устройстве свопинга. На этой стадии страница становится анонимной.

На рис. 14.9. показана реализация анонимных страниц сегментами vnode. Закрытая структура данных сегмента содержит указатель на структуру anon_map, которая, в свою очередь, хранит указатель на *массив ссылок на структуры anon* и данные о его размере. Этот массив имеет количество элементов, равное количеству страниц сегмента. Каждый элемент является ссылкой на структуру anon соответствующей страницы или обладает значением NULL в том случае, если страница не анонимна. Структура anon содержит сведения о местонахождении страницы в физической памяти или на устройстве свопинга, в ней также хранится счетчик ссылок на страницу. Если значение счетчика становится равным нулю, страница и ее структура anon может быть удалена.

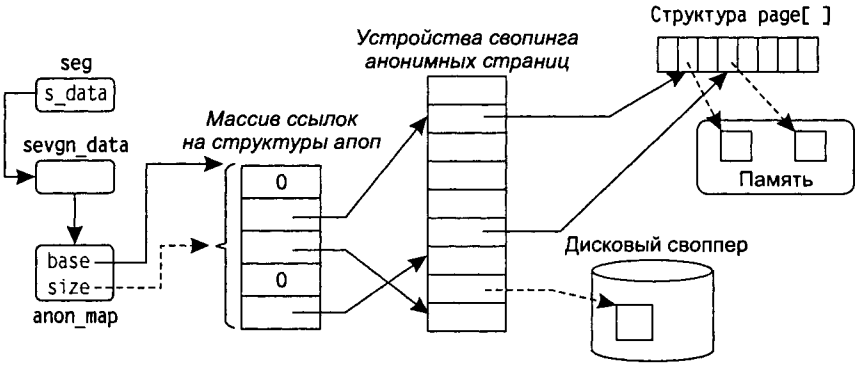


Рис. 14.9. Анонимные страницы сегмента vnode

Описанные структуры данных не создаются одновременно с выделением сегмента. Чаще всего они формируются при необходимости, то есть при первой попытке записи в страницу сегмента. Данный подход имеет явные преимущества, поскольку во многих сегментах vnode никогда не возникает такой потребности в анонимных страницах (например, тексты программ обычно изменяются только во время отладки, при установке отладчиком контрольной точки).

Первая попытка записи в закрытую отображаемую страницу становится причиной возникновения исключения (ошибки защиты). Обработчик умеет распознавать причину возникновения исключительной ситуации по отношению к отображениям, имеющим тип MAP_PRIVATE, и установок защиты сегментов, не равных «только для чтения» (в противоположность защите аппаратного преобразования адресов страницы, как правило, доступной «только для чтения», что вызывает ошибку при попытке записи). Обработчик размещает структуру anon, предоставляя тем самым пространство свопинга (так как каждая структура anon указывает на уникальную страницу на устройстве свопинга). Он также создает ссылку на эту структуру (более подробно читайте о ссылках на структуры в разделе 7.9) и сохраняет ее в соответствующем элементе в массиве ссылок на структуры anon.

Затем обработчик создает новую копию страницы, используя выделенную по такому случаю новую физическую страницу. Обработчик сохраняет указатель на структуру page страницы в структуре anon. На последней стадии управление передается уровню HAT для установки преобразования новой копии страницы (доступной для записи). Все последующие модификации страницы будут происходить в только что созданной ее закрытой копии.

Иногда необходимо производить дополнительные действия. Если создаваемая страница является первой анонимной страницей сегмента, обработчик выделяет и инициализирует структуру anon_map и массив ссылок на структуры anon. Если страница, вызвавшая исключение, не находится в памяти, обработчик производит ее чтение с устройства резервного хранения.

Ядро системы может периодически перемещать страницы в устройство свопинга. Однако процесс в состоянии повторно повлечь исключительное состояние из-за той же страницы. Тогда обработчик ошибок находит, что сегмент уже имеет ссылку на структуру anon для такой страницы и использует эту ссылку для получения страницы из устройства свопинга.

При вызове fork процессом все его анонимные страницы становятся совместно используемыми с его потомком (на условиях копирования при записи). Предок и потомок обладают собственными структурами anon_map и массивами ссылок на структуры anon, которые содержат ссылки на одни и те же страницы. Более подробно о создании процессов читайте в следующем разделе.

14.7.3. Создание процесса

После выделения и инициализации структуры proc системный вызов fork вызывает функцию as_dup() для дублирования адресного пространства родительского процесса. Эта функция, в свою очередь, вызывает as_alloc() с целью размещения новой структуры as для потомка. Затем она просматривает спи-

сок сегментов и вызывает для каждого из них операцию *дублирования* (зависящую от конкретного драйвера сегмента).

Операция дублирования сегмента начинается с размещения новой структуры seg, а также структуры для содержания закрытых данных, зависящих от типа создаваемого сегмента. Многие поля структур, такие как базовый адрес, размер, указатель vnode, смещение и атрибуты защиты, копируются от родительского процесса. Отображения текстов, данных и стека обладают атрибутом MAP_PRIVATE как для родительского процесса, так и для предка. Любые отображения с атрибутом MAP_SHARING процесса-предка остаются такими же и для его потомка, чтобы сохранить семантику принятых в UNIX правил наследования разделяемых областей памяти. Другие поля, такие как указатель на as, имеют значения, актуальные для потомка.

Для закрытых отображений структура anon_map должна быть продублирована так, чтобы позволить совместное использование анонимных страниц на основе технологии копирования при записи. Первой ступенью реализации этой задачи является вызов hat_chprot() для установки защиты от записи всех анонимных страниц. Это гарантирует возникновение исключения (страничной ошибки) при любой попытке изменения страниц, осуществляемой как родительским процессом, так и его потомком. Обработчик создаст новую копию страницы и разрешит к ней доступ в режиме записи.

После завершения работы hat_chprot() драйвер сегмента вызовет функцию anon_dup() для дублирования структуры anon_map. Функция формирует новую структуру anon_map и массив ссылок anon. Она *клонирует* все ссылки массива путем копирования указателей и увеличения счетчиков ссылок соответствующих структур anon. На рис. 14.10 показано состояние процессов после дублирования сегментов.

После завершения создания дубликатов всех сегментов функция as_dup() вызовет hat_dup() для копирования структуры hat и информации о преобразовании адресов страниц. Функция hat_dup() может выделять для потомка новые таблицы страниц и инициализировать их путем наследования данных от родительского процесса.

Как мы видим, вызов fork производит дублирование адресного пространства без непосредственного копирования самих страниц. Он работает только с отображениями и таблицами страниц. Несмотря на то, что системный вызов vfork производит сходные действия, разработчики системы SVR4 сохранили его по нескольким причинам. Во-первых, вызов vfork работает быстрее даже по сравнению с применением копирования при записи, так как не клонирует отображения. Во-вторых, остаются программы, зависящие от vfork, которые позволяют предкам изменять адресное пространство их потомков. Запрет на vfork приведет к невозможности работы таких программ. По этой причине, хотя системный вызов vfork и не входит в оригинальную версию System V UNIX, разработчики добавили его в SVR4 вместе с архитектурой VM.

14.7.4. Совместное использование анонимных страниц

Функция anon_dup(), вызываемая во время выполнения fork, производит дублирование ссылок на все анонимные страницы родительского процесса. После этого страницы могут совместно использоваться предком и потомком на основе технологии копирования при записи. Несмотря на то, что операция anon_dup() имеет дело с сегментами целиком, совместное использование является постраничным.

На рис. 14.10 показано состояние сегмента родителя и потомка, а также соответствующих структур anon после завершения выполнения anon_dup (). На рис. 14.11 представлено состояние тех же объектов через некоторое время. За этот период процесс-предок внес изменения в страницы 0 и 1 сегмента. Страница 0 изначально не являлась анонимной. Однако при модификации ее потомком ядро произвело выделение новой структуры anon и физической страницы, а также добавило ссылку в массив ссылок на структуры anon потомка. Так как родительский процесс не производил изменения страницы, она по-прежнему остается отображенной на объект vnode в его адресном пространстве.

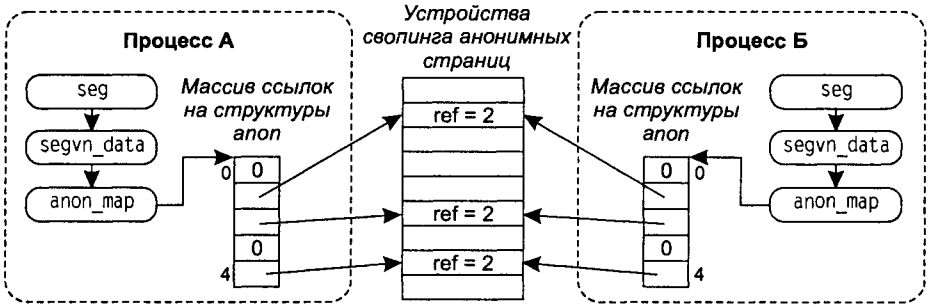


Рис. 14.10. Совместное использование анонимных страниц (стадия 1)

Страница 1 изначально же являлась анонимной и разделялась между родительским и дочерним процессом. Значение счетчика ссылок этой страницы равнялось 2. Так как отображение было закрытым, все изменения, внесенные потомком, не влияли на общую копию. Ядро системы создало новую копию страницы и разместило для нее структуру anon. Процесс-потомок сбросил ссылку на оригинальную структуру anon и получил ссылку на новую. В результате родитель и потомок стали обладать различающимися друг от друга ссылками на структуры anon страницы. Значение счетчика каждой структуры anon стало равным 1.

Из приведенного примера видно, что совместное использование страниц имеет постраничную основу. На рис. 14.11 показано состояние, в котором оба процесса разделяют между собой анонимные страницы для страниц 2 и 4 сегмента, но не имеют копий для страниц 0, 1 и 3.

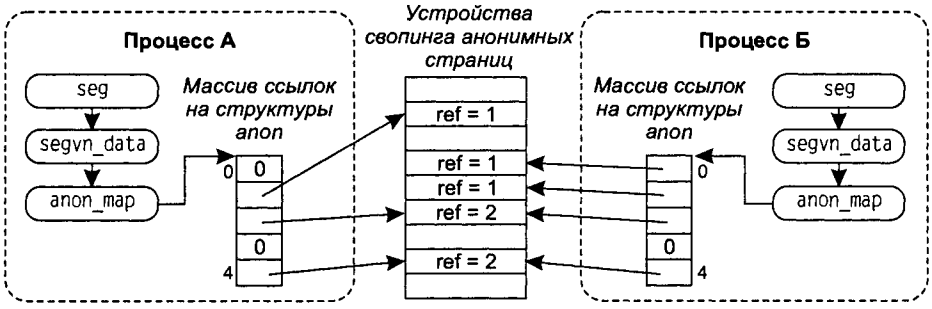


Рис. 14.11. Совместное использование анонимных страниц (стадия 2)

14.7.5. Обработка страничных ошибок

Для обработки исключений любого типа (ошибок корректности или защиты) вызывается одна и та же процедура trap(). Она определяет тип исключения, тип доступа (чтение/запись), находит структуру as процесса, послужившего причиной ошибки, и вызывает as_fault() для обработки исключения. Процедура as_fault() выявляет сегмент, содержащий обрабатываемый адрес, и вызывает нужную операцию, зависящую от типа драйвера. Для выполнения задачи уровень as сортирует сегменты в порядке увеличения базовых адресов, а также поддерживает указатель на сегмент, внутри которого произошла последняя страничная ошибка. Этот указатель используется как рекомендация для выбора места начала поиска: принцип *локальности ссылок* говорит о том, что следующая ошибка, скорее всего, будет иметь место в том же сегменте.

Обработка ошибки зависит от типа сегмента. В этом разделе мы остановимся только на наиболее часто встречающихся ошибках в сегментах vnode. Обработчик исключений драйвера seg_vn имеет имя segvn_fault(). Он преобразует сбойный адрес и логический номер страницы сегмента, который в дальнейшем будет использован как индекс в anon_map и массив элементов защиты страниц (если таковой существует).

Ошибки защиты происходят из-за установок защиты в аппаратном преобразователе адресов и могут быть как фактическими (real), так и ложными (spurious). Процедура segvn_fault() отличает тип ошибки путем проверки информации в закрытых данных сегмента. Узнать о фактической защите страницы можно по ее вхождению в массиве защиты страниц. Если уровень защиты равняется NULL, то данные о ней можно почерпнуть из информации о защите сегмента. Если эти данные запрещают доступ к странице, процедура segvn_fault() уведомляет об этом процесс путем отправки сигнала SIGSEGV.

Мнимые ошибки защиты возникают в том случае, если аппаратура умышленно запрещает защиту в преобразовании аппаратного адреса для этой страницы с целью реализации копирования при записи или имитации бита ссылки (см. раздел 13.5.3). Дальнейшие действия процедуры segvn_fault() по

обработке таких ошибок (а также всех ошибок корректности) зависят от состояния страницы:

- если существует элемент anon_map для страницы, процедура вызывает anon_getpage() для получения анонимной страницы. Операция anon_getpage() может найти нужную страницу в памяти (в этом случае структура anon будет указывать на нее), либо прочесть ее содержимое из устройства свопинга;
- когда для страницы не существует элемента anon_map, а сегмент отображает файл, процедура вызывает операцию VOP_PAGE по отношению к vnode файла. Операция VOP_GETPAGE может найти нужную страницу в памяти, либо считать ее содержимое с диска;
- если для страницы не существует элемента anon_map, а сегмент отображает анонимный объект, процедура вызывает anon_zero() для выделения страницы, заполненной нулями.

Большинство специфических случаев обрабатываются операциями VOP_GETPAGE и anon_getpage(). Если страница находится в списке свободных страниц, она должна быть перенесена из этого списка. Когда страница помечена как некорректная с целью имитации бита ссылки, обработка исключения состоит в обратной установке бита корректности. Если страница находится в стадии передачи (in transit), вызывающий процесс должен дождаться завершения операции ввода-вывода. Текстовые страницы, оставшиеся в памяти после предыдущих запусков программ, могут быть получены путем проведения поиска в таблице хэширования на основе пары <vnode, offset> (<vnode, смещение>). Эти процедуры также обрабатывают все случаи кластеризации или упреждающего чтения.

Мы подошли к той стадии выполнения, когда страница уже находится в памяти, а процедура segvn_fault() располагает указателем на нее. Рассмотрим случай применения копирования при записи. Здесь имеются несколько вариантов.

1. Предпринимается попытка записи по отношению к закрытому отображению.
2. Атрибуты защиты сегмента (или страницы, что в данном случае не важно) разрешают запись, но дальнейшее поведение зависит от следующего:
   1) страница не имеет структуры anon;
   2) структура anon страницы содержит более чем одну ссылку.

Случай 2.1 приводит к созданию анонимного отображения файла, все изменения должны быть ограничены закрытой копией страницы. В ситуации 2.2 имеет место размножение анонимной страницы (на условиях копирования при записи), что может происходить после завершения выполнения fork. В обоих вариантах обработчик исключения производит вызов anon_private() для создания закрытой копии страницы.

Последним этапом работы обработчика исключений является вызов hat_memload() для загрузки нового преобразования страницы в структуры аппаратного преобразования адресов (в таблицы страниц и элементы TLB).

14.7.6. Разделяемая память

В ОС System V механизм разделяемой памяти является частью средства взаимодействия процессов (IPC, см. раздел 6.3.4). После создания процессом разделяемой области памяти этот процесс совместно с другими взаимодействующими процессами может произвести присоединение этого участка в свое адресное пространство. Каждый процесс имеет право расположить такие области по различным адресам, а также присоединять один и тот же участок сразу на несколько адресов в своем пространстве. После этого процесс может выполнять чтение или запись в область, используя обычные инструкции доступа к памяти. Все изменения, внесенные в разделяемую область памяти одним процессом, сразу же становятся видны остальным взаимодействующим процессам. Область памяти продолжает существовать до тех пор, пока не будет удалена принудительно (даже в том случае, если к ней не подключено ни одного процесса). Это позволяет процессу создать область разделяемой памяти, записать в нее данные и завершить работу. Через некоторое время другой процесс может подключиться к области в той же точке и запросить оставленные в памяти данные.

В этом разделе мы рассмотрим реализацию технологии разделяемой памяти в архитектуре VM. Каждая область разделяемой памяти представляется при помощи сегмента vnode, отображающего анонимный объект с атрибутом MAP_SHARED. Так как отображение является обобществленным, все его изменения влияют непосредственно на единую разделяемую копию данных и, следовательно, сразу же окажутся видимыми всеми процессами. Однако анонимный объект не предлагает резервного копирования таких страниц. Следовательно, при первой записи страницы должно происходить ее преобразование в анонимную страницу и копирование на устройство свопинга.

Форма совместного использования страниц памяти существенно отличается от разделения анонимных страниц между родителем и его предком на основе технологии копирования при записи. В рассматриваемом варианте оба процесса обладают отдельными ссылками на структуры anon, а разделение происходит на уровне индивидуальных страниц.

На рис. 14.12 показана реализация разделяемой памяти. Область, представленная единственной структурой anon_map, разделяется между всеми процессами, подключившими эту область в свое адресное пространство. Каждый процесс поддерживает собственную структуру seg для этой области, в которой хранится базовый адрес, атрибуты защиты и другая информация. Структура anon_map имеет счетчик ссылок, каждый процесс обладает ссылкой на нее. Еще одну ссылку на структуру данных, ассоциированную с областью, поддерживает подсистема IPC.

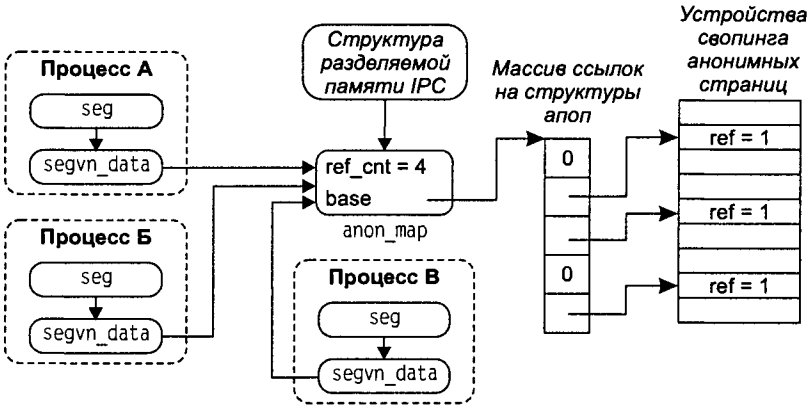


Рис. 14.12. Использование разделяемой памяти

Ссылки IPC создаются при выделении разделяемой области памяти. Их освобождение происходит при принудительном удалении области (командой IPC_RMID системного вызова shmctl). Такой подход дает гарантию, что область памяти не исчезнет автоматически при возникновении ситуации, когда к ней не подключен ни один процесс.

Структура anon_map содержит ссылки на отдельные анонимные страницы. Если значение счетчика ссылок достигает нуля (то есть все процессы произвели отключение области из своих адресных пространств), ядро сначала производит освобождение всех анонимных страниц и затем удаляет массив ссылок на структуры anon и anon_map.

14.7.7. Другие компоненты

Пул свободных страниц

Интерпретация пула свободных страниц схожа с принятой в операционной системе BSD. Специальный процесс pagedaemon реализует алгоритм «часов с двумя стрелками» (см. раздел 13.5.3). Такой же подход был адаптирован в архитектуре VM. Однако, говоря о свободных страницах, необходимо упомянуть следующие аспекты:

- при первом проходе вызывается hat_pagesync() для отключения ссылки и изменения битов в структурах hat;
- все «грязные» страницы сбрасываются на диск операцией VOP_OPERATE, обрабатывающей все варианты кластеризации;
- для сброса бита корректности страниц, перемещаемых в список свободных страниц, вызывается процедура hat_pageunload();
- при необходимости может производиться имитация бита ссылки.

Свопинг

При загрузке UNIX коды инициализации системы создают процесс swapper, который является системным процессом, не имеющим прикладного контекста. Идентификатор PID этого процесса равен нулю. Процесс swapper выполняет процедуру sched(), которая работает непрерывно (до завершения работы системы или ее краха) и, как правило, он находится в режиме сна. Система будит процесс каждую секунду, либо при возникновении некоторых событий.

Процесс swapper производит проверку количества свободной памяти и на основе полученного значения принимает решение о дальнейших действиях. Если количество свободной памяти меньше, чем величина (настраиваемого) параметра t_gpgslo, демон swapper выгружает процесс из памяти. Для выбора такого процесса процесс вызывает для каждого класса зависимую от его приоритета операцию CL_SWAPOUT, которая возвращает наиболее подходящего кандидата в этом классе (см. раздел 5.5). Процедура класса CL_SWAPIN осуществляет обратную операцию, то есть выбирает процесс, возвращаемый в основную память при увеличении ее свободного размера.

Задача swapper выгружает процесс с помощью вызова функции as_swapout(), которая один за другим просматривает каждый сегмент и вызывает специфическую применительно к нему операцию *выгрузки*. На этом этапе работы драйверу сегмента необходимо сохранить данные всех страниц сегмента, находящихся в памяти, на резервный носитель. Большинство сегментов, как правило, имеют тип seg_vn. Реализация операции выгрузки для сегментов такого типа возложена на операцию segvn_swapout(). Последним этапом работы процесса swapper является выгрузка области u процесса.

Для обратной загрузки процесса в память swapper восстанавливает в памяти только его область u. А уже при выполнении процесс вызовет исключительные состояния при обращении к страницам, что приведет к их загрузке в основную память.

14.8. Взаимодействие с подсистемой vnode

Файловая система обеспечивает резервное копирование большого количества сегментов VM. Из этого можно сделать вывод, что подсистема VM постоянно взаимодействует с файловой системой, производя обмен данных между файлами и памятью. Файловая система в реализациях вызовов read и write применяет отображения в памяти. В этом разделе будет описано взаимодействие VM и файловой системы. Здесь также будут затронуты некоторые интересные проблемы, имевшие место при реализации этого взаимодействия.

14.8.1. Изменения в интерфейсе vnode

В таких ОС, как 4.3BSD, структуры данных управления памятью содержат информацию о физическом размещении файловых блоков (располагающих-

ся в элементах заполняемой по мере необходимости таблицы страниц). Основным недостатком такого подхода является обособление друг от друга файловой подсистемы и подсистемы VM. В архитектуре VM все специфические детали файловой системы перенесены на уровень vnode, обращение к файлам осуществляется через процедурный интерфейс vnode. Для встраивания такой возможности в системе SVR4 появились три дополнительные операции интерфейса vnode: VOP_MAP, VOP_GETPAGE и VOP_PUTPAGE.

Операция VOP_MAP вызывается из mmap и выполняет зависимые от файловой системы действия по инициализации и проверке параметров. Эта операция контролирует, например, отсутствие блокировки отображаемого файла или невозможность дальнейшего отображения при достижении конца файла. Файловая система также может читать информацию, необходимую для преобразования смещений файла в физические номера блоков (из индексных дескрипторов и блоков косвенной связи). Причины, в силу которых это делается, описаны ниже.

Операция VOP_GETPAGE вызывается подсистемой VM при необходимости получения страниц из файла. Страницы могут при этом уже находиться в памяти, но система VM не обладает данными об их преобразованиях. Операция VOP_GETPAGE начинает свою работу с проверки наличия страниц в памяти. Ядро хранит информацию обо всех страницах ядра в глобальной таблице хэширования, индексом к которой является пара <vnode, offset>. Ядро помещает данные о странице в таблицу при размещении ее в памяти и удаляет при сбросе бита корректности или переназначении страницы. Несмотря на то, что каждая файловая система реализует операцию VOP_PAGE по-своему, все они применяют единую функцию поиска в хэш-таблице. Если страница там не обнаруживается, операция произведет ее чтение из файла (посредством специфичных для конкретной файловой системы функций). В локальных файловых системах, например ufs, следует определять местонахождение данных на диске на основе из индексного дескриптора и блоков косвенной связи файла, в то время как в удаленных файловых системах (например, NFS) необходимо производить запрос на получение страницы от сервера.

Операция VOP_GETPAGE обрабатывает любые случаи обращения к файлам, именно поэтому возможна оптимизация ее работы при помощи таких технологий, как упреждающее чтение. Для той или иной файловой системы, как правило, известны данные об оптимальном размере передачи, а также специфика логического разбиения дисков, поэтому операция может быть оптимизирована за счет *кластеризации* (klustering[1]). Этот термин обозначает процесс объединения смежных страниц в рамках операции ввода-

[1] Обратите внимание на этот термин, который не следует путать с *clustering*, обозначающим процесс логической группировки смежных физических страниц в памяти.

вывода[1]. Операция VOP_GETPAGE также может производить операции vnode, такие как обновление данных об обращениях или количестве изменений соответствующего индексного дескриптора.

Операция VOP_PUTPAGE вызывается для сброса страниц, потенциально заполненных полезной информацией (так называемых «грязных» страниц) обратно в файл. Параметры операции включают в себя флаг, указывающий на синхронность или асинхронность записи. Если операция вызывается демоном pagedaemon с целью освобождения некоторого количества памяти, может произойти взаимная блокировка. Например, операции VOP_PUTPAGE необходимо производить определение физического расположения страниц на диске. Для этого иногда приходится читать данные из блока косвенной связи (см. раздел 9.2.2), что невозможно из-за отсутствия свободной памяти.

Одним из способов предупреждения возникновения взаимной блокировки является хранение информации о преобразованиях из блоков косвенной связи в памяти в течение всего периода поддержки отображения файла. Такой подход увеличивает производительность работы обеих операций (VOP_GETPAGE и VOP_PUTPAGE), так как не требует проведения чтения данных из блоков косвенной связи. С другой стороны, хранение в памяти информации об этих блоках приводит к некоторому избыточному расходованию оперативной памяти.

14.8.2. Унификация доступа к файлам

Подсистема памяти и файловая система тесно взаимоувязаны друг с другом. Файловая система отвечает за резервное хранение сегментов VM, в том время как подсистема VM реализует доступ к файлам (см. рис. 14.13). Если процесс вызывает read или write по отношению к файлу, ядро производит временное отображение его части в своем адресном пространстве, используя сегмент seg_map. После этого в системе возникает исключение для переноса данных в образ, и затем необходимая информация копируется в адресное пространство

[1] Термин klustering употребляется относительно редко, и несмотря на то, что в переводной литературе не делается различия между двумя понятиями (единый термин подразумевает выделение групп объектов с общими признаками), оно существует, но бытует путаница в семантике. Имеет смысл дополнить сказанное автором. В традиционной файловой системе UNIX кластеризацией (clustering) называется «сбор смежных измененных страниц и одновременная их запись в одном запросе ввода-вывода» (глава 16). В Sun-FFS под кластеризацией (clustering) понимается методика поддержания «высокой производительности системы при увеличении дробления операций ввода-вывода» (глава 11). Но в рамках модели VM UNIX определения таковы: логический cluster — набор смежных страниц, кратный размеру страницы устройства хранения (сектору или блоку в разных ОС), kluster — набор кластеров (cluster) в виртуальной памяти или на запоминающем устройстве (типичный размер 4-8 Кбайт при размере cluster'а в 1 Кбайт). Klustering имеет место *при чтении* с носителя и означает упреждающую дополнительную загрузку страниц полного kluster'а при *возникновении страничного исключения* для страницы cluster'а. Что и дает выигрыш в общей производительности. — *Прим. ред.*

процесса или из него. Драйвер seg_map считывает блоки файловой системы в страничную память, для этой цели уже не нужен буферный кэш. Такой подход расширяет семантику доступа к отображаемым файлам на традиционные методы доступа. Унифицированный подход к интерпретации файлов позволяет избавиться от проблем достоверности данных, которые могут возникать при одновременном обращении к одному и тому же файлу разными методами.

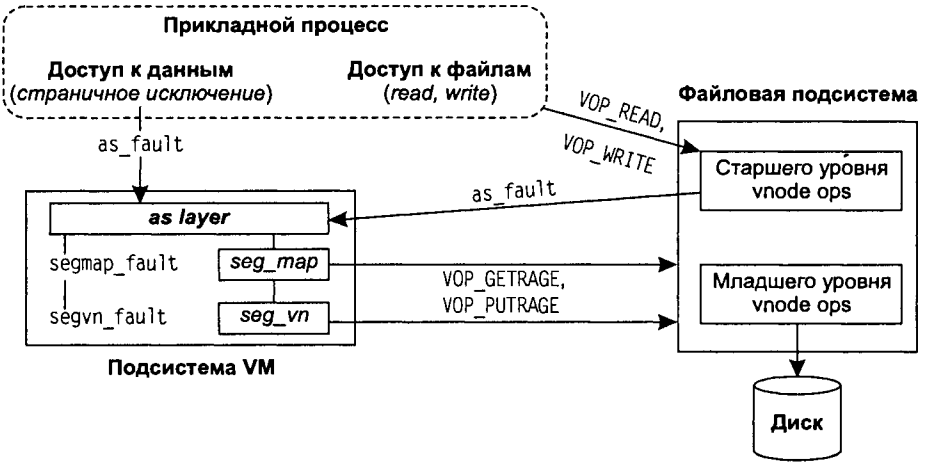


Рис. 14.13. Взаимосвязи между VM и файловой системой

Драйвер seg_map является упрощенной версией драйвера vnode и предоставляет возможность быстрого создания временного отображения файла в ядре. На рис. 14.14 показаны используемые при этой операции структуры. Закрытые данные сегмента (структура segmap_data) содержат указатели на массив элементов структур smap, а также на таблицу хэширования и список свободных элементов smap. Каждое вхождение smap содержит связь с таблицей хэширования и список свободных элементов, указатель на vnode и смещение представляемой им страницы, а также счетчик ссылок, отслеживающий количество процессов, обращающихся к элементу.

Каждый элемент smap описывает одну страницу сегмента. Виртуальный адрес страницы ядра определяется по следующей формуле:

```
addr = base + entrynum * MAXBSIZE;
```

где base — начальный адрес сегмента seg_map, entrynum — индекс smap в массиве, а MAXBSIZE — размер страницы (зависит от конкретной аппаратной архитектуры). На одной машине способны сосуществовать сразу несколько различных файловых систем, поэтому одна страница сегмента может указывать на один блок, несколько блоков или даже некоторую часть блока файловой системы.

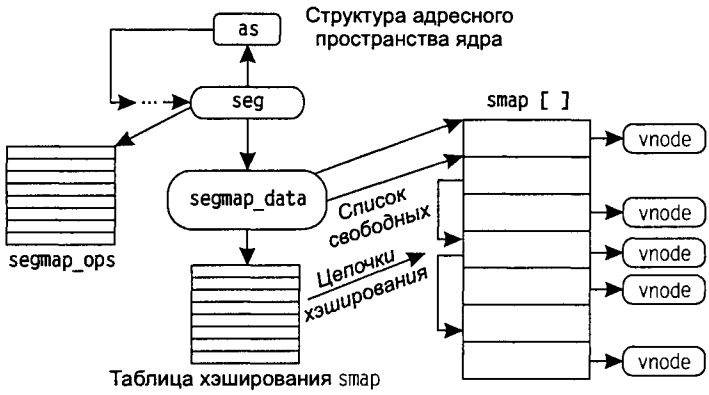


Рис. 14.14. Структура seg_map

При вызове процессом read файловая система определяет vnode и смещение для получения запрашиваемых данных и затем вызывает segmap_getmap() для создания образа страницы. Операция segmap_getmap() проверяет структуры хэширования на предмет существования отображения. В этом случае происходит увеличение счетчика ссылок smap. Иначе операция запрашивает свободную структуру smap и сохраняет в ней данные о vnode и смещении. Последним этапом выполнения операции является возврат виртуального адреса страницы, представленной smap.

После этого файловая система вызывает uiomove() для копирования данных со страницы в адресное пространство процесса. Если необходимая страница не находится в памяти, либо ее преобразование отсутствует в HAT, возникает исключение. Обработчик исключений вызывает segmap_fault() для решения проблемы. Обработка ошибки заключается в вызове операции VOP_GETPAGE файловой системы для получения страницы в памяти и последующего вызова hat_memload() для загрузки преобразования адреса страницы.

После копирования данных в адресное пространство задачи файловая система вызывает segmap_release() для обратного переноса элемента smap в список свободных вхождений (если он не имеет других ссылок). В случае выполнения системного вызова write на этом этапе работы операция segmap_release() также сбрасывает информацию со страницы обратно в файл (как правило, это происходит синхронно). Страница остается отображенной по тому же смещению и vnode до тех пор, пока не будет отсоединена принудительно. Преобразование адреса страницы кэшируется, что позволяет быстро обнаружить ее при возникновении такой необходимости.

Читатель может сказать: а не слишком ли большой набор операций используется только лишь для чтения файла? Однако все эти действия необходимы, так как главной ролью функций драйвера seg_map является интеграция работы с файлами посредством read или write с доступом через отображения.

Такой подход дает гарантию, что даже в случае одновременного обращения к одному и тому же файлу двумя методами только одна копия страницы окажется в памяти и не нарушится корректность данных.

Задача переноса информации из файла в страницу памяти лежит на операции VOP_GETPAGE. Иногда необходимая страница уже находится в памяти (через таблицу хэширования seg_map можно обнаружить лишь те страницы, которые отображены в сегмент типа seg_map, то есть к этим страницам некоторое время назад происходили вызовы read или write). Операция VOP_GETPAGE получает страницу либо из памяти, либо из файла, как это было описано в разделе 14.8.1.

Важным требованием унификации доступа к файлам является необходимость применения единого пространства имен для файлов и страниц процессов. В традиционных реализациях UNIX буфер файловой системы идентифицируется парой <device, block number> (<имя устройства, номер блока>). В архитектуре VM для определения пространства имен служит пара <vnode, offset>, для локализации страницы по имени используется глобальная таблица хэширования. При необходимости обращения к диску подсистема VM вызывает функцию файловой системы для преобразования имени в адрес страницы на диске.

Буферный кэш по-прежнему поддерживается системой. Он требуется для кэширования метаданных файловой системы (суперблоков, индексных дескрипторов, блоков косвенной связи и каталогов), то есть тех данных, имя которых нельзя представить в виде пары <vnode, offset>. Система поддерживает буферный кэш небольшого объема только с целью хранения метаданных.

14.8.3. Важные замечания

Размер страницы может не совпадать с размером блока файловой системы. Кроме этого, на одной машине могут сосуществовать сразу несколько файловых систем, имеющих неодинаковые размеры блоков. Подсистема VM работает только со страницами, запрашивая файловую систему о загрузке страниц в память и сбросе их на диск. Задача чтения данных с диска лежит на менеджере объектов vnode файловой системы. Получение страницы данных может потребовать чтения одного блока, сразу нескольких блоков или всего лишь части блока. Если размер файла не пропорционален размеру страниц (то есть при чтении часть последней страницы окажется пустой), менеджер vnode заполнит эту часть нулями.

Система VM использует пространство имен, определяемое парой <vnode, offset>. Таким образом, при каждом обращении к диску файловая система должна производить пересчет смещения в номер физического блока на диске. Если информация о преобразовании отсутствует в памяти (например, используются блоки косвенной связи и т. д.), ее также необходимо прочесть с диска. Если это действие необходимо произвести при выполнении выгрузки стра-

ниц из памяти, операция VOP_PUTPAGE должна уметь предупреждать взаимоисключения. Они могут возникнуть в том случае, если в системе не хватает памяти для чтения блоков косвенной связи, в которых и находится информация о «грязных» страницах, чье содержимое требуется сбросить на диск.

14.9. Виртуальное пространство свопинга в Solaris

Реализация уровня свопинга в операционной системе SVR4 обладает несколькими недостатками. Размер анонимной памяти, доступной для выделения системой, ограничено объемом физического пространства свопинга. При размещении страниц на резервных носителях их физическое месторасположение выбирается случайно из доступного пространства свопинга и может быть изменено только при удалении устройства свопинга из системы. Реализация не обладает возможностями интеллектуального управления пространства свопинга, что позволило бы оптимизировать работу как страничной системы, так и уровня свопинга.

В системе Solaris 2.x было впервые применено понятие виртуального пространства свопинга, реализованного с целью избавления от ограничений, имеющихся в SVR4 [2]. При этом разработчики поставили перед собой следующие цели:

- увеличение доступного пространства свопинга путем задействования физической памяти;
- возможность динамического изменения расположения страниц в пространстве свопинга.

В операционной системе Solaris была представлена новая файловая система swapfs, в которой виртуальное резервное хранение было перенесено на уровень анонимности. В этом разделе книги мы рассмотрим некоторые возможности уровня свопинга системы Solaris.

14.9.1. Расширенное пространство свопинга

Как и в системе SVR4, ОС Solaris требует от клиентов подсистемы анонимной памяти резервировать пространство свопинга сразу, то есть при создании сегмента. Однако в Solaris доступное пространство равняется объему всех физических устройств свопинга плюс почти весь доступный объем физической памяти. При этом часть физической памяти всегда остается свободной с целью поддержания некоторого количества места для размещения структур ядра. Это позволяет системе Solaris работать с меньшим объемом пространства свопинга по сравнению с SVR4.

Резерв пространства свопинга в физической памяти используется только при исчерпании всего возможного пространства на устройствах свопинга. Область памяти, используемая для свопинга, является «зашитой» (*wired down*) (сделанной нестраничной). При выполнении операции освобождения пространства свопинга система в первую очередь делает все, чтобы сохранить резервы в оперативной памяти. То есть основная память машины используется для свопинга в последнюю очередь и освобождается при первой же возможности.

После резервирования области памяти система может создавать анонимные страницы, не обладающие физическими резервными копиями. Имена для таких страниц предоставляются файловой системой swapfs (см. следующий раздел). Так как область памяти для свопинга не является страничной, такие анонимные страницы не могут быть выгружены, и, следовательно, не требуют физического пространства свопинга для хранения.

14.9.2. Управление виртуальным свопингом

При создании процессом анонимной страницы в первую очередь необходимо выделить для нее пространство свопинга. В отличие от резервирования, при размещении происходит присвоение странице определенного адреса в области свопинга, или назначение *имени*. В SVR4 такие адреса всегда указывают на физическое устройство свопинга. Позиция структуры anon массива anon устройства равна позиции страницы в устройстве свопинга. Таким образом, адрес структуры anon выступает в качестве имени страницы. Для присвоения странице адреса применяется процедура *уровня свопинга* swap_alloc(), результатом выполнения которой является указатель на соответствующую структуру anon.

В ОС Solaris операция swap_alloc() производит присвоение виртуального адреса свопинга страницы. Она управляет специальным системным файлом и выделяет необходимое пространство свопинга из него. Имя виртуальной страницы свопинга описывается vnode «виртуального» файла и смещением страницы внутри этого файла. Вместо использования набора массивов anon (по одному на каждое устройство) операция выделяет структуры anon динамически. Она сохраняет имя страницы (в виде пары <vnode, offset>) в структуре anon вместо вычисления его из адреса структуры. Следует отметить, что в этой точке выполнения процедура еще не произвела выделения и ограничения какого-либо физического пространства свопинга для страницы (см. рис. 14.15, *а*).

Физическое пространство свопинга становится необходимым странице только при выгрузке ее из памяти. Для этого демон pageout получает имя страницы и вызывает оператор VOP_PUTPAGE по отношению к соответ-

ствующему объекту vnode. Если страница является анонимной, это приводит к запуску процедуры файловой системы swapfs, входными параметрами которой являются vnode и смещение виртуального файла. Процедура производит следующие действия:

1. Вызывает *уровень свопинга* для выделения резервного пространства страницы.
2. Вызывает оператор VOP_PUTPAGE физического устройства свопинга для сброса данных страницы.
3. Производит запись нового имени страницы (vnode и смещение физической страницы свопинга) в ее структуры anon и page (рис. 14.15, *б*).

Через какое-то количество времени страница оказывается выгруженной из памяти. После этого страница может быть восстановлена из области свопинга (если возникнет такая необходимость) на основе информации из ее структуры page (рис. 14.15, *в*).

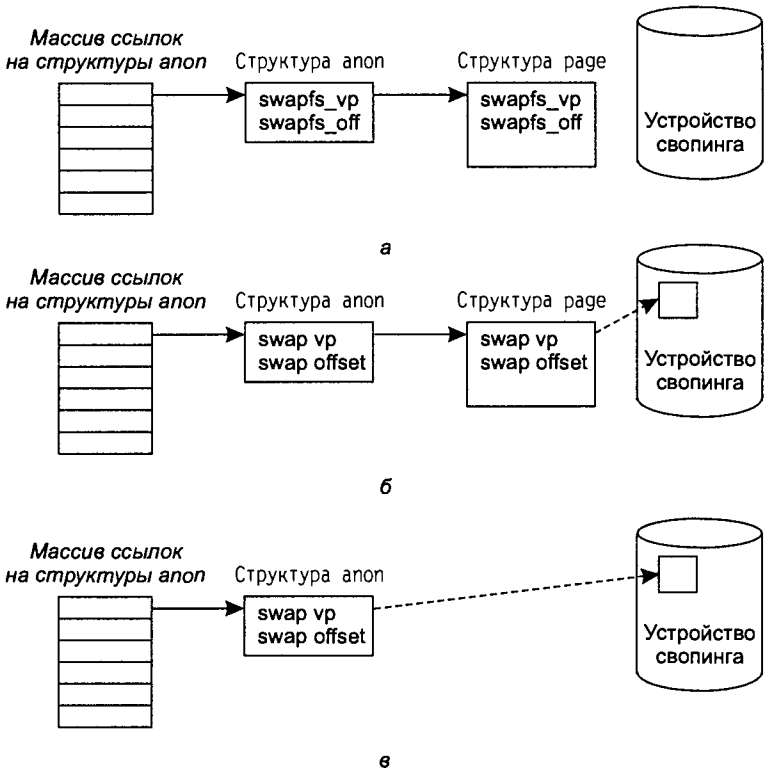


Рис. 14.15. Выделение виртуального и физического пространства свопинга: *а* — начальное состояние анонимной страницы; *б* — после операции выгрузки; *в* — после освобождения страницы в памяти

Файловая система swafs поддерживает выделение пространства свопинга в основной памяти. При проведении операции выгрузки страницы иногда возникает ситуация нехватки места в области свопинга на дисках. В этом случае система помещает страницу в область свопинга, располагаемую в памяти, не помечая такую страницу как чистую.

14.9.3. Некоторые выводы

Реализация виртуального свопинга в ОС Solaris позволяет системе работать при небольших объемах области свопинга (ниже 20% от объема оперативной памяти) без снижения при этом своей производительности. Такая возможность может оказаться весьма полезной, если объем диска ограничен.

Базовая структура системы Solaris имеет дополнительные средства, позволяющие производить интеллектуальное управление свопингом. Например, демон pagedaemon может группировать операции записи анонимных страниц. Это позволяет уровню свопинга выделять для этой цели непрерывное пространство свопинга и производить сброс страниц одной операцией ввода-вывода. В дополнение к сказанному, файловая система swapfs способна назначать отдельный объект vnode каждому клиентскому процессу. Затем она может организовать место для резервного хранения страниц одного и того же процесса таким образом, что все страницы будут располагаться рядом друг с другом на одном устройстве свопинга, что сказывается в лучшую сторону на быстродействии страничной системы.

14.10. Анализ

Архитектура VM сильно отличается от архитектуры управления памятью, реализованной в 4.3BSD. С одной стороны, VM кажется более сложной и обладающей большим числом основных понятий и базовых структур. С другой стороны, четкое разделение операций системы на несколько уровней позволило создать строгий интерфейс взаимодействия между компонентами ОС. Однако, главным мерилом, по которому можно произвести сравнение, является функциональность и производительность работы. Что получили пользователи новой архитектуры, и за счет чего это произошло? Архитектура VM имеет ряд важнейших преимуществ. Перечислим их.

- Модульность структуры. Каждый основной компонент подсистемы представлен элементом объектно-ориентированного интерфейса, что позволяет наследовать возможности и скрывать внутреннюю реализацию элемента от остальной части системы. Это преимущество следует из свойств объектно-ориентированного подхода, в котором каждый

компонент легко может быть видоизменен или усовершенствован, добавлена поддержка новых средств или новых аппаратных архитектур. Одним из примеров такого усовершенствования можно назвать добавление драйвера seg_u для управления размещением области u.

- Изоляция средств аппаратного преобразования адресов внутри уровня HAT сделала архитектуру VM легко переносимой на новые платформы. Подсистема VM уже перенесена на такие популярные системы, как Motorola 680x0, Intel 80x86, SPARC, AT&T 3B2 и IBM 370/XA.
- Архитектура поддерживает различные формы коллективного использования памяти: разделение отдельных страниц в режиме копирования при записи, отображения MAP_SHARED на анонимные объекты для традиционных разделяемых областей памяти и общий доступ к файлам посредством интерфейса mmap. Такая способность уменьшает загрузку физической памяти и избавляет от необходимости вызова лишних дисковых операций по управлению несколькими копиями одной и той же страницы в памяти.
- Интерфейс mmap является весьма мощным не только из-за предлагаемой поддержки разделения памяти, но и в силу возможности прямого доступа к данным файла без перегрузки системы вызовами.
- Несмотря на то что разделяемые библиотеки не являются частью ядра, они могут быть легко отображены в адресное пространство процесса.
- Архитектура VM использует операции vnode для всех случаев обращения к файлам и дискам, что позволяет применить имеющиеся преимущества интерфейса vnode в полной мере. В частности, система VM не требует создания дополнительных кодов для поддержки выполнения файлов на удаленных узлах. В дополнение, система может указывать удаленные диски в качестве устройств свопинга, что дает право говорить о настоящей поддержке бездисковых систем.
- Интеграция буферного кэша с системой VM предоставляет возможность автоматической настройки физической памяти. В традиционных архитектурах размер буферного кэша являлся постоянным и указывался при сборке ядра (равняясь примерно 10% объема физической памяти). Однако идеальный размер кэша не постоянен и зависит от применения конкретной системы. Буфер большого объема потребуется для систем, производящих интенсивный ввод-вывод (например, в файл-серверах), в то время как системам разделения времени, используемым для выполнения приложений, требуется много оперативной памяти для осуществления страничного обмена, но достаточно небольшого дискового буфе-

ра. Даже на автономных машинах размер дискового кэша желательно варьировать при изменении конфигурации системы. Архитектура VM умеет динамически поддерживать баланс между страницами процессов и страницами файлов в зависимости от текущих потребностей в памяти, что позволяет эффективно решить проблему удовлетворения нужд всех процессов системы в любой момент времени.

- Полностью решена проблема точек останова, устанавливаемых в разделяемых областях текстов программ (см. раздел 13.6): все текстовые области имеют атрибут MAP_PRIVATE. При задании отладчиком контрольной точки (вызовом ptrace) ядро создает закрытую копию страницы для процесса и записывает в нее данные о точке останова. Такой подход гарантирует невидимость контрольной точки для остальных процессов, разделяющих текстовую область программы.

Несмотря на то что преимущества VM кажутся просто потрясающими, система не лишена недостатков. Ниже перечислены основные слабости подсистемы, влияющие на ее производительность.

- Подсистеме VM необходимо поддерживать большой объем информации о своих основных абстракциях. Результатом этого является появление большего количества структур данных по сравнению с BSD, часто оказывающихся объемнее своих предшественниц. Например, структура page имеет размер как минимум 40 байтов, в то время как структура cmap системы BSD — всего 16 байтов. Это означает, что подсистема VM использует для поддержки своего функционирования больший объем памяти, ограничивая тем самым память, доступную процессам.
- VM не сбрасывает страницы текстов программ и неинициализируемых данных в область свопинга. Вместо этого система при необходимости считывает информацию прямо из исполняемых файлов. Это приводит к сокращению объема пространства свопинга, необходимого для работы, и позволяет сэкономить на операциях сброса таких страниц на резервные носители. С другой стороны, чтение страницы из файла происходит медленнее по сравнению с загрузкой ее из области свопинга. Причиной этого является необходимость выполнения большего количества кодов файловой системы и (в некоторых случаях) чтение дополнительных блоков метаданных с целью обнаружения страницы на диске. Влияние политики работы со свопингом подсистемы VM на общую производительность системы зависит от частоты запроса таких страниц.
- Алгоритмы VM являются более сложными и медленными по сравнению с предшественниками. Разнесение компонентов на разные уров-

ни выполнения требует применения большего числа функциональных вызовов, часть из которых вызывается косвенно путем просмотра таблиц функций. Это влияет на общую производительность системы. Например, в [3] упоминается тот факт, что функциональные вызовы на 20% увеличивают время выполнения обработки страничных исключений.

- Система VM не поддерживает практику BSD сохранения в PTE дисковых адресов всех страниц, заполняемых текстами, при выполнении exec. Это означает, что каждый дисковый адрес страницы будет вычисляться индивидуально при возникновении исключения применительно к этой странице. Если блоки косвенной адресации файла не находятся в физической памяти, их необходимо считывать с диска. Это уменьшает скорость работы страничной системы в отношении страниц текста и инициализируемых данных.
- Применение объектно-ориентированного подхода привело к появлению инвариантных интерфейсов взаимодействия с основными абстракциями VM, достаточно гибких, чтобы создавать различные реализации объектов. Отсюда следует, что система не может быть настроена на какую-то специфическую реализацию. Оптимизации, описываемые в предыдущем абзаце, невозможны в SVR4.
- Технология копирования при записи не всегда работает быстрее упреждающего копирования. Когда страница сдублирована с помощью копирования при записи, в результате возникает большее количество исключений. При этом необходимо обновлять вхождение TLB, так как страница изначально защищена от записи. Если страницу все равно придется копировать, то более эффективным решением этой задачи будет прямое копирование по сравнению с ожиданием исключения.
- Пространство свопинга выделяется постранично, а размещение страниц на устройствах произвольно. Такой подход не позволяет применять преимущества кластеризации, поддерживаемой BSD, заключающиеся в выделении непрерывных областей пространства свопинга для каждого процесса. Но следует заметить, что в BSD незанятое пространство оказывается потерянным.

Все перечисленные минусы влияют на производительность и могут быть компенсированы достоинствами новых средств системы, таких как разделение памяти или отображение файлов. С другой стороны, постоянное увеличение скорости процессоров и размеров памяти делает уступки производительности не столь значительными по сравнению с функциональностью, по части которой архитектура VM имеет множество преимуществ.

14.11. Увеличение производительности

В те времена, когда архитектура VM была впервые перенесена из SunOS в SVR4, ее скоростные характеристики оказались ниже по сравнению с сегментной архитектурой SVR2 и SVR3 UNIX. В частности, было обнаружено, что системе VM присуще большое количество исключений (страничных ошибок) в условиях испытаний обычными многопользовательскими тестами. Подробный анализ системы привел к некоторым улучшениям ее работы, которые впоследствии были поддержаны в SVR4.

14.11.1. Причины большого количества исключений

Одной из важнейших проблем систем является интерпретация карт аппаратного преобразования адресов. В архитектуре VM такие карты ассоциируются с адресным пространством, а не с сегментами по отдельности. Сегмент может начинаться с границы любой страницы, одну карту преобразований могут разделять сразу несколько сегментов. Из этого следует, что два процесса не в состоянии разделять между собой карту трансляции адресов, так как в этом случае они получат в совместное использование и все сегменты, отраженные в ней.

Страница памяти разделяется несколькими преобразованиями, поэтому все изменения одного отображения должны быть перенесены во все преобразования для этой страницы. Существует два типа изменений, каждое из которых обрабатывается по-разному. Первым типом является изменение атрибута корректности страницы на некорректный (valid-to-invalid). Примером его применения является сброс бита корректности страницы демоном pagedaemon с целью имитации бита ссылки (см. раздел 13.5.3). Подсистема вносит такие изменения во все преобразования страницы незамедлительно, так как противное грозит неправильным поведением системы.

Второй тип — это обратное изменение атрибута страницы с некорректного на корректный (invalid-to-valid). Представьте ситуацию, когда два процесса разделяют между собой страницу, сброшенную в текущий момент времени из памяти. Когда первый процесс обращается к странице, возникает страничная ошибка, которая обрабатывается путем загрузки страницы в память и создания действительного преобразования адреса страницы для этого процесса. Система VM использует подход, согласно которому второй процесс не уведомляется о произошедших изменениях сразу. Передача происходит тогда и только тогда, когда второй процесс обратится к той же странице (что станет причиной нового исключения). В этом случае обработчик найдет страницу в памяти и создаст корректное преобразование.

Преимущество такого подхода (часто называемого принципом отложенных вычислений) в том, что, откладывая действие, есть вероятность избавиться от необходимости производить его вообще. Второй процесс может не обратиться к странице или сделать это уже после выгрузки страницы из памяти. Недостатком подхода является большое количество страничных ошибок (исключений). Некоторые из них являются ложными: страница на самом деле находится в памяти, но запрашивающий ее процесс не обладает корректным преобразованием для нее.

Оригинальная реализация VM использует принцип отложенных вычислений также при инициализации карт преобразования адресов. В отличие от системы BSD, в которой размещение и инициализация всех таблиц страниц происходит во время выполнения fork или exec, подсистема VM старается по возможности отсрочить эту задачу. Инициализация каждого элемента карты происходит только при первом обращении процесса к странице. Точно так же выделение каждой страницы карты происходит только при возникновении такой необходимости. Несмотря на то, что описываемый подход позволяет избавиться от некоторого объема вычислений, его применение приводит к увеличению количества системных исключений.

Преимущества «отложенного» подхода становятся очевидными в том случае, когда перегрузка системы вследствие роста числа исключений компенсируется экономией времени на излишние операции. В подсистеме VM количество исключений очень велико. Измерения на машине AT&T 3B3/400 показали, что обработка мнимых ошибок памяти (то есть возникающих при наличии страницы в памяти при отсутствии ее корректного преобразования) занимает 3,5 мс в архитектуре VM по сравнению с 1,9 мс в сегментной архитектуре *SVR2/SVR3*. Появление такой разницы обусловлено, в первую очередь, модульностью архитектуры VM, результатом которой является потребность в большем количестве вызовов и «разбухании» исходных текстов.

Сходные тенденции наблюдаются и при разделении страниц между процессами на основе копирования при записи (на стадии выполнения fork). Цель методики состоит в копировании только тех страниц памяти, которые были изменены родителем или предком. Для этого подсистема VM откладывает копирование всех страниц до тех пор, пока родительский процесс или его потомок не вызовут исключение. Недостатком метода опять же является увеличение числа исключений. На машине 3B2 обработка исключения и копирование страницы занимает 4,3 мс, в то время как само копирование требует всего только 1 мс. Следовательно, применение копирования при записи будет выгоднее лишь в том случае, если достаточно копировать менее четверти от общего количества страниц. Если же количество изменяемых страниц после вызова fork превышает это значение, то наиболее быстрым окажется копирование всех страниц.

Измерения показали, что в первоначальной реализации VM возникает примерно в четыре раза больше исключений по сравнению с сегментной ар-

хитектурой. Системные вызовы fork и exec оказались наиболее критичны, так как именно они отвечают за настройку адресного пространства процессов и разделение памяти. Знание этого важно, разработчикам системы необходимо представлять, в каких наиболее распространенных ситуациях выделения страниц следует быть готовыми к страничным ошибкам.

В следующих реализациях подсистемы VM появились три новых дополнения, позволивших решить вышеописанные проблемы. Им будет посвящен следующий подраздел.

14.11.2. Новые возможности подсистемы VM в SunOS

Первым и наиболее значимым расширением VM стало появление возможности выделения и инициализации карт преобразований адресного пространства процесса-предка по время выполнения fork. Даже в том случае, если предок через некоторый промежуток времени вызовет exec, как правило, между вызовами fork и exec происходит выполнение некоторого количества прикладного кода (например, перенаправление ввода-вывода, закрытие нескольких дескрипторов и т. д.). Эти инструкции становятся причиной определенного количества ложных исключений. Обработка этих ошибок занимает практически то же время, что и построение и копирование карт преобразования адресов.

Вторым изменением подсистемы является частичная инициализация карт преобразования адресов при выполнении exec. Каждый сегмент ассоциируется с vnode, который в свою очередь, поддерживает связанный список всех страниц в памяти, относящихся к этому объекту. В новой версии VM системный вызов exec просматривает список и инициализирует все вхождения карты преобразования, отображающие такие страницы. Он также выделяет новые карты преобразования, необходимые для таких вхождений. Это предупреждает возникновение ложных исключений, причиной которых является принцип отложенных вычислений.

Описанный метод производит приблизительное определение рабочего набора сегмента путем подсчета количества страниц vnode, уже находящихся в памяти. Часть из этих действий может оказаться излишними, так как процесс не всегда обращается к страницам, еще не выгруженным из основной памяти. С другой стороны, такой подход явно увеличивает производительность системы, поскольку потери на настройку отображений меньше по сравнению с обработкой дополнительных исключений.

Еще одно нововведение относится к разделению памяти на основе копирования при записи. Затраты на копирование всех страниц остаются попрежнему высокими (по этой причине в системе продолжает использоваться методика копирования при записи), поэтому важно уметь предугадать, какие

из них необходимо дублировать в любом случае. Применение копирования при записи происходит, как правило, при выполнении вызова fork. Эта функция чаще всего вызывается интерпретаторами для реализации выполнения указанной команды. Анализ схем использования памяти несколькими различными интерпретаторами (sh, csh, ksh и т. д.) показал, что каждый процесс вызывает fork по несколько раз, при этом выполнение каждого последующего вызова очень похоже на выполнение предыдущего. Как правило, после каждой операции fork производится изменение одних и тех же переменных. В терминах страничной подсистемы это означает, что набор страниц, послуживших причиной исключения при первом вызове fork, скорее всего останется тем же самым при последующем вызове.

При внесении изменений в страницу, имеющую атрибут копирования при записи, страница становится анонимной. Это позволяет легко произвести оптимизацию системы. В новой реализации VM системный вызов fork просматривает набор анонимных страниц родительского процесса и производит физическое копирование тех из них, которые оказались в памяти. Ожидается, что именно эти страницы будут модифицированы после завершения работы fork. Упреждающее копирование позволяет уменьшить загрузку системы по обработке мнимых исключений.

14.11.3. Итоги

Перечисленные выше улучшения системы VM значительно сократили общее количество исключений (страничных ошибок). Из табл. 14.1 видно, что общее число исключений новой реализации системы значительно меньше по сравнению с оригинальной версией VM и сегментной архитектурой.

Таблица 14.1. Результаты многопользовательских тестов различной направленности

Реализация	Ложные срабатывания	Ошибки защиты
SVR3	1 172	1 306
VM, порт инициализации	3 040	1 098
VM с усовершенствованным fork	1 116	1 273
VM с усовершенствованными fork и exec	840	1 122
VM с fork, exec и копированием при записи	640	142

«Отложенный» подход широко применяется как в системе SVR4, так и Mach. Однако, к чему и ведет наше повествование, преимущества этой методики остаются спорными. Как мы видим, при применении принципа отложенных вычислений следует быть очень осторожными, пользуясь им только в том случае, если его эффективность доказана.

14.12. Заключение

В этой главе описывалась архитектура SVR4 VM. Она обладает дополнительными средствами и повышенной функциональностью, а также предлагает различные формы наследования и разделения памяти, необходимые для сложных приложений. Однако для достижения высокой производительности работы необходима тщательная оптимизация системы. Современные тенденции развития ОС заключаются в добавлении новых средств и ожидании появления более быстрых аппаратных средств, которые бы компенсировали усложнение систем[1].

Выделение памяти ядром в SVR4 описывалось в разделе 12.7. Об интерпретации понятия целостности буфера преобразования в этой системе можно прочесть в разделе 15.11. Архитектура управления памятью Mach, имеющая определенное сродство с SVR4 VM, будет обсуждаться в разделе 15.2.

14.13. Упражнения

1. В каких случаях обращение к отображению файлов семантически отличается от доступа к ним при помощи системных вызовов read или write?
2. Можно ли сохранить семантику вызова mmap неизменной при использовании его в распределенных файловых системах? Опишите эффекты, которые могут возникнуть при импортировании отображений файлов в серверы DFS, RFS и NFS?

---

[1] Очень спорное утверждение. Никлаус Вирт в статье «Долой жирные программы» оппонирует: «Ограничения на вычислительные ресурсы не считаются сколь либо серьезными и с легкостью игнорируются: кажется, присутствует всеобщая вера в то, что быстрый рост скорости процессоров и размеров памяти компенсируют допущенные при проектировании ПО небрежности... взрывной рост размеров программного обеспечения был бы, конечно, неприемлем, если бы не ошеломляющий прогресс полупроводниковой технологии... Так, с 1978 по 1993 гг. для семейства процессоров Intel 80x86 производительность увеличилась в 335 раз, плотность размещения транзисторов — в 107 раз, в то время как цена — только в 3 раза. Перспективы для постоянного увеличения производительности сохраняются, при том, что нет никаких признаков, что волчий аппетит, присущий ныне программному обеспечению, будет в обозримом будущем утолен». Вирт цитирует Сирила Паркинсона: «Программное обеспечение увеличивается в размерах до тех пор, пока не заполнит всю доступную на данный момент память», — и в этом смысле его нельзя опровергнуть. Хотя наблюдение Гордона Мура, сделанное в 1965 г., до сих пор подтверждается: с выпуска микропроцессора 4004 в 1971 г. и вплоть до появления Pentium 4 количество транзисторов выросло более чем в 18 000 раз — с 2 250 до 42 000 000, программы увеличиваются в размере ничуть не медленнее. Согласно статистике, собранной известным криптологом Брюсом Шнайером, количество системных вызовов UNIX прогрессировало с 33 в первой редакции 1971 г. до 230 в версии Linux 2.0 1998 года и 190 в SunOS5.6. Windows 3.1 (1992 г.) имела 3 миллиона строк кода, а Windows XP (2002 г.) — уже 45 миллионов. Можно добавить, что объем кода Debian GNU/Linux 2.2 — 55 млн строк. Но при этом, например, сокращенная на 10,8 Мбайт Windows 95 (без 164 файлов) сохраняет достаточную функциональность. И не стоит забывать о дополнении к закону Мура — законе Артура Рока (стоимость основных фондов, используемых в производстве полупроводников, удваивается каждые 4 года). — *Прим. ред.*

3. Чем отличается структура page системы SVR4 от структуры cmap ОС 4.3BSD?
4. С какой целью в структуре as хранится подсказка, указывающая на сегмент, ставший причиной последнего исключения (страничной ошибки)?
5. Чем отличается анонимная страница от анонимного объекта?
6. Почему анонимные страницы не требуют резервных носителей?
7. Почему в системе имеется всего лишь один сегмент seg_map?
8. В ОС SVR4 выделение страниц свопинга откладывается до тех пор, пока процесс не создаст анонимную страницу, в то время как в системе 4.3BSD все необходимое пространство свопинга выделяется заранее при создании процесса. Опишите преимущества и недостатки каждого подхода.
9. Создается ли новый сегмент при каждом вызове функции mmap? Всегда ли это сегмент vnode?
10. В каких случаях процессы разделяют anon_map?
11. Почему структура разделяемой памяти IPC запрашивает ссылку на anon_map сегмента?
12. Каким образом файловая система поддерживает подсистему VM?
13. Почему в системе SVR4 не используется буферный кэш для страниц данных файлов?
14. Почему SVR4 и Solaris выделяют необходимое пространство свопинга во время создания сегмента?
15. В каких ситуациях в SVR4 применяется упреждающее размещение страниц? Назовите преимущества и недостатки такого подхода.
16. Что представляет собой принцип отложенных вычислений? В каких случаях он применяется в SVR4?
17. Назовите преимущества и недостатки применения методики копирования при записи.

14.14. Дополнительная литература

1. Balan, R., and Gollhardt, K., «A Scalable Implementation of Virtual Memory HAT Layer for Shared Memory Multiprocessor Machines», Proceedings of the Summer 1992 USENIX Technical Conference, Jun, 1992, pp. 107–115.
2. Charlock, H., and Snyder, P., «Virtual Swap Space in SunOS», Proceedings of the Autumn 1991 European UNIX Users' Group Conference, Sep. 1991.

3. Chen, D., Barkley, R. E., and Lee, T. P., «Insuring Improved VM Performance: Some No-Fault Policies», Proceedings of the Winter 1990 USENIX Technical Conference, Jan. 1990, pp. 11–22.
4. Ellis, M. A., and Stroustrup, B., «The Annotated C++ Reference Manual», Addison-Wesley, Reading, MA, 1990.
5. Gingell, R. A., Moran, J. P., and Shannon, W. A., «Virtual Memory Architecture in SunOS», Proceedings of the Summer 1987 USENIX Technical Conference, Jun. 1987, pp. 81–94.
6. Kleiman, S. R., «Vnodes: An Architecture for Multiple File System Types in Sun UNIX», Proceedings of the Summer 1986 USENIX Technical Conference, Jun. 1986, pp. 238–247.
7. Moran, J. P., «SunOS Virtual Memory Implementation», Proceedings of the Spring 1988 European UNIX Users Group Conference, Apr. 1988.
8. UNIX System Laboratories, «Operating System API Reference, UNIX SVR4.2», UNIX Press, Prentice-Hall, Englewood Cliffs, NJ, 1992.

Глава 15
Дополнительные сведения об управлении памятью

15.1. Введение

Эта глава посвящена обсуждению трех различных тем. Сначала мы поговорим об архитектуре виртуальной памяти ОС Mach, обладающей таким уникальным свойством, как реализация большинства возможностей системы через задачи прикладного уровня. Вторая часть главы будет посвящена вопросам корректности буфера ассоциативной трансляции (TLB) в многопроцессорных системах[1]. Последние разделы главы описывают проблемы, связанные с обеспечением достоверности и эффективности функционирования виртуально адресуемых кэшей.

15.2. Структура подсистемы управления памятью Mach

Операционная система Mach разработана в середине 80-х годов в Университете Карнеги—Меллона. Создание архитектуры памяти для этой ОС шло практически параллельно с разработками SunOS/SVR4 VM (см. главу 14). Несмотря на то, что разработчики этих двух альтернативных архитектур использовали неодинаковую терминологию для описания своих методик, цели,

---

[1] В оригинале книги (ранее, здесь и далее) используется слово consistency, имеющее русскую «кальку» *консистентность*, что означает внутреннюю непротиворечивость, соответствие значений всех компонентов объекта его состоянию, но из-за некоторой громоздкости определения в рассматриваемых контекстах мы будем пользоваться понятием корректности (достоверности, действительности, непротиворечивости), то есть говорить о свойстве информации быть правильно воспринятой, вероятности отсутствия ошибок. В рамках теории распределенной обработки данных модель консистентности подразумевает соглашение между программами и памятью. Память не может гарантировать правильность результатов операций ввода-вывода, если правила ее использования нарушаются. Существуют несколько моделей консистентности: строгая, причинная, PRAM (Pipelined RAM), последовательная, по входу и слабая. Их тонкости выходят за уровень изложения данной книги. — *Прим. ред.*

структура и практическая реализация подсистем управления памятью оказались схожими. Многие объекты VM и функции системы SVR4 имеют точные аналоги в Mach. В этом и следующих разделах книги вы увидите описание структуры подсистемы виртуальной памяти операционной системы Mach версии 2.5. Изменения в Mach 3.0 являются незначительными, поэтому они не попали в поле зрения повествования.

Вы не найдете здесь подробного описания всех возможностей Mach VM. Основной причиной этому является сходство архитектур управления памятью в SVR4 и Mach. Вместо этого читателю предлагается сравнительный анализ двух систем. Подробнее остановимся на тех качествах Mach VM, которые не имеют аналогов в SVR4.

15.2.1. Цели, стоявшие перед разработчиками

Так же как и в случае операционной системы SVR4, основной предпосылкой появления новой архитектуры стали ограничения подсистемы управления памятью 4.3BSD, большинство из которых были следствием привязки ОС к архитектуре машины VAX, что сильно затрудняло задачу переноса BSD на другие платформы. Возможности 4.3BSD являются слишком простыми и ограничены поддержкой выделения страниц по запросу. В системе отсутствуют механизмы разделения памяти, что позволяет использовать совместно лишь страницы кодов программ в режиме «только для чтения». Более того, подсистема управления памятью 4.3BSD не годится для применения в распределенных средах. В дополнение к тому, что Mach полностью совместима с 4.3BSD на двоичном уровне, разработчики заложили в нее богатый набор новых возможностей, в том числе:

- разделение памяти между связанными и несвязанными задачами на основе копирования при записи или чтения-записи;
- отображение файлов;
- адресные пространства больших размеров, заполняемые неплотно;
- разделение памяти между процессами, выполняемыми на разных машинах;
- управление правилами замены страниц на прикладном уровне.

В системе Mach все машинно-зависимые коды вынесены на *уровень* pmap. Такой подход упрощает задачу портирования ОС на новые аппаратные платформы, так как разработчикам достаточно переписать заново только уровень pmap. Остальная часть кодов системы является машинно-независимой, то есть не настроенной под какую-либо определенную архитектуру MMU.

Одной из важнейших целей при разработке структуры Mach являлся перенос большинства функций подсистемы VM из ядра наружу. Разработчики Mach стремились к созданию архитектуры микроядра, в которой большин-

ство функций ядра выполняется на уровне прикладных серверных задач. Например, такие механизмы подсистемы VM, как поддержка страниц, реализованы в Mach как внешние задачи прикладного уровня.

Также необходимо сказать о тесной интеграции подсистем управления памятью и взаимодействия процессов (IPC) между собой. Такой подход дал сразу два преимущества. Свойство независимости IPC от местонахождения клиентов (см. раздел 6.9) позволило легко расширить средства виртуальной памяти на распределенные среды. В разделе 15.5.1 показан один из примеров программы прикладного уровня, позволяющей разделение памяти между приложениями, выполняющимися на нескольких машинах. Поддержка разделения памяти на основе технологии копирования при записи позволяет быстро передавать сообщения большого объема.

При описании подсистемы VM мы часто будем пользоваться несколькими фундаментальными понятиями системы Mach. О них мы уже подробно говорили на страницах книги, поэтому упомянем их определения лишь вкратце. *Задача* (task) представляет собой наборов ресурсов, включающих в себя адресное пространство и некоторое количество портов. Внутри задачи может выполняться одна или несколько нитей. *Нить* представлена контрольной точкой программы и является выполняемой и планируемой единицей в составе задачи. В системе Mach традиционный процесс UNIX понимается как задача, имеющая единственную нить. О задачах и нитях Mach можно узнать из раздела 3.7. *Порт* является защищенной очередью сообщений. Правами на отправку сообщений в один и тот же порт могут обладать сразу несколько задач, но лишь единственное задание обладает возможностью получать сообщения из него. *Сообщения* представляют собой типизированные наборы данных. Размер сообщения варьируется от нескольких байтов до целого адресного пространства. Порты и сообщения системы Mach описаны в разделе 6.4. Объекты памяти, предоставляющие возможность резервного хранения страниц, рассматриваются в этой главе.

15.2.2. Программный интерфейс

Операционная система Mach поддерживает большое количество операций управления памятью, которые можно разделить на четыре основные категории [17].

- **Выделение памяти.** Прикладной задаче может быть выделена одна или несколько страниц виртуальной памяти при помощи функций vm_allocate (для страниц, заполняемых нулями) и vm_map (для страниц, отображающих специфические *объекты памяти*, например, файлы). Вызов этих процедур не приводит к незамедлительному получению ресурсов, так как в Mach страницы появляются физически только после первого обращения к ним. Для открепления страниц применяется вызов vm_deallocate.

- **Защита.** Система Mach поддерживает ограничения на чтение, запись и выполнение для каждой страницы, однако их практическая реализация зависит от аппаратной части машины. Многие блоки MMU не распознают права на выполнение. В таких системах аппаратура разрешает выполнение любой страницы, доступной для чтения. Каждая страница памяти имеет атрибуты текущей и максимальной степени защиты. После установки максимальный уровень защиты страницы может быть только снижен. Атрибуты текущего уровня защиты не имеют права превышать максимальные значения. Для изменения обоих типов защиты страниц применяется вызов vm_protect.
- **Наследуемость.** Каждая страница обладает собственным признаком наследуемости, который определяет действия системы при создании задачей потомка посредством вызова task_create. Атрибут имеет три возможных значения (см. рис. 15.1):

VM_INHERIT_NONE	Страница не наследуется потомком и не появляется в его адресном пространстве
VM_INHERIT_SHARE	Страница разделяется между родителем и потомком. Обе задачи обращаются к единственной копии страницы. Изменения, внесенные одной задачей, сразу же становятся видны другой
VM_INHERIT_COPY	Потомок получает собственную копию страницы. В системе Mach такое разделение реализовано с помощью копирования при записи, таким образом данные будут продублированы только в том случае, если родитель или потомок попытаются модифицировать страницу. Этот атрибут является устанавливаемым по умолчанию для всех впервые создаваемых страниц

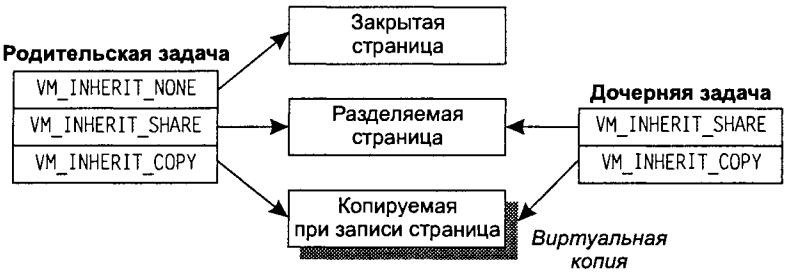


Рис. 15.1. Наследование страниц памяти во время выполнения task_create

Изменение признака наследуемости для набора страниц производится при помощи вызова vm_inherit. Важно упомянуть, что этот атрибут не зависит от текущих установок задачи по разделению страницы. Например, задача A запрашивает страницу и устанавливает ее атрибут наследуемости в значение VM_INHERIT_SHARE. После этого происходит созда-

ние задачи-потомка Б. Задача Б изменяет значение атрибута для этой страницы в VM_INHERIT_COPY, а затем создает задачу В. В итоге страница оказывается совместно используемой в режиме «чтение-запись» задачи А и Б, и разделяемой в режиме копирования при записи с задачей В (см. рис. 15.2). Описанное свойство системы Mach отличает ее от SVR4, где метод наследования определяется текущими установками отображения страницы в процессе (*разделяемый* или *закрытый* доступ).

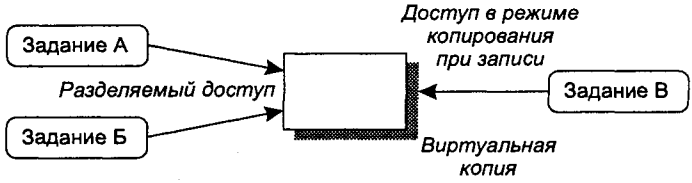


Рис. 15.2. Несколько режимов разделения по отношению к одной и той же странице

- **Другие.** Вызовы vm_read и vm_write позволяют задаче осуществлять доступ к страницам, относящимся к другим задачам. Эти функции по большей части применяются отладчиками и профилировщиками. Вызов vm_regions возвращает информацию обо всех участках адресного пространства. Вызов vm_statistics собирает статистические данные о подсистеме виртуальной памяти.

15.2.3. Фундаментальные понятия

Система Mach разрабатывалась на основе объектно-ориентированного подхода. Все ее основные абстракции представлены в виде объектов, доступных при помощи четко определенного интерфейса. На рис. 15.3 схематично показаны наиболее важные объекты подсистемы управления памятью. Объектом наивысшего уровня является *карта адресации*, описываемая структурой vm_map. В ней находится двунаправленный связанный список вхождений карты адресации и подсказка на последнее вхождение, ставшее причиной возникновения исключения. Каждое *вхождение карты адресации* (структуры vm_map_entry) описывает непрерывную область виртуальной памяти, имеющую одинаковые установки защиты и наследуемости и управляемую одним и тем же объектом памяти. Объект vm предоставляется задачам интерфейсом доступа к страницам объектов памяти. *Объект памяти* (memory object) является абстрактным базовым классом — совокупностью байтов данных, определяющей такие операции, как read или write. Эти операции выполняются *менеджером данных* (data manager или pager) объекта. Менеджер данных представляет собой задачу (прикладного уровня или уровня ядра), управляющую одним или несколькими объектами памяти. В качестве примера объектов памяти можно привести файлы, базы данных и сетевые серверы разделяемой памяти (см. раздел 15.5.1).

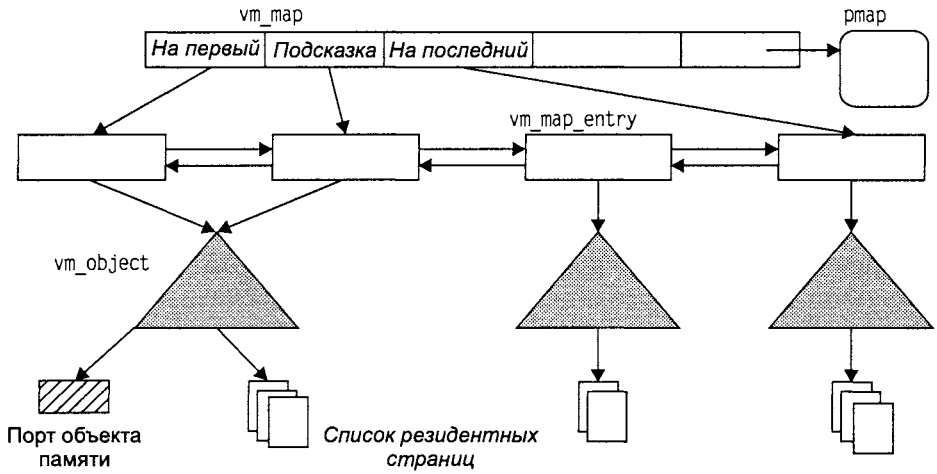


Рис. 15.3. Объекты, описывающие адресное пространство в системе Mach

Объект памяти представлен портом, владельцем которого является менеджер этого объекта памяти. Это означает, что менеджер обладает правами на получение данных, передаваемых в такой порт. Объект vm содержит ссылку (или *право на отправку*) в этот порт и может использовать ее с целью взаимодействия с объектом памяти. Более подробно читайте об этом в разделе 15.4.3. Объект памяти также поддерживает связанный список всех своих резидентных страниц, что позволяет увеличить скорость выполнения таких операций, как освобождение объекта, сброс битов корректности страниц или копирование всех страниц объекта на диск.

Каждый объект памяти ассоциируется с уникальным объектом vm. Если две задачи отображают один и тот же объект памяти в свои адресные пространства, они разделяют между собой и объект vm (см. раздел 15.3). Объект vm также содержит счетчик ссылок на себя.

Сходства подсистемы памяти Mach и SVR4 являются просто поразительными. Объект vm_map подобен структуре as, а объект vm_map_entry аналогичен структуре seg. Роль менеджера памяти близка назначению функций, выполняемых драйверами сегментов (расхождения проявляются лишь в том, что менеджер реализован в виде отдельной задачи). Важным отличием подсистем памяти двух ОС является отсутствие в Mach массива элементов защиты страниц. Если задача изменяет защиту некоторого количества страниц области памяти[1], то система Mach разделит эту область на два[2] отдельных участка, отображаемых на том же объект памяти. Некоторые опе-

[1] В данном контексте слово «область» означает диапазон адресов памяти, описываемый одним вхождением карты адресации.

[2] Или даже на три отдельных участка, если набор страниц окажется не в начале или конце области памяти.

рации могут привести к обратному эффекту, то есть соединению смежных участков памяти.

В системе Mach имеются еще две важные структуры данных, называемые таблицей резидентных страниц и структурой pmap. *Таблица резидентных страниц* представляет собой массив (структуру vm_page[]), в котором каждый элемент описывает одну физическую страницу памяти. Размер физической страницы равен размеру аппаратной страницы, умноженному на некоторую степень числа 2. Физическая память интерпретируется системой как кэш содержимого объектов памяти. Пространство имен страниц образуют пары <object, offset> (<объект, смещение>), указывающие на объект памяти, к которому принадлежит данная страница и смещение от начала этого объекта. Информация о каждой странице таблицы хранится также в следующих трех списках:

- *список объектов памяти*. Соединяет все страницы одного объекта, ускоряет операции открепления объекта и копирования при записи;
- *очереди выделения памяти*. Поддерживаются *страничным демоном* (paging daemon). Страница может находиться в очередях активных, неактивных страниц и свободных страниц;
- *таблицы хэширования «объекта/смещения»*. Применяются для быстрой локализации страницы в памяти.

Массив vm_page[] похож на аналогичный массив page[], имеющийся в SVR4.

Каждая задача Mach располагает машинно-зависимой структурой pmap (аналог уровня HAT системы SVR4), которая служит для описания определяемой аппаратной частью карты преобразования виртуальных адресов в физические. Эта структура невидима для остальной части системы и доступна при помощи процедурного интерфейса. Ниже перечислены некоторые функции интерфейса:

- pmap_create() — вызывается при необходимости в новом адресном пространстве. Создает новую структуру pmap и возвращает указатель на нее;
- pmap_reference() и pmap_destroy() — увеличивают или уменьшают на единицу значение счетчиков ссылок pmap;
- pmap_enter() и pmap_remove() — применяются для ввода и удаления преобразований адресов;
- pmap_remove_all() — удаляет все преобразования адреса физической страницы. Каждая страница может обладать несколькими преобразованиями, поскольку вправе разделяться несколькими задачами (или быть отображенной по нескольким адресам одного адресного пространства);
- pmap_copy_on_write() — понижает защиту всех преобразований страницы до «только для чтения»;
- pmap_activate() и pmap_deactivate() — вызываются при переключениях контекста с целью изменения активной структуры pmap процессора.

15.3. Средства разделения памяти

Операционная система Mach поддерживает разделение чтения-записи или копирования при записи между связанными и несвязанными друг с другом задачами. Задачи наследуют области памяти от своих родителей при вызове task_create. Это позволяет задаче использовать одну и ту же память совместно со своими потомками. Не связанные между собой задачи могут разделять память посредством отображения диапазонов своих адресных пространств в единый для всех объект памяти. В этом разделе мы подробно разберем все перечисленные средства системы.

15.3.1. Разделение памяти на основе копирования при записи

Этот вид разделения памяти имеет место при дублировании вызовом task_create области, имеющей атрибут VM_INHERIT_COPY. Несмотря на то, что такой признак требует от системы создания для потомка собственной копии области памяти, ядро оптимизирует процедуру, откладывая копирование страниц. Если родитель или его предок попытаются внести в такие страницы изменения, в ядре возникнет *исключение*. Только после этого ядро произведет копирование страницы и изменение карт адресации каждого процесса так, чтобы каждый процесс обладал собственной копией страницы. Ядро копирует лишь те страницы, которые подвергаются обновлению родителем или предком, что значительно уменьшает суммарное количество выполняемых им операций.

Реализация разделения памяти по типу копирования при записи основана на том, что структуры vm_map_entry обеих задач имеют ссылку на один и тот же объект vm_object. Ядро также устанавливает для вхождения vm_map_entry флаг, помечающий данную область памяти как копируемый при записи. Для защиты области от записи родителем или потомком ядро вызывает pmap_copy_on_write(). После этого любая попытка записи в страницу приводит к возникновению исключения. Пример совместного использования памяти двумя задачами проиллюстрирован на рис. 15.4. Задача A создала задачу Б. Обе задачи имеют разделяемую область памяти по технологии копирования при записи. Область состоит из трех страниц, управляемых общим объектом vm.

Если любая из этих задач попытается внести в страницу изменения, в системе возникнет исключение (страничная ошибка), так как уровень защиты страниц был понижен до «только для чтения». Обработчик ошибки размещает новую физическую страницу и производит ее инициализацию путем копирования данных страницы, ставшей причиной сбоя. Кроме этого, обработчик должен установить новые отображения, поскольку на этом этапе задачи обладают ссылками на собственные копии страницы, продолжая при этом разделять неизмененные страницы.

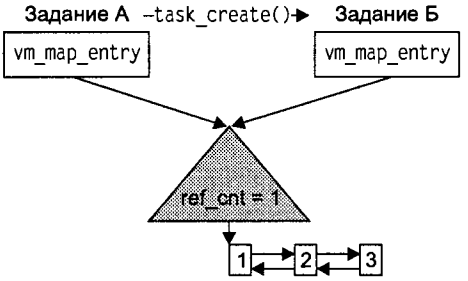


Рис. 15.4. Наследование по технологии копирования при записи

В системе Mach для реализации подобных отображений реализована концепция под названием *теневых объектов* (shadow objects). На рис. 15.5 показано состояние объектов после модификации страницы 1 задачей А и изменения страницы 3 задачей Б. Каждая задача получает в пользование теневой объект, управляющий модифицированными страницами задачи. Теневые объекты имеют указатели на объект, «тенью» которого они являются, в описываемом случае это оригинальные объекты.

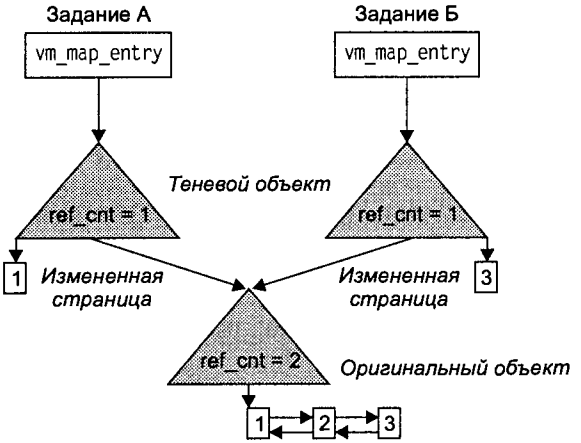


Рис. 15.5. Теневые объекты управляют изменяемыми страницами памяти

При обработке исключения ядро производит обход цепочки указателей теневых объектов в направлении сверху вниз. В нашем примере задача А обнаружит страницу 1 в теневом объекте, а страницы 2 и 3 выберет из оригинального объекта, в то время как задача Б найдет страницу 3 в теневом объекте и страницы 1 и 2 — в оригинальных объектах.

Когда родитель создает новых потомков, может возникать длинная цепочка теневых объектов. Это не только приводит к расходованию ресурсов, но и к снижению скорости обработки ошибок, так как ядру приходится производить просмотр нескольких теневых цепочек. Система Mach обладает алгоритмами, позволяющими обнаружить подобные ситуации и по возмож-

ности сократить цепочки. Если все страницы, управляемые объектом, окажутся в объектах выше его цепочки, то такой объект можно уничтожить. Тем не менее, описанный подход к сжатию цепочек нельзя считать приемлемым. Если страницы объекта переместятся в область свопинга, такой объект удалить будет невозможно.

Разделение по типу копирования при записи также применяется при обмене сообщениями большого размера. Задача может отправлять сообщения, содержащие *внешнюю память* (out-of-line) (см. раздел 6.7.2). Приложения используют такую возможность для передачи друг другу больших объемов данных без физического копирования информации (если это возможно). Ядро отображает такие страницы в адресном пространстве нового процесса путем создания нового вхождения vm_map, разделяющего страницы в режиме копирования при записи с отправителем. Полученная связь сходна с разделением памяти между родителем и его потомком.

Одним из важных различий этих двух способов отображений является тот факт, что во время транспортирования данных ядро временно отображает их в собственном адресном пространстве. Для этого в карте ядра создается новая структура vm_map_entry, а страницы разделяются с отправителем. После того, как данные будут отображены в адресном пространстве получателя, ядро удаляет временный образ. Такой подход позволяет защитить от изменения данных со стороны отправителя раньше, нежели получатель обработает сообщение. Это также упреждает возникновение проблем при завершении работы задачи-отправителя.

15.3.2. Разделение памяти на основе чтения-записи

Разделение в режиме чтения-записи возникает при дублировании вызовом task_create области, имеющей атрибут VM_INHERIT_SHARED. В этом случае в памяти хранится только одна ее копия. Все изменения автоматически становятся видны всем процессам, разделяющим страницы памяти. Изменения могут включать не только модификацию самих данных, но и переопределение уровня защиты и других параметров страницы. Модификация атрибутов может привести к делению области на несколько диапазонов, при этом каждый из участков будет совместно использоваться в режиме чтения-записи задачами, разделявшими до этого оригинальную область.

Описанная форма разделения фундаментально отличается от методики, описанной в предыдущем разделе, поэтому для нее существует отдельная реализация в ядре системы. В ОС Mach для описания разделяемой области памяти в режиме чтения-записи применяется термин *карта разделения* (share map), которая представляет собой структуру vm_map, обладающую флагом, подчеркивающим, что структура управляет участком разделяемой памяти, а также счетчиком ссылок, значение которого равно количеству задач, разделяющих между собой данный участок памяти. Структура содержит

список вхождений vm_map_entry. Изначально в этом списке находится только одно вхождение.

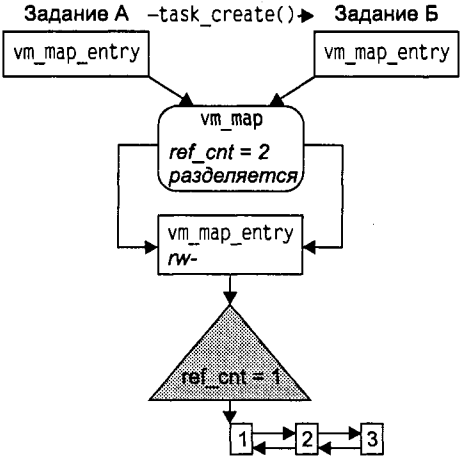


Рис. 15.6. Разделение памяти в режиме чтения-записи

На рис. 15.6 показана реализация карт разделения. Система требует, чтобы вхождение vm_map_entry указывало либо на vm_object, либо на карту разделения. В нашем примере разделяемая область памяти содержит три страницы, изначально доступных в режиме чтения и записи. Через некоторый промежуток времени одна из задач производит вызов vm_protect для присвоения атрибута «только для чтения» странице 3. Это приводит к делению области на два участка, как это показано на рис. 15.7. Применение карты разделения позволяет легко реализовать подобные операции над совместно используемыми областями памяти.

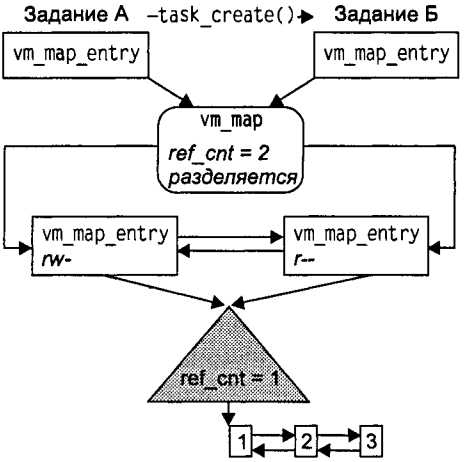


Рис. 15.7. Пример разбиения разделяемой области памяти

15.4. Объекты памяти и менеджеры памяти

Объект памяти представляет собой специфический источник данных, например, файл. Менеджер памяти — это задача, управляющая одним или несколькими объектами памяти и отвечающая за взаимодействие с ядром системы при обмене данными между объектом и физической памятью. Взаимодействие между ядром, менеджером памяти и прикладными задачами реализовано при помощи строго определенного процедурного интерфейса.

15.4.1. Инициализация объектов памяти

Для обращения к данным объекта памяти прикладная задача должна сначала получить права на отправку информации в порт, представляющий данный объект. Эти права запрашиваются у менеджера объекта, так как только владельцу порта разрешена раздача подобных полномочий. Процесс получения прав выходит за рамки описания подсистемы VM и может требовать дополнительного взаимодействия между задачей и менеджером (а также и другими компонентами системы). Например, *менеджер vnode*[1] (vnode pager) отвечает за управление объектами файловой системы и предоставляет возможность прикладным задачам открывать файлы, используя их имена. При открытии файла менеджер возвращает право на отправку в порт, представляющий этот файл.

После получения порта (термин «получение порта» означает, что задаче предоставляется право на отправку данных в порт) задача может отобразить объект памяти в своем адресном пространстве при помощи системного вызова

```
vm_map (pager_task, base_addr, size, mask, flag, memory_object_port, offset,
copy, cur_prot, max_prot, inheritance);
```

Этот вызов аналогичен функции mmap системы SVR4. Вызов vm_map производит отображение байтов диапазона [offset, offset+size) объекта памяти в диапазон адресного пространства [base_addr, base_addr+size) вызывающей его задачи. Параметр flag указывает ядру на разрешение отображать объект по другому базовому адресу. Функция vm_map возвращает адрес, по которому объект был в результате отображен.

При первичной настройке отображения объекта ядро создает для него два дополнительных порта: *порт управления* (control port), используемый

[1] В ранних реализациях системы Mach применялся *менеджер индексных дескрипторов* (inode pager). Для отображений файлов использовался вызов vm_allocate_with_pager [22]. Менеджер vnode появился в системе с целью поддержки интерфейса vnode/vfs [10].

менеджером для создания запросов управления кэшем ядру, и *порт имени* (name port), идентифицирующий объект для других задач, которые могут запросить информацию об объекте через вызов vm_regions. Владельцем этих портов является ядро, которое обладает правами как на отправку, так и на получение данных из них. Инициализация объекта памяти осуществляется вызовом

```
memory_object_init(memory_object_port, control_port, name_port, page_size);
```

Это приводит к получению менеджером памяти прав на отправку запросов в порты запросов и имени. Результат работы описанного набора операций показан на рис 15.8.

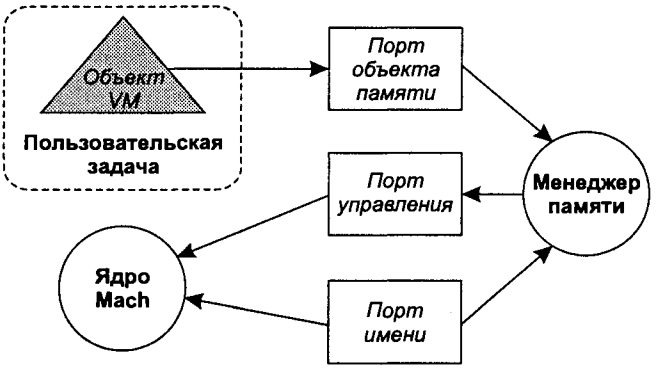


Рис. 15.8. Взаимодействие с менеджером памяти

15.4.2. Интерфейс взаимодействия ядра и менеджера памяти

После настройки каналов взаимодействия ядро и менеджер памяти могут использовать их для посылки запросов друг другу. Каждый запрос является удаленным вызовом процедуры и реализован в виде пересылки сообщения, содержащего параметры запроса, в определенный порт. Ядро отправляет сообщения менеджеру через порт объекта памяти, в то время как менеджер использует для передачи посланий ядру свой порт ответа. Для обмена сообщениями в системе Mach предусмотрен набор функций высокого уровня. Ниже перечислены некоторые функции, позволяющие ядру осуществлять взаимодействие с менеджером памяти:

```
memory_object_data_request (memory_object_port, control_port, offset, length,
desired_access);
memory_object_data_write (memory_object_port, control_port, offset, data, data_
count);
memory_object_data_unlock (memory_object_port, control_port, offset, length,
desired_access);
```

Ядро предоставляет интерфейс взаимодействия, используемый менеджером памяти для управления кэшем. Для этой цели используются следующие функции:

```
memory_object_data_provided (control_port, offset, data, data_count, lock_value);
memory_object_lock_request (control_port, offset, size, should_clean, should_
flush, lock_value, reply_port);
memory_object_set_attributes (control_port, object_ready, may_cache_object,
copy_strategy);
memory_object_data_unavailable (control_port, offset, size);
```

Результатом выполнения каждой из перечисленных функций является создание асинхронного сообщения. Если сообщение инициируется ядром, то менеджер памяти реагирует на него посредством создания ответного послания, используя функции интерфейса с ядром. В следующем подразделе будут показаны некоторые аспекты взаимодействия менеджера памяти с ядром системы.

15.4.3. Обмен данными между ядром и менеджером памяти

Существуют четыре основные области применения, в которых возможно взаимодействие между ядром и менеджером памяти.

- **Страничные ошибки** (page faults). При обработке страничных ошибок ядро просматривает карту адресации на предмет обнаружения необходимого объекта vm, в котором содержится ссылка на объект памяти, позволяющий получить запрашиваемые данные. Затем ядро вызывает memory_object_data_request(), указывая необходимый диапазон страниц и тип доступа. Менеджер возвращает данные ядру посредством вызова memory_object_data_provided(), передавая ему указатель на данные в своем адресном пространстве. Менеджер сам решает, каким образом он получает необходимые данные. Например, их можно считать из файла или взять из сетевого носителя информации.
- **Выгрузка страниц** (pageout). Если ядро системы решает сбросить содержимое «грязной» страницы на носитель, то для этой цели оно отправляет такую страницу менеджеру памяти через вызов memory_object_data_write(). После удачного сохранения страницы менеджер памяти вызывает vm_deallocate для освобождения ресурсов в кэше.
- **Защита** (protection). Ядро может запрашивать дополнительные привилегии доступа к странице (например, доступ в режиме записи для страницы, в данный момент доступной только для чтения). Для этой цели ядро вызывает memory_object_data_unlock(). Менеджер памяти имеет право понизить права доступа страницы функцией memory_object_lock_request(), сделав ее, к примеру, снова доступной только для чтения.
- **Управление кэшем** (cache management). Менеджер может вызвать memory_object_lock_request() с флагом should_flush для запроса ядра на

сброс атрибута корректности копии данных в кэше и выгрузку на носитель всех произведенных модификаций страницы. Для обратной записи всех страниц в указанном диапазоне менеджер вызывает memory_object_lock_request() с флагом should_clean, однако ядро может после этого продолжать использование кэшированных данных. В обоих случаях ядро отвечает на запрос посредством вызова memory_object_data_write(), как и при выгрузке страниц.

15.5. Внешние и внутренние менеджеры памяти

Интерфейс взаимодействия между ядром и менеджером памяти в Mach значительно отличается от традиционных реализаций UNIX. Менеджеры памяти в системе Mach являются отдельными задачами ядра и прикладного уровня. Менеджеры и ядро взаимодействуют друг с другом при помощи обмена сообщениями. Несмотря на то, что такой подход кажется слишком сложным, он обладает некоторыми бесспорными преимуществами. Самые разнообразные менеджеры, обслуживающие самые различные типы объектов вторичного хранения данных, могут быть реализованы в системе в виде задач прикладного уровня. В сравнении с этим система SVR4 позволяет использовать сразу несколько драйверов сегментов, но ядро ОС должно обладать информацией о каждом из них, а также отвечать за управление всеми драйверами. В SVR4 не существует возможности добавления драйверов сегментов, определяемых пользователем. Напротив, ядру Mach не требуется владеть информацией в полном объеме о менеджерах памяти прикладного уровня. Ядро системы взаимодействует с ними через объекты vm.

Технология обмена сообщениями делает взаимодействие расширяемым и независимым от дислокации его участников. Менеджер памяти не обязательно должен выполняться на одной машине. Он может работать на любом удаленном узле сети и представляться на локальной машине через порты-заместители объектов памяти. Такой подход позволяет расширить управление памятью Mach на многопроцессорные системы, в которых машины разделяют между собой некоторую часть памяти на основе обмена сообщениями. В следующем подразделе вы увидите пример, описывающий такое взаимодействие.

Менеджеры памяти, реализованные в виде задач прикладного уровня, называются внешними. Система Mach также поддерживает *менеджер по умолчанию* (default pager), являющийся внутренним, то есть работающим в качестве задачи ядра[1]. Этот менеджер управляет всеми временными объектами

[1] Существует возможность реализации менеджера по умолчанию как прикладной задачи. Одна из таких реализаций описана в [7].

памяти, для которых не требуется постоянное резервное хранение. Такие объекты можно разделить на два типа:

- теневые объекты, содержат измененные страницы областей памяти, разделяемых в режиме копирования при записи;
- области, заполняемые нулями, такие как стеки, кучи или неинициализированные данные. Они выделяются вызовом

```
vm_allocate (target_task, address, size, flag);
```

Ядро создает менеджер по умолчанию при инициализации системы при вызове memory_object_create(). Управляемые объекты изначально не обладают памятью. При первом обращении к таким объектам ядро вызывает memory_object_data_request(), а менеджер возвращает memory_object_data_unavailable(), что указывает на необходимость заполнения запрашиваемых страниц памяти нулями.

15.5.1. Сетевой сервер разделяемой памяти

Сетевой сервер разделяемой памяти часто используется в качестве примера гибкости интерфейса внешних менеджеров памяти. Сервер представляет собой задачу прикладного уровня, позволяющую разделять набор страниц в своем адресном пространстве клиентскими задачами, физически «расположенными» в любой точке сети. Сервер передает данные клиентам при возникновении страничной ошибки и синхронизирует доступ к разделяемым данным так, чтобы изменения, внесенные одним клиентом, незамедлительно становились видны остальным. При этом для клиентов обращение к совместно используемой памяти выглядит абсолютно одинаковым, независимо от того, разделяется ли область между задачами на одной машине или на нескольких сразу.

Разделение памяти в пределах одной машины является несложным делом. В памяти физически присутсвует лишь одна копия данных, поэтому все изменения, внесенные одним клиентом, незамедлительно попадают в поле зрения остальных. Разделение данных между задачами, выполняемыми на разных компьютерах, намного сложнее, так как на каждой машине могут оказаться собственные копии страниц, что усложняет задачу синхронизации изменений.

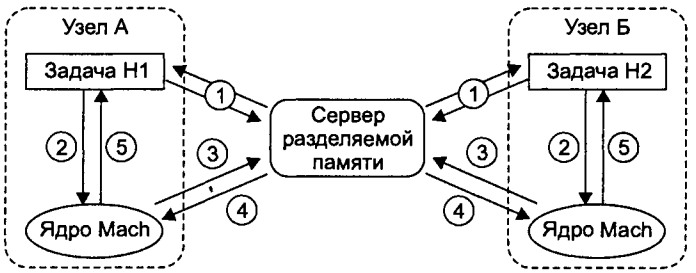


Рис. 15.9. Два клиента осуществляют соединение с сетевым сервером разделяемой памяти

Давайте представим, какие взаимодействия могут происходить между двумя задачами, выполняемыми на двух разных машинах, и разделяющими между собой страницу памяти. Сервер разделяемой памяти вправе находиться как на одном из этих узлов, так и на совершенно другом, третьем компьютере. На рис. 15.9 показана схема первоначального отображения объекта в адресных пространствах клиентов. Для каждой клиентской задачи выполняется последовательность операций, представленная ниже.

1. Клиент запрашивает порт объекта памяти от сервера.
2. Клиент вызывает vm_map для отображения объекта в своем адресном пространстве.
3. Ядро вызывает memory_object_init() для посылки сообщения об инициализации нового объекта серверу. При этом сервер получает право на отправку в управляющий порт, прослушиваемый ядром.
4. Сервер вызывает memory_object_set_attribute() с целью уведомления ядра о готовности объекта.
5. Ядро завершает выполнение системного вызова vm_map и передает управление клиенту.

Пятое действие в принципе может производиться и без ожидания завершения предыдущего, однако при этом клиент не сможет обращаться к данным (все попытки доступа будут заблокированы) до тех пор, пока объект не будет готов. На рис. 15.10 показано, что произойдет в том случае, если задача H1 попытается произвести запись страницы.

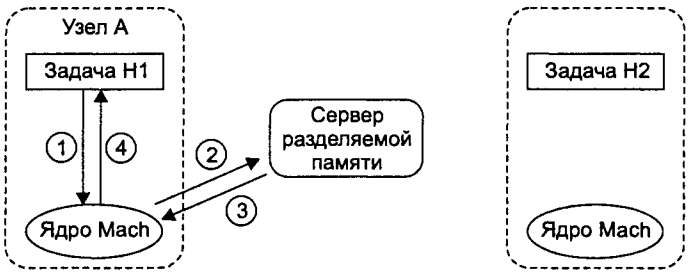


Рис. 15.10. Задача H1 пытается осуществить запись страницы

1. Задача H1 становится причиной возникновения исключения (ошибки записи), что приводит к загрузке ядром обработчика ошибки.
2. Ядро вызывает memory_object_data_request() для передачи запроса серверу на создание копии страницы, доступной в режиме записи.
3. Сервер возвращает страницу через вызов memory_object_data_provided().
4. Ядро вносит изменения в таблицы преобразования адресов задачи H1 и возобновляет ее выполнение.

После этого задача H1 может производить изменения страницы, при этом сервер еще не обладает информацией об вносимых в страницу изменениях. Представим, что произойдет, если в этот момент времени задача H2 осуществит чтение страницы (рис. 15.11).

1. Задача H2 становится причиной возникновения исключения (ошибки чтения), что приводит к загрузке обработчика ядра Б.
2. Ядро Б вызывает memory_object_data_request() для запроса сервера на создание копии страницы, доступной в режиме «только для чтения».
3. Сервер обладает информацией о том, что узел А имеет дубликат этой страницы, доступной для записи. Он вызывает memory_object_lock_request(), указывая в качестве параметров управляющий порт ядра узла А, флаг should_clean и значение переменной lock_value, равное VM_PROT_WRITE[1].
4. Ядро А сужает права доступа к странице до «только для чтения» и записывает обратно изменения, внесенные в нее, на сервер через memory_object_data_write().
5. На этом этапе сервер обладает наиболее свежей версией страницы. Он отправляет ее ядру Б через вызов memory_object_data_provided().
6. Ядро Б вносит изменения в таблицы преобразования адресов и возобновляет выполнение задачи H2.

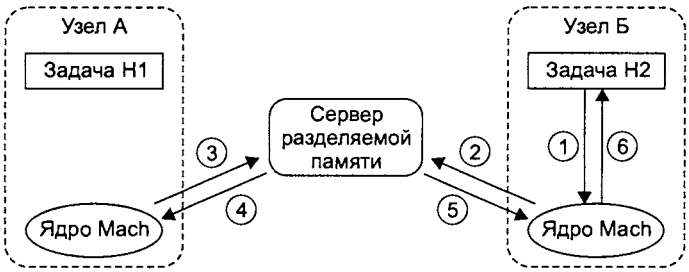


Рис. 15.11. Задача H2 пытается осуществить чтение страницы

Очевидно, что описанное взаимодействие происходит весьма медленно. Если две задачи часто изменяют страницу, каждой из них придется повторять вышеописанные итерации, что приведет к размножению сообщений IPC и необходимости повторного копирования страницы от одного узла сети другому. Большинство описываемых действий являются следствием проблемы необходимости синхронизации данных, передаваемых через сеть. Следует заметить, что каждое исключение приводит к двойному перемещению страницы по сети: сначала от клиента серверу и затем от сервера клиенту, ставшему причиной исключения. Нас может устроить сокращение операций копирова-

[1] Если задача H2 попытается не прочесть, а записать данные в страницу, серверу необходимо будет использовать флаг should_flush для передачи запроса ядру А на сброс атрибута корректности его копии страницы.

ния до одной, если сервер попросит клиента отправить страницу другому клиенту напрямую. Однако такой подход разрушает модульность структуры и влечет усложнение взаимодействий, если число клиентов выше двух. Очевидно, что размещение сервера на одной из клиентских машин поможет избавиться от второй операции копирования и уменьшит трафик сообщений в сети.

Несмотря на перечисленные недостатки, пример показывает, как тесная взаимоувязка технологии обмена сообщениями и подсистемы управления памятью позволяет создавать гибкие и мощные приложения. Еще одной иллюстрацией внешнего менеджера памяти является задача, описанная в [19]. Она управляет выгружаемыми страницами памяти, используемыми приложениями, самостоятельно осуществляющими сбор мусора. Менеджер получает информацию от клиентских задач независимо от того, какая из страниц оказалась выгруженной, и уменьшает число замен страниц путем упреждающего сброса таких страниц. Это позволяет высвободить дополнительный объем памяти для использования его системой.

15.6. Замена страниц

Алгоритм замены страниц в Mach отличается от технологии, реализованной в системах SVR4 и 4.3BSD, где применяется методика «часов с двумя стрелками» (см. раздел 13.5.3). Алгоритм Mach получил название *FIFO с возможностью второй попытки* (FIFO with second chance) [6]. Он использует для работы три списка FIFO («первым вошел — первым вышел»): активных, неактивных и свободных страниц, как это показано на рис. 15.12.

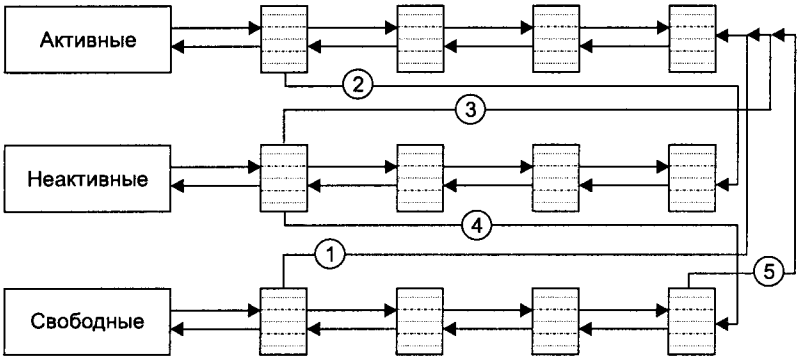


Рис. 15.12. Замещение страниц

Страницы перемещаются из одного списка в другой тогда, когда складывается одна из нижеперечисленных ситуаций.

- Первое обращение к странице является причиной исключения (страничной ошибки). Обработчик удаляет страницу из начала свободного списка и инициализирует ее, заполняя необходимыми данными (нулями, либо данными, полученными из объектов памяти). Затем обработчик

помещает страницу в конец активного списка. Постепенно страница приближается к голове этого списка по мере того, как страницы, находящиеся перед ней, становятся неактивными.

- Если объем свободной памяти в системе падает ниже порогового значения (vm_page_free_min). В этом случае пробуждается демон pagedaemon. Он производит перемещение части страниц из начала активного списка в хвост неактивного списка. Он также сбрасывает для таких страниц бит ссылки в картах аппаратного преобразования адресов.
- Демон pagedaemon проверяет количество страниц в начале неактивного списка. Страницы, чей бит ссылки окажется установленным, возвращаются в конец активного списка.
- Если демон pagedaemon находит страницу, бит ссылки которой остался по-прежнему сброшенным, то он делает вывод о том, что к странице не произошло ни одного обращения за время нахождения в неактивной очереди и такая страница может быть перемещена в конец свободного списка. Если страница оказывается «грязной» (заполненной данными), демон сначала производит их сброс в соответствующий объект памяти.
- Если к странице произошло обращение во время нахождения ее в свободном списке, такая страница восстанавливается, то есть переносится в хвост активного списка. В случае отсутствия попыток доступа страница переместится в начало свободного списка и будет использована повторно по мере надобности.

В системах, где аппаратная архитектура не поддерживает бит ссылки, система Mach имитирует информацию о ссылках путем сброса бита корректности отображений страницы, точно так же, как и в BSD. Более того, в системе имеются несколько параметров, определяющих объем работы, выполняемой демоном pagedaemon при каждом его пробуждении. Эти параметры описаны в табл. 15.1.

Таблица 15.1. Параметры демона pagedaemon ОС Mach

Параметр	Применение параметра	Значение по умолчанию, страницы
vm_page_free_min	Пробуждает демон в том случае, если количество страниц в свободном списке окажется меньше указанного в этом параметре	25 + physmem/100
vm_page_free_target	Демон переносит страницы из неактивного списка в свободный до тех пор, пока тот не достигнет объема, указанного в этом параметре	30 + physmem/80
vm_page_free_reserved	Количество свободных страниц, зарезервированных для демона	15

Параметр	Применение параметра	Значение по умолчанию, страницы
vm_page_ inactive_targed	Демон переносит страницы из активного списка в неактивный до того момента, пока последний список не вырастет до указанного размера	physmem + 2/3

Несмотря на то, что структуры данных и алгоритмы Mach отличаются от метода «часов с двумя стрелками», реализованного в SVR4 и 4.3BSD, их эффект сходен. Неактивный список эквивалентен страницам, расположенным «между» двумя «стрелками» часов. В приведенном выше перечне условий страничной активности второй случай аналогичен функции первой стрелки часов, при прохождении которой осуществляется сброс битов ссылки просматриваемых страниц. Функция второй стрелки подобна действиям в третьем и четвертом вариантах, то есть проверке страниц в начале неактивной очереди.

Основным различием методик являются правила выбора демоном pagedaemon страниц для переноса из активной в неактивную очередь. В алгоритме часов страницы просматриваются последовательно, исходя из их расположения в физической памяти, а не на основе определенных правил. В системе Mach активный список также является и FIFO, следовательно, демон выбирает те страницы, обращение к которым происходило наиболее давно. Это свойство привносит некоторые преимущества по сравнению с алгоритмом часов.

Система Mach не поддерживает альтернативных правил замещения страниц. Некоторые приложения, такие как системы управления базами данных, не демонстрируют локальность ссылок. Для них принцип LRU может оказаться неприемлемым. В [13] описывается простое расширение интерфейса внешних менеджеров памяти, позволяющее задачам прикладного уровня выбирать собственные правила замены страниц.

15.7. Анализ

Операционная система Mach имеет весьма развитую архитектуру VM, большинство средств Mach совпадает с аналогичными возможностями SVR4. Это — разделение памяти в режиме копирования при записи, отображение файлов и поддержка адресных пространств большой величины. Точно так же как и система SVR4, подсистема памяти в ОС Mach основана на объектно-ориентированной модели и использует небольшой набор объектов, представляющих модульный программный интерфейс. В Mach четко разграничены аппаратно-зависимые и аппаратно-независимые фрагменты кода. Все коды, обусловленные конкретной аппаратной архитектурой, вынесены на отдельный уровень pmap, доступный при помощи четко определенного, тщательно продуманного

процедурного интерфейса. При необходимости переноса системы на новую аппаратную платформу достаточно переписать заново только уровень pmap.

Подсистема VM Mach обладает и некоторыми средствами, не имеющими аналогов в SVR4. Система поддерживает более гибкие средства разделения памяти, что достигнуто путем обособления понятий разделения (совместного использования) и наследования. Подсистема управления памятью тесно интегрирована со средствами взаимодействия процессов. Подсистема VM используется IPC для организации эффективного обмена сообщениями большого размера (иногда равного целому адресному пространству задачи) на основе технологии копирования при записи. В то же время подсистема VM задействует IPC с целью обеспечения независимости месторасположения своих объектов и расширения собственных возможностей для работы в распределенных средах. В частности, задачи прикладного уровня могут управлять резервным хранением объектов памяти, при этом ядро осуществляет взаимодействие с такими задачами через сообщения IPC. VM и IPC составляют вместе гибко расширяемую среду, обеспечивающую сосуществование нескольких внешних менеджеров памяти (прикладного уровня), которые предоставляют различные способы управления страницами. Прекрасным примером использования интерфейса взаимодействия двух технологий является сетевой менеджер разделяемой памяти.

Однако подсистема виртуальной памяти Mach обладает и некоторыми ограничениями, большинство из которых сходно с недостатками SVR4. Подсистема VM больше, медленнее и сложнее по сравнению со структурой аналогичной подсистемы BSD. Она оперирует большим количеством структур данных, превышающих размеры аналогичных структур BSD. Следовательно, подсистеме VM Mach необходимо большее количество памяти для своих собственных нужд, что сокращает тем самым объем памяти, доступный процессам. Так как разработчики системы постарались свести количество машинно-зависимых кодов к минимуму, они не могут быть оптимизированы для какой-то определенной архитектуры MMU.

Применение технологии обмена сообщениями становится причиной дополнительной перегрузки системы. В некоторых случаях нагруженность можно уменьшить путем оптимизации обмена сообщениями внутри ядра, но в целом эта технология остается менее производительной по сравнению с обычными вызовами функций. Внешние менеджеры памяти не получили широкого распространения, кроме сетевого менеджера разделяемой памяти. Применение внешних менеджеров стало причиной логичного вопроса: являются ли такие менеджеры настолько удобными, чтобы оправдать свою высокую стоимость? Коммерческая операционная система Digital UNIX, основанная на Mach, не поддерживает внешних менеджеров памяти и не экспортирует интерфейс Mach VM. Подсистема виртуальной памяти в Digital UNIX отличается от VM Mach по многим параметрам.

15.8. Управление памятью в 4.4BSD

В разделе 13.4 уже говорилось о модели управления памятью системы 4.3BSD. Описанная структура была эффективна на машинах, на которых она выполнялась. Во времена разработки 4.3BSD типичный компьютер представлял собой централизованную систему разделения времени, к которой обращались сразу несколько пользователей через терминалы, подключаемые к машине по последовательным линиям связи. Машина, как правило, имела диски большого по тем меркам объема (несколько сотен мегабайтов), медленный процессор (1–2 MIPS), а также малый объем памяти (которая была очень дорогой — объем памяти, равный 4 мегабайтам, считался огромным). Несмотря на то, что UNIX уже поддерживала сети, удаленные файловые системы еще не завоевали в те времена популярность, поэтому большинство систем использовало для файловых систем и пространства свопинга локальные диски. Подсистема виртуальной памяти 4.3BSD оптимизирована на экономию оперативной памяти за счет дополнительного использования внешних носителей (или операций ввода-вывода).

В начале 90-х годов, когда разрабатывалась система 4.4BSD, ситуация на рынке вычислительной техники разительно изменилась. Теперь каждый пользователь стал работать за собственным компьютером, обладающим большим объемом оперативной памяти (32 Мбайт уже не считалось редкостью) и быстрым процессором (имеющим производительность, равную десяткам MIPS). С другой стороны, появление сетевых файловых систем, таких как NFS, позволило пользователям хранить свои файлы на централизованных файл-серверах. Рабочие станции имели диски малого объема, либо не имели их совсем. Для таких сред структура управления памятью 4.3BSD оказалась совершенно неприемлемой, поэтому разработчики 4.4BSD заменили ее новой моделью [12].

Внутренняя базовая структура VM системы 4.4BSD схожа с аналогичной структурой ОС Mach. Однако ее внешний интерфейс больше напоминает таковой в SVR4. Прежде чем ассимилировать понятия внешних обработчиков памяти и объектов памяти, создатели 4.4BSD предложили поддержку системного вызова mmap, синтаксис которого оказался родственным с аналогичным вызовом SVR4 (см. раздел 14.2):

```
paddr = mmap (addr, len, prot, flags, fd, off);
```

Вызов осуществляет отображение диапазона адресов [paddr, paddr+len) процесса в байтовую область [off, off+len) файла, представленного дескриптором fd[1]. Также как и в SVR4, addr является адресом, по которому необходимо отобразить файл, а параметр prot характеризует защиту (составляется из комбинации PROT_READ, PROT_WRITE и PROT_EXECUTE). Флаги MAP_SHARED, MAP_PRIVATE и MAP_FIXED имеют те же значения, что и их аналоги в SVR4.

[1] Мы по-прежнему пользуемся стандартными обозначениями скобок. Квадратная скобка означает, что число включается в диапазон возможных значений, круглая — значение не входит в этот диапазон.

Системный вызов mmap 4.4BSD обладает несколькими дополнительными возможностями по сравнению с предыдущей реализацией. Параметр вызова flags должен быть равным либо MAP_FILE (отображение файла или устройства), либо MAP_ANON (отображение в анонимной памяти). Система поддерживает также два дополнительных флага — MAP_INHERIT (указывает на необходимость продолжения поддержки отображения после выполнения вызова exec) и MAP_HASSEMAPHORE (признак того, что область памяти может содержать семафор).

Процессы могут разделять память между собой двумя различными способами. Один и тот же файл может быть отображен в адресных пространствах нескольких процессов, при этом файл определяет изначальное содержимое образа и место резервного хранения области памяти. Процесс также имеет право отобразить файл на анонимную область памяти, ассоциировать с ней дескриптор файла и затем передать его другим процессам, желающим присоединить эту область памяти. Такой подход позволяет избавиться от лишних операций с отображением файла, при этом файловый дескриптор используется только как имя области.

Система 4.4BSD поддерживает быструю синхронизацию процессов путем помещения семафоров в разделяемую область памяти. В традиционных вариантах UNIX процессы применяют семафоры с целью синхронизации доступа к разделяемой памяти (см. раздел 6.3). Для управления семафорами требуется вызов системных функций, что является основной причиной перегрузок и сводит на нет преимущества разделяемой памяти. В системе 4.4BSD такие перегрузки были уменьшены путем помещения семафоров в области разделяемой памяти.

Практика показала, что большинство приложений, использующих синхронизацию, при попытке процесса получить разделяемый ресурс в большинстве случаев находят его незаблокированным. Если взаимодействующие процессы будут помещать семафоры прямо в разделяемые области памяти, то смогут работать с ними без вызова системных функций (неделимых операций *проверки-установки* (test-and-set) или их эквивалентов, см. раздел 7.3.2). Процессу необходимо вызывать системные функции только тогда, когда ресурс окажется заблокированным. Тогда ядро заново проверит семафор и заблокирует выполнение процесса только в том случае, если семафор все еще будет занят. Процесс, освобождающий ресурс, может сделать это на прикладном уровне. После открепления он проверяет наличие других процессов, ожидающих тот же ресурс, и в случае их обнаружения производит системный вызов с целью пробуждения таких процессов.

Система 4.4BSD предлагает интерфейс взаимодействия для управления семафорами, который будет показан ниже. Область разделяемой памяти, содержащая семафор, должна создаваться с указанием флага MAP_HASSEMAPHORE. Получение семафора процессом осуществляется при помощи

```
value = mset (sem, wait);
```

где sem — указатель на семафор, wait — логическая переменная, установленная в значение true, если вызывающий функцию процесс желает приостановить свое выполнение по семафору. Системный вызов mset возвращает в переменной value значение 0 при условии, что процесс успешно захватил семафор. Для освобождения семафора процесс-владелец должен воспользоваться вызовом

```
mclear (sem);
```

В системах, поддерживающих неделимую инструкцию проверки-установки, вызовы mset и mclear могут быть реализованы на прикладном уровне. Система 4.4BSD предлагает для блокировки и разблокирования процессов на основе состояния семафора следующие вызовы:

```
msleep (sem);
```

проверяет семафор и блокирует вызвавший процесс, если sem окажется захвачен;

```
mwakeup (sem);
```

пробуждает, по крайней мере, один процесс, заблокированный по семафору, либо не производит никаких действий, если не обнаруживает ни одного приостановленного процесса.

Интерфейс семафоров в System V IPC (см. раздел 6.3) существенно отличается от представленного в этом разделе. ОС System V поддерживает управление сразу же несколькими семафорами посредством одного системного вызова. Гарантом неделимости всех операций с набором семафоров внутри системного вызова выступает ядро. Такой интерфейс может быть реализован в 4.4BSD путем ассоциирования одиночного *защитного* семафора (guardian) с каждым таким набором. При выполнении любой операции над семафором процесс должен в первую очередь стать владельцем семафора-«опекуна». Затем процесс проверяет искомый семафор на предмет возможности продолжения необходимых для себя операций. Если такое невозможно, процесс освобождает защитный семафор и блокируется по одному из недоступных семафоров набора. После пробуждения процессу следует заново повторить все вышеперечисленные действия.

15.9. Корректность буфера ассоциативной трансляции (TLB)

Трафик обмена памятью может стать чрезмерным, если процесс при каждом преобразования адреса не будет обладать доступом к таблицам страниц, размещаемым в памяти компьютера. Из этого можно сделать вывод, что диспетчеры MMU должны содержать некий буфер опережающей выборки, являющийся кэшем последних преобразований адресов. Буфер ассоциативной

трансляции (TLB) реализуется в машине аппаратно и, как правило, содержит небольшое количество ячеек (от 64 до 256), каждая из которых отображает виртуальную страницу памяти в физическую. Диспетчер MMU проверяет кэш во время выполнения каждой операции преобразования адреса. Если необходимое преобразование окажется в кэше (кэш-попадание), операции не потребуется производить просмотр таблиц преобразования, что ускоряет скорость доступа к памяти.

Кэш TLB является ассоциативным. Это означает, что MMU производит последовательный поиск во всех вхождениях кэша, пытаясь обнаружить совпадение с данным виртуальным адресом. Вхождение содержит номер соответствующей физической страницы, а также информацию о ее защите. Формат вхождения MMU зависит от конкретной аппаратной составляющей. Семантика доступа к буферу TLB в архитектурах MIPS R3000, IBM RS6000 и Intel 80x86 была разобрана в разделе 13.3.

Буфер TLB поддерживает две основные операции для вхождений: инициализацию (load) или сброс (invalidate или purge). Вхождение TLB инициализируется в случае его отсутствия в кэше (кэш-промах). В большинстве архитектур такая операция производится на аппаратном уровне. Машина обладает знаниями о местонахождении и формате карт вторичного преобразования адресов (таких как таблицы страниц большинства систем или инвертированные таблицы страниц, поддерживаемые IBM RS6000). Диспетчер MMU находит вхождение для страницы в таблице страниц и загружает его во вхождение TLB перед завершением преобразования адреса. Ядро системы не принимает никакого участия в этих действиях и даже не знает о самом факте их проведения.

В некоторых архитектурах, например в MIPS R3000 [9], поддерживается программный перезапуск (software reload). В таких системах MMU знает о присутствии только TLB. Если блок не находит необходимое вхождение в TLB, генерируется исключение. Картами преобразования адресов управляет ядро, а аппаратная часть не знает об их существовании и не определяет их структуру. Ядро, как правило, использует для этой цели обычные таблицы страниц, но это не является обязательным требованием. После генерации диспетчером MMU ошибки об отсутствии необходимого вхождения TLB ядро находит преобразование и принудительно загружает вхождение TLB, содержащее корректную информацию.

Все блоки MMU поддерживают возможность программного сброса корректности вхождений TLB. Ядро может вносить изменения в элемент TLB сразу по нескольким причинам, например, при потребности установить другие атрибуты защиты или сбросить атрибут корректности страницы после ее выгрузки в область свопинга. Такие действия ядра способны привести к недействительности информации, содержащейся во вхождении TLB. Это вхождение

должно быть удалено из кэша (либо объявлено недействительным). Ядро может сделать это тремя различными способами:

- сбросить корректность одного вхождения TLB, идентифицируемого по виртуальному адресу. Если буфер не имеет соответствующего вхождения, то ядро не выполняет никаких действий. Примером такой методики управления буфером является инструкция TBIS (*Translation Buffer Invalidate Single*, сброс корректности одного вхождения буфера преобразований) VAX-11;
- объявить недействительным все содержимое кэша. Например, архитектура Intel 80x86 поддерживает сброс содержимого в любой момент времени путем записи в регистр PDBR (*Page Directory Base Register*, базовый регистр каталога страниц). Такая запись может быть выполнена принудительно командой move или косвенно при переключении контекста. Архитектура VAX-11 поддерживает аналогичную инструкцию TBIA (*Translation Buffer Invalidate All*, сброс корректности всех вхождений буфера преобразований), осуществляющую те же действия;
- загрузить новое вхождение TLB, перезаписав при этом предыдущую информацию для данного адреса, если таковая находилась в буфере ранее. Этот метод используется в таких архитектурах, как MIPS R3000, позволяющих программно перезагружать вхождения TLB.

15.9.1. Корректность TLB в однопроцессорных системах

Управление буфером TLB в однопроцессорных системах является несложной задачей. В системе происходит определенное число событий, наступление которых приводит к изменению корректности информации одного или нескольких вхождений TLB. Некоторые такие события перечислены ниже.

- **Protection change** (изменение атрибутов защиты страницы). Процесс может вызывать mprotect с целью снижения или повышения степени защиты заданного диапазона адресов. Изменять защиту страниц вправе также и ядро системы, например, при имитации битов ссылки или в случае копирования при записи.
- **Pageout** (выгрузка страницы из памяти). При удалении страницы из физической памяти ядро системы должно произвести сброс корректности всех вхождений таблиц страниц и буфера TLB, имеющих ссылки на страницу.
- **Context switch** (переключение контекста). Если ядро производит переключение на выполнение нового процесса, все элементы TLB предыдущего процесса становятся недействительными. При этом вхождения, отображающие адреса ядра, остаются по-прежнему корректными, так

как ядро разделяется всеми процессами системы. Некоторые архитектуры поддерживают самоидентифицируемые (tagged) TLB, в которых каждое вхождение сопровождается тегом, указывающим, к какому процессу оно относится. В таких системах при переключении контекста не происходит сброса корректности вхождений TLB, поскольку все вхождения нового процесса имеют уже другой тег.

- **Выполнение** exec. Если процесс вызывает exec для выполнения новой программы, все элементы TLB, соответствующие старому адресному пространству, становятся недействительными. В этом случае те же виртуальные адреса ссылаются на страницы уже нового отображения.

Сброс элементов TLB является весьма трудоемкой операцией, поэтому ядро должно по возможности оптимизировать выполнение поставленной задачи. При этом важно определить, насколько недостоверность информации может оказаться безвредной для системы. Поясним на примере. Представьте, что ядро производит изменение атрибута защиты страницы от «только для чтения» до «чтения-записи». После этой операции информация в соответствующем вхождении TLB не совпадает с данными элемента таблицы страниц, однако все еще нет необходимости сбрасывать корректность такого вхождения. В худшем случае процесс попытается произвести запись в страницу и обнаружит, что вхождение TLB этой страницы показывает, что она доступна только для чтения. В этот момент времени обработчик ошибки (или аппаратная часть системы) может загрузить новое вхождение TLB, обладающее правильной информацией. Отсрочка выполнения операции загрузки данных во вхождение TLB чревата тем, что через некоторое время вообще не понадобится производить эти действия. Например, вхождение TLB может оказаться сброшенным при возникновении такого события, как переключение контекста.

При работе с несколькими страницами ядро вправе выбирать, сбросить ли весь буфер TLB или выполнить действия по отношению к индивидуальным вхождениям. Сброс корректности буфера целиком является более быстрым, так как осуществим одной операцией. Однако такой подход может привести к большему числу последующих промахов, что приводит к увеличению длительности дальнейших операций (системе придется потратить время на загрузку отсутствующих вхождений). Оптимальное решение, как правило, зависит от количества сбрасываемых вхождений.

15.9.2. Корректность TLB в многопроцессорных системах

Поддержка актуальности информации TLB в многопроцессорных системах разделения времени является намного более сложной проблемой. Несмотря на то, что все процессоры системы разделяют между собой память машины, каждый из них имеет свой собственный буфер TLB. Определенная трудность

возникает в том случае, если один из процессоров изменяет вхождение таблицы страниц, которое используется в это же время другим процессором. Такой процессор может обладать копией вхождения в своем буфере TLB и продолжать обращаться к уже устаревшим данным. Становится очевидным, что необходимо передавать изменения TLB всем процессорам системы, которые осуществляют обработку страниц.

На некоторых машинах синхронизация буферов TLB реализована аппаратно. Например, инструкция системы ipte IBM/370 производит неделимую операцию изменения вхождения таблицы страниц и сброса его корректности во всех буферах TLB системы. Однако чаще аппаратура не поддерживает автоматическую синхронизацию буферов TLB. Большинство систем не позволяют процессору даже выполнить сброс корректности вхождений буфера TLB другого процессора.

В системе может возникать множество ситуаций, при которых изменение одной страницы должно влиять на данные буферов TLB сразу нескольких процессоров:

- страница принадлежит ядру;
- страница разделяется между несколькими процессами, каждый из которых выполняется на отдельном процессоре машины;
- в многонитевых системах, в которых несколько нитей одного процесса выполняются одновременно на нескольких процессорах машины. Если одна из нитей вносит изменения в отображение, то обновленные данные должны быть доступны всем остальным нитям.

В системе также вероятна ситуация, когда один процессор производит изменения в адресном пространстве процесса, выполняющегося на другом CPU. В этом случае происходит перезагрузка вхождений лишь в одном буфере TLB, но не принадлежащем процессу, ставшему инициатором этих изменений.

Если аппаратная часть не поддерживает синхронизацию, то решать возникающие проблемы приходится ядру на программном уровне. Для этого используется механизм уведомлений, основанный на межпроцессорных прерываниях. Здесь нам необходимо дать определение двух терминов: *инициатора* (initiator) и *исполнителя* (responder). Инициатором называется процессор, который изменяет отображение и тем самым делает неактуальным некоторое количество удаленных буферов TLB. Исполнитель — это удаленный процессор, который, возможно, содержит вхождение TLB для такого отображения. Инициатор отправляет прерывание исполнителю, который, в свою очередь, сбрасывает вхождения соответствующего буфера TLB.

Возникающая ситуация является весьма сложной, так как для ее решения необходимо произвести по крайней мере два действия: внести изменения во вхождение таблицы страниц и сбросить корректность элемента TLB. Представьте, что исполнитель успевает сбросить вхождение TLB до того, как инициатор внесет изменения в PTE. В этом случае процесс, выполняющийся на

процессоре-исполнителе, может произвести попытку обращения к странице между означенными двумя действиями, что будет иметь следствием аппаратную перезагрузку некорректного вхождения TLB. Изменение порядка следования операций на обратный также приводит к возникновению проблем. Если инициатор будет производить изменение PTE в первую очередь, исполнитель получит возможность записать устаревшее вхождение TLB в таблицу страниц с целью модификации битов ссылки или изменения.

Для того, чтобы оба действия производились согласованно, исполнитель обязан ожидать (находиться в ждущем цикле) завершения изменения PTE и лишь после этого сделать недействительным вхождение своего буфера TLB. В следующем разделе будет рассмотрен алгоритм синхронизации TLB, реализованный в операционной системе Mach.

15.10. Алгоритм синхронизации TLB системы Mach

Алгоритм синхронизации TLB операционной системы Mach носит название *перезагрузки TLB* (TLB shootdown) [2] и является сложным набором взаимодействий между инициатором и исполнителем. Термин «перезагрузка» означает сброс вхождений TLB на другом процессоре. Для реализации этого метода в системе Mach используются следующие структуры данных (для каждого процессора):

- атрибут active (текущий). Показывает, использует ли процессор какую-либо таблицу страниц. Если атрибут сброшен, процессор принимает участие в перезагрузке и не может обращаться к любому изменяемому вхождению pmap (pmap представляет собой аппаратную карту преобразования адресов задачи и, как правило, состоит из таблиц страниц);
- очередь запросов на сброс корректности. В каждом таком запросе указывается отображение, которое должно быть выключено;
- набор структур pmap, активных в текущий момент. Обычно каждый процессор системы имеет две активных структуры pmap: ядра и текущей задачи.

Любая структура pmap защищается при помощи простой блокировки (spin lock), из-за чего все операции над ней производятся последовательно. Каждая структура pmap также поддерживает список процессоров, на которых она активна в текущий момент.

Ядро применяет технику перезагрузки в том случае, если один из процессоров системы производит изменения в преобразованиях адресов, которые могут привести к недостоверности вхождений TLB на других процессорах. На рис. 15.13 показана работа алгоритма при наличии одного исполнителя.

Инициатор в первую очередь запрещает все прерывания и сбрасывает собственный флаг активности. Далее он блокирует структуру pmap и отправляет запросы на очистку TLB каждому процессору, где активна та же структура pmap. Затем инициатор пересылает таким процессорам межпроцессорные прерывания и ждет от них подтверждения.

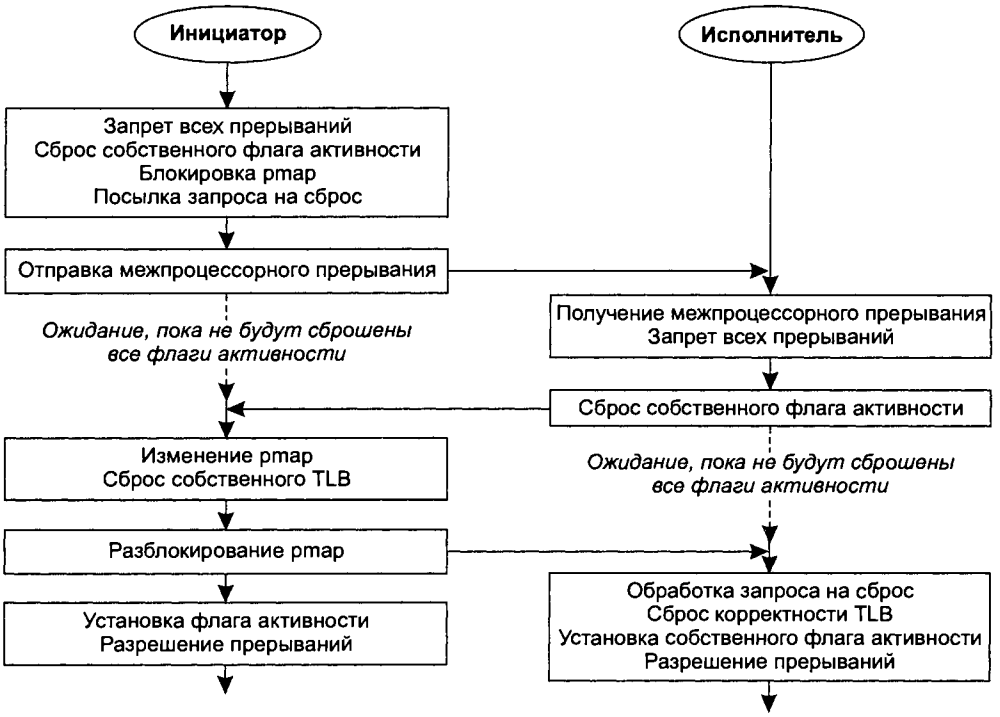


Рис. 15.13. Алгоритм перезагрузки буферов TLB в системе Mach

По получении прерывания исполнитель также запрещает обработку всех прерываний. Затем он подтверждает факт приема прерывания посредством сброса индикатора активности и переходит в режим ждущего цикла, условием выхода из которого является разблокирование структуры pmap инициатором. В это время инициатор ожидает, пока все процессоры, разделяющие отображение, станут неактивными. После того, как каждый из них подтвердит получение прерывания, инициатор сбросит собственное вхождение буфера TLB, изменит структуру pmap и затем сделает ее доступной другим процессорам. На этот момент все исполнители могут выйти из цикла ожидания, начать обработку своих очередей запросов и объявить недействительными все устаревшие вхождения TLB. На последнем этапе инициатор и исполнители изменяют значения флагов активности, разрешают прерывания и возобновляют работу в обычном режиме.

15.10.1. Синхронизация и предупреждение взаимоблокировок

Метод перезагрузки использует несколько механизмов синхронизации. Порядок следования операций является важным для его правильной работы [18]. Обработка прерываний устройств может привести к длительным простоям процессоров, поэтому важно запретить выполнение прерываний во время выполнения алгоритма. Блокировка pmap пресекает одновременную инициализацию перезагрузки TLB по отношению к одной и той же структуре pmap. Прерывания необходимо запрещать до блокировки таблицы страниц, так как захват межпроцессорного прерывания во время блокировки может привести к взаимоблокировке (deadlock) процессора.

Перед блокировкой pmap инициатор сбрасывает собственный индикатор активности с целью защиты от взаимной блокировки. Представим, что два процессора, П1 и П2, одновременно пытаются внести изменения в одну и ту же структуру pmap. Процессор П1 запрещает прерывания, блокирует pmap и посылает процессору П2 прерывание. В это время процессор П2 также налагает запрет на прерывания и блокируется по тому же объекту. Это приводит к тупиковой ситуации, так как процесс П1 находится в режиме ожидания подтверждения прибытия прерывания от П2, в то время как П2 дожидается освобождения pmap.

Сброс флага активности является эффективным способом подтверждения прерываний. В нашем примере П1 не заблокируется, поскольку процессор П2 сбросил свой флаг активности до попытки блокировки pmap. После того, как П1 освободит pmap, процессор П2 продолжит выполнение и обработает сообщение о необходимости сброса TLB, отправленное ему процессором П1.

Эффект от применения алгоритма перезагрузки далеко не всегда очевиден по отношению ко всем блокируемым ресурсам. Этот метод требует соблюдения четкой политики запрета определенных прерываний при захвате объекта блокировки. Представьте, что процессор П1 удерживает ресурс, разрешая при этом обработку прерываний. Процессор П2 пытается захватить тот же ресурс, при этом прерывания на нем запрещены. Процессор П3 инициализирует выполнение алгоритма перезагрузки, а процессоры П1 и П2 выступают в роли исполнителей. Процессор П3 посылает П1 и П2 межпроцессорные прерывания и затем блокируется до тех пор, пока не получит подтверждения. Процессор П1 подтверждает получение и затем блокируется в ожидании освобождения pmap. Однако процессор П2 уже заблокирован с запрещением прерываний и вследствие этого не может увидеть или подтвердить факт приема. В результате возникает трехстороннее взаимоисключение. Для предупреждения подобных ситуаций система должна установить фиксированное состояние прерываний для каждого случая блокировки: блокировка может быть получена в собственность при «всегда» отключенных прерываниях, либо, наоборот, при «всегда» обрабатываемых прерываниях.

15.10.2. Некоторые итоги

Алгоритм перезагрузки TLB системы Mach помогает решить весьма сложную проблему без вовлечения каких-либо дополнительных аппаратных средств (кроме поддержки межпроцессорных прерываний). С другой стороны, эта техника является весьма затратной и плохо масштабируемой. Все исполнители должны находиться в состоянии ожидания до тех пор, пока инициатор производит изменение pmap. В больших системах, имеющих десятки и сотни процессоров, операция перезагрузки TLB может привести к простою по крайней мере нескольких CPU.

Представленный алгоритм является весьма сложным, но это необходимо сразу же по двум причинам. Во-первых, некоторые MMU автоматически записывают данные о вхождениях TLB в таблицы страниц при модификации битов изменения и ссылки. Такая модификация приводит к перезаписи вхождения pmap целиком. Во-вторых, программные и аппаратные размеры страниц, как правило, различны. Ядру при изменении отображения одной страницы иногда приходится производить обновление нескольких вхождений pmap. Такие изменения должны быть неделимы для всех процессоров системы. Единственным способом достижения эффекта неделимости является приостановка работы всех процессоров, использующих pmap, на время перегрузки вхождения.

На сегодняшний день существует множество других способов перезагрузки TLB. В следующем разделе вы увидите описание некоторых специализированных (ad hoc) решений, позволяющих уменьшить частоту операций сброса корректности TLB. Отдельные методы зависят от конкретных аппаратных характеристик машины, что упрощает решение проблемы. Например, в [18] описывается алгоритм, эффективно работающий на RP3 (IBM Research Parallel Processor Prototype, [16]). Этот алгоритм основан на том, что процессор RP3 не производит автоматической отложенной записи содержимого вхождений в оперативную память, а также использует тот факт, что большой размер физической страницы (16 Кбайт) избавляет от необходимости программной странице занимать сразу несколько физических страниц памяти. Другие разработки требуют изменения архитектуры диспетчера MMU с целью вовлечения аппаратной составляющей в процедуру перезагрузки TLB [21].

15.11. Корректность TLB в SVR4 и SVR4.2

Группа компьютерных компаний, объединившаяся в Intel Multiprocessor Consortium, совместно разработала ОС SVR4/MP, вариант системы SVR4 для многопроцессорных машин, оборудованных CPU семейства Intel. Через некоторое время в UNIX Systems Laboratories создали SVR4.2/MP, версию операционной системы SVR4.2, поддерживающую многопроцессорную обработку,

а также другие возможности (например, легковесные процессы). В этом разделе мы рассмотрим аспекты корректности буферов TLB в обеих реализациях.

15.11.1. SVR4/MP

Алгоритм перезагрузки TLB системы Mach предлагал решение общей проблемы синхронизации буферов TLB. Разработчики попытались добиться максимальной эффективности его функционирования, которую можно достичь без изменения аппаратных характеристик систем (единственным требованием методики является поддержка межпроцессорных прерываний) или причин наступления события, следствием которого является необходимость перезагрузки TLB. В системе SVR4/MP применяется несколько другой подход к решению проблемы. Система пытается анализировать событие, обусловившее недействительность TLB, и найти наиболее удачный метод его обработки. В этой ОС существуют четыре типа событий, требующих синхронизации буферов TLB:

- процесс изменяет размер своего адресного пространства путем вызова brk (или sbrk), либо при освобождении области памяти;
- демон pagedaemon сбрасывает корректность страницы или с целью ее освобождения, или для имитации битов ссылки;
- ядро отображает системный виртуальный адрес в другую физическую страницу;
- процесс пытается произвести запись в страницу, разделяемую по методу копирования при записи.

В системе SVR4/MP аппаратура машины автоматически сбрасывает TLB при каждом переключении контекста [15]. Следовательно, случай первого типа событий не представляет никакой проблемы, поскольку система не поддерживает многонитевые процессы. Операционная система SVR4/MP обладает средствами оптимизации случаев 2 и 3.

Для уменьшения числа одиночных сбросов вхождений TLB во втором случае демон pagedaemon накапливает несколько таких заданий и производит их одной операцией. Это позволяет уменьшить затраты, связанные со сбросом каждого элемента в отдельности.

Основным «поставщиком» операций сброса корректности TLB в SVR4 является драйвер seg_map, используемый при вызовах read или write. Ядро реализует эти вызовы при помощи создания отображения файла в собственном адресном пространстве и последующего копирования данных прикладному процессу. Ядро управляет сегментом seg_map, позволяющим динамически подключать и отключать страницы файлов в адресном пространстве. Это приводит к частому изменению физических отображений виртуальных адресов сегмента. Так как ядро разделяется между всеми процессами системы,

именно оно ответственно за корректность отображений. Процессы не должны обращаться к уже устаревшим отображениям, хранящимся в TLB других процессоров системы.

Для отслеживания устаревших вхождений TLB ядро системы SVR4/MP поддерживает глобальный счетчик (generation count), а также набор локальных счетчиков (по одному на каждый процессор). После сброса процессором информации буфера TLB происходит увеличение глобального счетчика и копирование его нового значения в локальный счетчик. При освобождении отображения страницы драйвером seg_map ядро передает адресу значение глобального счетчика. Если драйвер отображает по этому адресу новую физическую страницу, ядро сравнивает сохраненное содержимое счетчика со значением локального счетчика каждого процессора. Если величина локального счетчика окажется ниже по сравнению с сохраненным значением, можно сделать вывод, что буфер TLB содержит устаревшие данные. В этом случае ядро объявляет недействительным весь буфер TLB. В противоположном случае можно сделать вывод о том, что все процессоры обновили информацию TLB после сброса корректности адреса, таким образом, для рассматриваемой страницы не осталось устаревших отображений.

Разработчики SVR4/MP в целях оптимального решения проблемы исходили из того, что после сброса корректности seg_map ядро не сможет обратиться к соответствующему адресу до тех пор, пока по нему не будет создано отображение нового физического адреса. С целью минимизации операций сброса корректности TLB страницы, освобождаемые seg_map повторно используются в порядке FIFO. Это позволяет удлинить промежуток времени между освобождением адреса и созданием нового отображения, что также увеличивает и вероятность сброса буферов TLB другими процессами за этот период.

15.11.2. SVR4.2/MP

Операционная система SVR4.2/MP представляет собой многонитевую версию SVR4.2, поддерживающую многопроцессорную обработку. Правила перезагрузки TLB и их реализация в системе обладают определенными сходствами со средствами SVR4/MP, описанными в разделе 15.11.1, однако они отличаются от предыдущей версии и некоторыми важными дополнениями. Все взаимодействия с буфером TLB перенесены в системе на аппаратно-зависимый уровень HAT (см. раздел 14.4). При разработке SVR4.2/MP основной задачей был перенос ОС на архитектуру Intel 386/486, однако алгоритмы поддержания корректности TLB и интерфейсы взаимодействия с ними легко портируются и на другие аппаратные платформы.

Как и в системе SVR4/MP, в новой реализации ОС ядро полностью управляет собственным адресным пространством и, следовательно, всецело отвечает за невозможность доступа к недействительным отображениям ядра (например, освобожденным сегментом seg_map). Отсюда следует, что ядро

может применять правила отложенной перезагрузки вхождений TLB, отображающих адреса ядра. Однако система SVR4.2/MP поддерживает легковесные процессы (или LWP, см. раздел 3.2.2). Несколько LWP одного и того же процесса могут выполняться в этой ОС одновременно на нескольких процессорах машины. Это говорит о том, что ядро не в состоянии манипулировать правилами обращений к памяти прикладных процессов и должно применять принцип немедленной перезагрузки потерявших актуальность вхождений TLB, принадлежащих прикладным процессам.

Средства глобальной перезагрузки TLB, заложенные в SVR4/MP, не могут эффективно работать в системах с большим количеством процессоров, так как их применение приводит к простою всей системы во время операции перезагрузки. В SVR4.2/MP поддерживается список процессоров для каждой структуры hat. Структура hat адресного пространства ядра включает в себя список всех работоспособных процессоров машины, следовательно, ядро потенциально может выполняться на любом из них. Структура hat прикладного процесса содержит список только тех процессоров, на которых этот процесс активен (то есть выполняется один из LWP этого процесса). Доступ к списку осуществляется при помощи интерфейсных функций, представленных ниже:

- hat_online() и hat_offline() — операции предназначены для добавления или удаления процессоров из списка структуры hat ядра;
- hat_asload(as) — добавляет процессор в список структуры hat адресного пространства as и загружает это адресное пространство в MMU;
- hat_asunload(as, flags) — выгружает отображения процесса и удаляет процессор из списка адресного пространства as. Параметр flags поддерживает единственный флаг, указывающий на необходимость сброса информации из локального буфера TLB после выгрузки отображения[1].

Ядро системы также использует объект под названием *cookie*, видимый только на уровне HAT. Этот объект может быть реализован как временная метка (в формате timestamp), либо как счетчик (порт ссылок использует временную метку). Объект должен удовлетворять следующему условию: значение нового объекта cookie должно быть всегда выше предыдущего. Для получения нового объекта cookie используется процедура hat_getshootcookie(), возвращающая значение, равное «возрасту» TLB. Ядро передает cookie процедуре hat_shootdown(), ответственной за перезагрузку вхождения ядра в буферах TLB. Если на каком-то процессоре остается прежний объект cookie, это означает, что информация буфера TLB может быть устаревшей и должна быть сброшена.

В следующих разделах будут обсуждаться алгоритмы отложенной и немедленной перезагрузки TLB.

---

[1] Реализация переключения контекста в архитектуре Intel не запрашивает ask_asunload() для сброса TLB, так как содержимое буфера будет сброшено в любом случае при подключении новой области u.

15.11.3. Отложенная перезагрузка

Как и в SVR4/MP, ядро применяет алгоритм отложенной перезагрузки для страниц драйвера seg_map. В ОС SVR4.2/MP это правило эффективно также и по отношению к драйверу seg_kmem, отвечающему за динамическую память ядра. Если драйверы seg_map или seg_kmem сбрасывают атрибут корректности страницы, операция перезагрузки TLB будет отсрочена до тех пор, пока ядро не соберется использовать страницу повторно. Например, драйвер seg_map объявляет страницу устаревшей только тогда, когда ее счетчик ссылок станет равным нулю (это означает, что ни один процесс не может обратиться к странице). Следовательно, продолжение хранения потерявших значимость данных о преобразовании адреса такой страницы не производит никакого отрицательного воздействия на систему. Ядро возвращает страницу в свободный список и затем вызывает процедуру hat_getshootcookie() для связывания со страницей нового объекта cookie[1].

Перед повторным использованием страницы ядро передает объект cookie функции hat_shootdown(), которая производит его сравнение со значениями cookie на других процессорах. Если какой-то процессор владеет cookie, более давним по сравнению со сравниваемым, это означает, что информация буфера о странице изжила себя и такой буфер является кандидатом на уничтожение.

Процедура hat_shootdown() выполняется в контексте процесса-инициатора. Сначала она захватывает глобальный объект простой блокировки (spin lock), что гарантирует проведение в один момент времени только одной операции перезагрузки. Затем операция посылает межпроцессорное прерывание всем процессорам, обладающим устаревшим объектом cookie. После начала обработки запроса каждым процессором инициатор освобождает объект блокировки и завершает изменение отображения страницы.

Процессы обрабатывают прерывание сразу после получения, не ожидая каких-либо действий, обеспечивающих синхронизацию. Они производят сброс своих буферов TLB и продолжают после этого работу в обычном режиме. Так как инициатор выполнил изменения преобразований адресов перед началом перезагрузки, все последующие операции доступа к странице со стороны исполнителей приведут к загрузке корректного вхождения TLB. Межпроцессорное прерывание обладает наивысшим уровнем приоритета. При его обработке запрещены любые другие прерывания, так как новая операция перезагрузки, начатая в тот же промежуток времени, в состоянии повлечь взаимную блокировку.

---

[1] Толкование страниц сегмента seg_kmem реализовано несколько по-иному. Такие страницы управляются при помощи битовой маски, поделенной на зоны (по умолчанию каждая зона имеет размер 128 битов). Объект cookie ассоциируется с каждой такой зоной и устанавливается, когда адреса в зоне освобождаются.

Средства синхронизации, используемые при реализации описанного алгоритма, намного проще и эффективнее по сравнению с методикой системы Mach. Основной причиной этого является поддержка ядром системы SVR4.2/MP гарантий не обращения к некорректным страницам, не имевшихся в ОС Mach.

15.11.4. Незамедлительная перезагрузка

При сбросе ядром системы корректности пользовательского PTE необходимо произвести безотлагательную перезагрузку буферов TLB на всех процессорах, где соответствующий процесс может быть активен. Такая необходимость проистекает из-за того, что ядро не может контролировать правила доступа к памяти прикладных процессов и не гарантирует отсутствие попыток обращения к недействительным адресам с их стороны. Методика немедленной перезагрузки сложнее по сравнению с предыдущей, так как в этом случае процессам-исполнителям приходится ждать, пока инициатор завершит изменение элементов PTE, и только после этого заканчивать обработку прерывания.

Алгоритм синхронизации прикладных вхождений TLB в системе SVR4.2/MP схож со своим аналогом, реализованным в Mach. Ядро использует для этой цели дополнительный счетчик синхронизации, который играет роль семафора, разделяемого между инициатором и исполнителями. Последовательность операций продемонстрирована на рис. 15.14.

Если ядру нужно сбросить корректность прикладного вхождения PTE процессора, сначала производится блокировка структуры hat и последующее овладение глобальным объектом простой блокировки (с целью предупреждения конфликтов с другими операциями перезагрузки). Дальнейшим этапом работы ядра является отправка межпроцессорного прерывания каждому процессору, разделяющему обрабатываемое адресное пространство (для этой цели используется список процессоров, хранящийся в структуре hat). После приема прерывания исполнители переходят в ждущий цикл с выходом по условию увеличения значения счетчика синхронизации.

В тот момент, когда все исполнители переключились в состояние ожидания, инициатор проводит операцию изменения вхождений таблицы страниц и инкрементирует счетчик синхронизации. Исполнители покидают цикл ожидания, сбрасывают свои TLB и возвращаются из обработки прерывания. Последней стадией операции перезагрузки является сброс локального буфера TLB инициатором и последующее освобождение объекта блокировки и структуры hat.

Описанный алгоритм позже был оптимизирован с учетом архитектуры Intel для случая необходимости изменения инициатором сразу же нескольких вхождений PTE в одном адресном пространстве. Такое может происходить, к примеру, при освобождении процессом сегмента памяти. Процессор i386 имеет двухуровневую таблицу страниц (см. раздел 13.2.2), в которой

таблица страниц уровня 1 содержит PTE для таблиц второго уровня. В случае необходимости сброса нескольких PTE инициатор просто-напросто объявляет недействительными соответствующие вхождения таблицы первого уровня, после чего производит инкремент счетчика синхронизации.

Такой подход позволяет исполнителям завершить свою часть операции перезагрузки и продолжить работу в нормальном режиме без необходимости ожидания завершения модификации всех элементов PTE инициатором. Если легковесный процесс исполнителя пытается произвести обращение к устаревшей странице, то это приведет к страничной ошибке, так как вхождение таблицы первого уровня является некорректным. Загруженный в результате этого обработчик ошибки блокируется, поскольку структура hat по-прежнему остается блокированной инициатором. Таким образом, исполнитель не может обратиться к противоречивым преобразованиям до того времени, пока инициатор не завершит свою часть работы и не загрузит корректное отображение. Несмотря на элегантность описанного метода, он является весьма специфичным и применяется лишь в архитектуре Intel и помогает лишь в ограниченном количестве ситуаций.

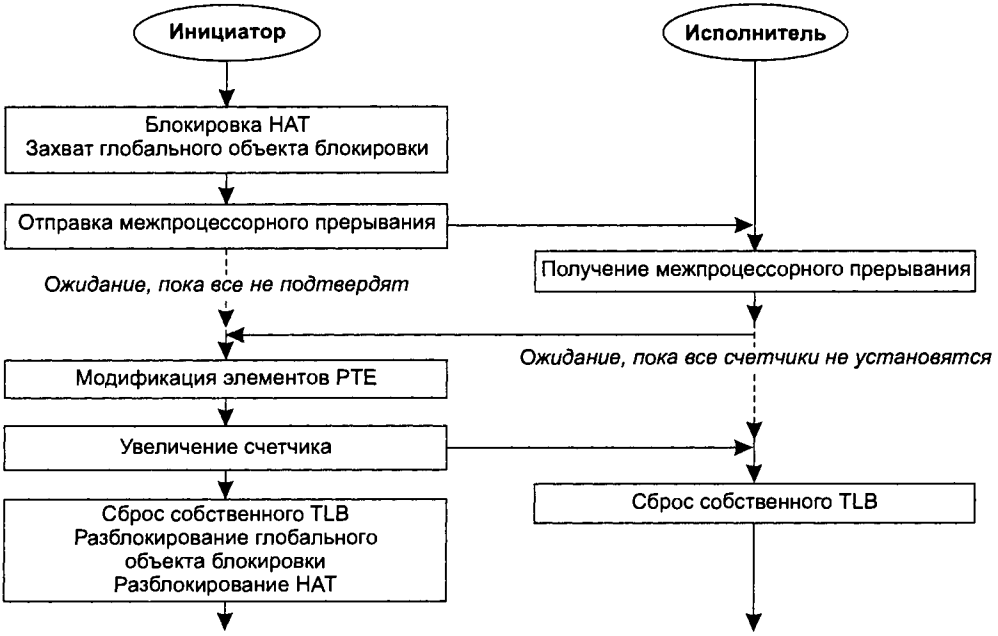


Рис. 15.14. Синхронизация буферов TLB прикладных процессов в SVR4.2/MP

Еще одним событием системы, требующим незамедлительной перезагрузки TLB, является выгрузка страниц. До появления SVR4.2/MP демон pagedaemon использовал правило глобальной замены страниц. Он мог производить проверку определенного количества страниц в общем пуле и сбрасывать бит

ссылки (для сбора информации о ссылках) или бит изменений (после освобождения страницы). Обе эти операции требуют глобальной перезагрузки вхождений TLB. Разработчики системы SVR4.2/MP заменили алгоритм, использовавшийся в предыдущих версиях, на методику, основанную на локальных установках устаревания. Перехватываются все процессы, являющиеся устаревшими (в том числе и все LWP процесса, кроме выполняемых на процессоре-инициаторе, для которого и производится выгрузка). При обратном подключении LWP механизм контекстного переключения i386 автоматически сбрасывает TLB.

15.11.5. Некоторые итоги

Разработчики SVR4/MP и SVR4.2/MP искали способ оптимизации перезагрузки TLB, опираясь на отличительные особенности аппаратных архитектур. Более того, каждая такая ситуация интерпретируется системами индивидуально, что дает возможность использовать преимущества синхронизации, унаследованной от функций, послуживших «пускателем» перезагрузки.

Такой подход показывает большую производительность по сравнению с алгоритмом, реализованным в Mach, в которой для всех машин и ситуации в сборе применяется единственная простейшая методика. Однако алгоритм SVR4 более сложен с точки зрения переноса на другие платформы, так как он в немалой степени зависит от конкретных аппаратной платформы и программной специфики. Например, архитектура машины MIPS R3000, обладающая поддержкой самоидентифицируемых вхождений TLB, имеет и свои отрицательные стороны. Основной проблемой для нее является отсутствие автоматического сброса TLB при переключении контекста. В разделе 15.12 показано решение, реализованное специально с учетом особенностей этой архитектуры.

Если подвести черту, мы снова видим попытку найти «золотую середину» между неким единым универсальным решением и использованием нескольких специализированных методик для разрешения каждой ситуации в отдельности.

15.12. Другие алгоритмы поддержания корректности TLB

Многопроцессорная версия системы SVR3, созданная специально для MIPS R3000 [23], поддерживает еще один вариант программного решения проблемы достоверности буфера TLB. В архитектуре MIPS предусмотрены самоидентифицируемые (tagged) вхождения TLB (см. раздел 13.3.4), [9]. Каждое вхождение TLB имеет шестиразрядный тег, называемый *TLBpid*, идентифицирующий адресное пространство, с которым соотносится преобразование. Такая особенность архитектуры имеет несколько важных последствий. Нет

необходимости сбрасывать TLB при переключении контекста, так как новый процесс будет обладать новым значением TLBpid. В результате процесс может выполняться независимо от вхождений TLB любого используемого им процессора. Если в какой-то момент времени процесс продолжит выполнение на том же CPU, он сможет повторно использовать буфер TLB до тех пор, пока тот не будет сброшен или его вхождения заменены индивидуально.

В описываемой системе необходимо отслеживать легальность операций изменения размера адресного пространства. Представьте, что процесс начинает выполнение на процессоре А и через некоторое время продолжает работу на процессоре Б. При выполнении на Б процесс изменяет размер своей области данных и сбрасывает отдельные вхождения TLB на Б. Если процесс в дальнейшем возобновит функционирование на А, то следует ожидать обращений к недействительным страницам через устаревшие вхождения TLB на процессоре А.

Для решения озвученной проблемы ядро присваивает процессу новый тег TLBpid при изменении размера адресного пространства. Это приводит к автоматическому сбросу корректности всех существующих вхождений TLB на всех процессорах[1]. Ядро должно производить глобальный сброс вхождений LTB, если оно собирается передать ранее использовавшееся значение TLBpid другому процессу. Ядро уменьшает вероятность появления необходимости глобального сброса посредством выделения тегов TLBpids в порядке FIFO (первым вошел, первым вышел), что позволяет производить сброс действительно устаревших вхождений TLB.

Реализация алгоритма для MIPS также учитывает случай записи процессом страницы, имеющей атрибут копирования при записи. Ядро создает новую копию страницы и передает ее процессу, осуществляющему запись. Это также сбрасывает данные о странице из локального буфера TLB. Если процесс ранее выполнялся на другом процессоре, вхождение TLB этого CPU может содержать устаревшую информацию о странице. Ядро поддерживает ведение записи процессоров, на которых выполняется процесс. Если после записи страницы, имеющей атрибут копирования при записи, процесс продолжит выполнение на одном из таких процессоров, после переключения контекста ядро произведет сброс буфера TLB процессора.

Оптимизации, описанные в этом разделе, способствуют минимизации операций глобальной синхронизации TLB и могут положительно повлиять на производительность системы. В частности, усовершенствования драйвера `seg_map` являются весьма полезными, так как отображения ядра разделяются между всеми процессами системы и вследствие этого `seg_map` используется практически постоянно. С другой стороны, представленные решения довольно специфичны и зависят как от особенностей аппаратных средств, так и функционирования системы, принимающей решение о синхронизации. Не

[1] Эта операция сбрасывает и все корректные вхождения TLB процесса.

существует какого-либо единого универсального алгоритма (кроме реализованного в ОС Mach), который являлся бы аппаратно-независимым и подходил для решения всех потенциально возможных ситуаций.

15.13. Виртуально адресуемый кэш

Буфер TLB представляет собой кэш преобразований адресов. Как правило, компьютеры оборудованы также высокоскоростными кэшами для физической памяти. В большинстве машин имеются либо отдельные кэши для данных и инструкций, либо единый кэш для обоих типов данных. Размер аппаратного кэша равен 64–512 Кбайт, скорость доступа к нему намного выше по сравнению со скоростью обращения к основной памяти. В большинстве случаев применяется технология кэширования с отложенной записью. Это означает, что операции записи данных изменяют только ту информацию, которая хранится в кэше. Данные сбрасываются в основную память непосредственно при необходимости проведения изменения внутри кэша, например, при удалении информации с целью освобождения пространства для новых данных.

Традиционные аппаратные архитектуры имеют физически адресуемый кэш (см. рис. 15.15). Диспетчер MMU производит преобразование виртуального адреса и лишь после этого обращается к физической памяти. Все операции с физической памятью осуществляются через кэш. Если необходимые данные найдены в кэше, блок MMU не обращается к физической памяти.

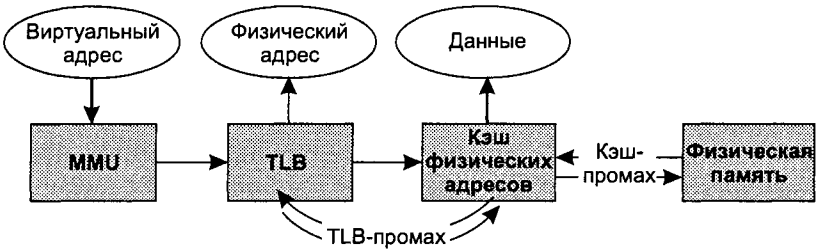


Рис. 15.15. Архитектура физически адресуемых кэшей

Такой подход имеет главное преимущество — простоту. Гарантом целостности кэша выступает аппаратная часть системы. Операционная система не выполняет эту задачу и не отвечает за информацию, размещаемую в кэше. Недостатком физически адресуемых кэшей является тот факт, что просмотр кэшированных данных можно осуществить только после преобразования адреса, что уменьшает преимущества применения кэширования. Более того, если буфер TLB не обладает корректным преобразованием, диспетчеру MMU приходится передавать элемент таблицы страниц из физической памяти. Это требует дополнительных обращений к кэшу и основной памяти машины.

Во многих современных архитектурах применяется виртуально адресуемое кэширование, что в некоторых случаях позволяет отказаться от буферов TLB (см. рис. 15.16). Диспетчер MMU производит поиск виртуального адреса в кэше. Если адрес будет найден, то на этом операция завершается. Если данные отсутствуют в кэше, устройство MMU продолжает работу с преобразованием адреса и получает необходимые данные из физической памяти.

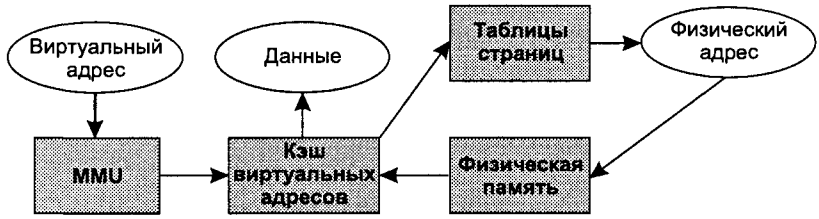


Рис. 15.16. Виртуально адресуемый кэш

Однако в системах могут использоваться одновременно и буфер TLB, и виртуально адресуемый кэш. В таких архитектурах (примерами которых являются MIPS R4000 [14] и Hewlett-Packard RA-RISC [11]) устройство MMU производит одновременный поиск информации в кэше и TLB. Этот подход дает высокую производительность работы за счет усложнения архитектуры.

Виртуально адресуемый кэш состоит из определенного количества *строк* (lines) кэширования, каждая из которых отображает определенное количество байтов памяти. Например, машина Sun-3 [20] имеет 64-килобайтовый кэш, состоящий из 16-байтовых строк. Кэш индексируется по виртуальному адресу, либо по комбинации виртуального адреса с *идентификатором процесса* или *идентификатором контекста*. В одну строку обычно отображается сразу несколько виртуальных адресов (относящихся к одному и тому же или различным адресным пространствам), поэтому строка должна содержать *тег*, идентифицирующий процесс и виртуальный адрес.

Применение виртуальных адресов для индексирования имеет одно важное последствие: для кэша может быть определен *фактор выравнивания* (alignment factor), например, если два виртуальных адреса отличаются на величину этого фактора, то они отображаются в одной строке кэша. Величина фактора выравнивания, как правило, равна длине кэша (или кратному значению). Для обозначения адресов, отображаемых в одной строке кэша, применяется термин *выравненные адреса* (aligned addresses).

Несмотря на то, что физически адресуемые кэши работают совершенно независимо от операционной системы, аппаратная часть машины не в состоянии гарантировать непротиворечивость виртуально адресуемых кэшей. Определенный физический адрес может быть отображен по нескольким виртуальным адресам и, следовательно, такое отображение может находиться в отличающихся друг от друга строках кэша, что является причиной внутрен-

ней проблемы корректности данных. Так как кэширование является отложенным, информация, размещаемая в основной памяти, может оказаться устаревшей по сравнению с данными кэша. Существуют три типа проблем достоверности данных виртуально адресуемых кэшей: изменение отображения (омонимы), псевдонимы адресов (синонимы) и операции прямого доступа к памяти (DMA).

15.13.1. Изменения отображений

Изменения отображения имеют место тогда, когда виртуальный адрес переназначается на другой физический адрес (см. рис. 15.17). Это может произойти в нескольких ситуациях.

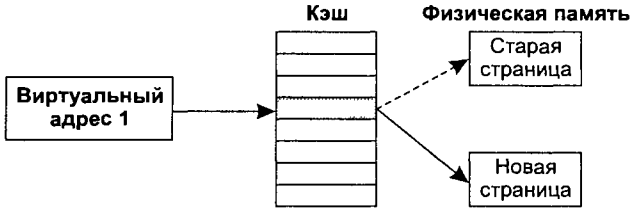


Рис. 15.17. Изменение отображения приводит к некорректности вхождения кэша

- **Переключение контекста.** При переключении контекста происходит замена предыдущего адресного пространства на пространство нового процесса. В большинстве архитектур тег строки кэша идентифицирует процесс, к которому относится эта строка. В результате переключение контекста не приводит к объявлению недействительной всей кэшируемой информации. Однако в большинстве систем область u размещается в адресном пространстве ядра. При переключении контекста ядро переназначает адреса области u в физические страницы области u другого процесса. Так как ядро является разделяемым, его вхождения в кэше имеют специальный тег, корректный для всех процессов. Таким образом, переключение контекста аннулирует все вхождения старой области u в кэше.
- **Выгрузка страницы.** Если демон `pagedaemon` выгружает страницу из памяти, то он также производит и сброс корректности всех вхождений кэша для этой страницы.
- **Изменение атрибутов защиты.** При изменении атрибута защиты страницы соответствующие вхождения кэша становятся некорректны. Изменение защиты может произойти после вызова функции `mprotect`, либо в результате имитации бита ссылки или вследствие применения копирования при записи ядром. Если защита страницы понижается (например, атрибут «только для чтения» замещается на «чтение-запись»), такое изменение является безопасным и не требует обновления

информации в кэше. Если процесс попытается обратиться к странице, произойдет исключение, при обработке которого в кэш будет загружена корректная информация. При упрочении защиты (например, если страница, доступная для записи, получает атрибут «только для чтения») произведенное изменение должно быть распространено и на все вхождения кэша для такой страницы.

- **Копирование при записи.** Если процесс осуществляет попытку записи страницы, разделяемой в режиме копирования при записи, ядро создаст для него новую копию страницы, сделает ее доступной для чтения и изменит отображения процесса для получения ссылки на созданную только что копию. Эти действия приводят к некорректности всех вхождений кэша, ссылающихся на изначальную страницу.

15.13.2. Псевдонимы адресов

Псевдонимы адресов (address aliases) или синонимы (synonyms) представляют собой набор виртуальных адресов одной и той же физической страницы (см. рис. 15.18). Когда процесс изменяет отображение, используя один из таких адресов, такое изменение не будет автоматически спроецировано на остальные строки кэша. Если другой процесс начнет чтение данных, основываясь на других адресах, то результатом станет обращение к устаревшим данным. Более того, если два процесса запишут местонахождение страницы по двум разным адресам, порядок, в котором эти записи будут помещены в память, не определен. Синонимы могут возникать вследствие нескольких причин.

- **Разделяемая память.** Если несколько процессов разделяют область памяти, каждый процесс отображает ее в собственном адресном пространстве. Если такие адреса не выровнены, то они будут попадать в разные строки кэша, что приведет к появлению синонимов. Так как процессы вправе отображать разделяемые области памяти по любому адресу в своем пространстве, ядро обычно не в состоянии гарантировать выравнивание.
- **Вызов mmap.** Процессы могут использовать вызов `mmap` для создания отображения файла или объекта памяти в любом месте своего адресного пространства. Если несколько процессов отображают один и тот же объект по не выровненным адресам, это приводит к созданию синонимов. Процесс также способен отображать одну и ту же область в различные части своего адресного пространства, с аналогичным результатом.
- **DVMA.** Системы, поддерживающие *прямой доступ к виртуальной памяти* (Direct Virtual Memory Access), позволяют устройствам создавать новое отображение виртуального адреса для страницы, уже имеющей другой виртуальный адрес. Эта операция требует сброса содержимого кэша при создании нового отображения и повторного сброса после ее завершения.

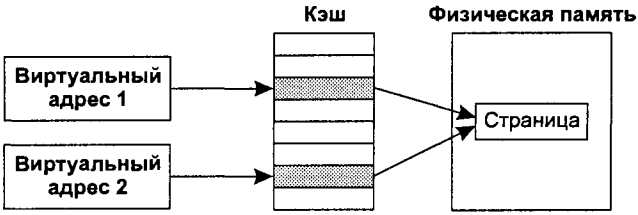


Рис. 15.18. Синонимы являются причиной многократных вхождений кэша для одних и тех же данных

15.3.3. Прямой доступ к памяти

Многие устройства обладают возможностью непосредственного обмена данных с памятью без участия процессора. Эта возможность получила название *прямого доступа к памяти* (Direct Memory Access, DMA)[1]. При использовании технологии DMA производится обращение сразу к основной памяти, минуя кэш. Несмотря на то, что это дает определенное увеличение скорости передачи, применение DMA становится причиной возникновения проблем корректности кэша. Представьте, что кэш содержит некоторое количество измененных данных, еще не записанных в основную память. Операция прямого чтения об этом не знает и, следовательно, считает из основной памяти устаревшую информацию. Сходные проблемы возникают и при прямой записи: операция не перезаписывает кэш, что делает содержимое кэша непригодным из-за устаревания данных. Через некоторое время устаревшая строка кэша может быть сброшена в память, тем самым будут «затерты» более свежие данные.

15.13.4. Поддержка корректности кэша

В системах с виртуально адресуемым кэшированием ядро должно уметь распознавать события, следствием которых является недостоверность кэша, и производить некоторые действия по исправлению данных. Обычно кэш экспортирует ядру две операции, flush и purge, которые позволяют очистить вхождение кэша или сбросить его. В отличие от операции purge (очистка), операция flush (сброс) также производит запись всех изменений в основную память машины. Действия ядра системы зависят от конкретной ситуации. Представим некоторые из них.

При изменении отображения страницы ядру необходимо сбросить все вхождения, относящиеся к этой странице. Сброс кэша является весьма трудоемкой операцией, поэтому должен производиться как можно реже. Ядро в любом случае сбрасывает вхождения избирательно, тогда, когда они были изменены с корректных на некорректные. Например, при переключении контекста ядро сбрасывает вхождения кэша области u предыдущего процесса, если ото-

[1] Следует отличать DMA от DVMA (см. раздел 15.13.2).

бражения помечены как некорректные. При подключении новой области u не потребуется сброса, так как это изменение в обратную сторону (invalid-to-valid).

Демон pagedaemon сбрасывает корректность освобождаемых страниц. Кэшированные вхождения таких страниц должны быть сброшены, чтобы не дать процессу возможности продолжать обращаться к неправильным страницам. Однако при заполнении свободной страницы новыми данными операция сброса не требуется.

При выполнении вызова fork ядра многих версий UNIX временно отображают страницы области u процесса-предка в часть адресного пространства ядра под названием forkutl. Процедуры, использующие область forkutl, не освобождают такое отображение после завершения работы. В случае применения систем с виртуально адресуемым кэшированием такие процедуры должны производить сброс соответствующих вхождений кэша.

Простейшим методом управления псевдонимами является их запрещение на аппаратном уровне [3]. Если процесс вызывает исключение (страничную ошибку), ядро проверяет страницу на предмет существования другого отображения. Если отображение будет обнаружено, ядро сбрасывает его вхождение и удаляет соответствующую страницу из кэша. После этого создается новое отображение. Для данной методики разделение памяти требует частого изменения отображений и сброса кэша, так как в один момент времени может быть корректно только одно отображение страницы.

Однако описанное выше решение является слишком затратным, в первую очередь, из-за длительности операции сброса кэша. Кроме этого, необходимо обрабатывать часто возникающие исключительные состояния, иначе при обращении к разделяемым страницам разными процессами будут постоянно возникать коллизии. Необходимо найти более подходящие альтернативы. Одним из наиболее очевидных улучшений является разрешение использования псевдонимов при обращении к страницам в режиме «только для чтения». Если процесс попытается произвести запись в такую страницу, ядро объявит недействительными все остальные отображения страницы.

Псевдонимы являются причиной появления некорректных данных только в том случае, если адреса не выравнены. Несмотря на то, что системы UNIX позволяют прикладным процессам создавать отображения разделяемых областей или других объектов памяти в адреса, указываемые пользователем, большинство приложений не зависят от наличия этой возможности. Системные вызовы типа shmat и mmap, как правило, имеют опции, предоставляющие выбор подходящего диапазона адресов ядру. Если это возможно, ядро выбирает для отображения выравненный адрес, что решает проблему некорректности кэша. Например, если на машине Sun-3 два адреса отличаются на некоторую величину, кратную 128 Кбайт, они отображаются в одно вхождение кэша. В этой системе ядро пытается находить адреса, которые эквивалентны, либо отличаются на значение, кратное 128 Кбайт, с целью создания отображения для другого процесса, разделяющего область памяти.

Многие реализации UNIX еще более консервативны и либо вообще запрещают подобные возможности, либо ограничивают их применение. Например, во многих версиях операционной системы HP-UX корпорации Hewlett-Packard [5] не поддерживается вызов mmap. При разделении области памяти должны быть отображены по одним и тем же адресам всех процессов. В таких системах совместное использование текстов реализовано за счет глобального разделяемого сегмента виртуальных адресов. В системе SunOS проблема решена путем запрета кэширования страниц, имеющих несколько невыравненных виртуальных адресов.

Операции прямого доступа к памяти должны также обрабатываться особым образом при использовании виртуально адресуемых кэшей. Перед началом чтения в режиме DMA ядро обязано сбрасывать из кэша всю информацию из считываемых страниц. Это дает гарантию, что основная память не содержит данных, являющихся более ранними по сравнению с кэшированной информацией. Точно так же при записи DMA ядро должно сначала удалить все вхождения кэша, содержащие перезаписываемые данные. Если этого не сделать, кэш будет содержать устаревшие вхождения, которые впоследствии могут быть сброшены в оперативную память, тем самым будет перезаписана информация, полученная при прямом доступе к памяти.

15.13.5. Анализ

Применение виртуально адресуемого кэширования может значительно увеличить скорость обращения к памяти. Однако это приводит и к возникновению множества проблем корректности данных, которые необходимо решать на программном уровне. Более того, виртуально адресуемые кэши сильно влияют на архитектуру памяти, что приводит к необходимости изменения структуры операционных систем. Несмотря на то, что кэши были созданы для увеличения производительности MMU, они конфликтуют с определенными допущениями операционной системы и могут отрицательно влиять на общее быстродействие систем.

Современные варианты UNIX поддерживают многие формы разделения и отображения памяти, такие как разделяемая память System V, отображение файлов, технология копирования при записи, используемые при наследовании памяти и взаимодействии процессов. В традиционных архитектурах эти технологии позволяют уменьшить число операций копирования внутри памяти и сэкономить пространство памяти путем отказа от хранения нескольких копий одних и тех же данных. Это обеспечивает значительное увеличение совокупной производительности.

Однако в архитектурах виртуально адресуемых кэшей такое разделение памяти может привести к появлению синонимов. Операционная система должна иметь процедуры восстановления, обеспечивающие гарантию целостности кэша. В их число входит сброс кэша, запрет кэширования части стра-

ниц, а также запрет или ограничение отдельных средств. Такие операции могут свести на нет все преимущества разделения памяти. В [5] демонстрируется, что, несмотря на тот факт, что сброс кэша занимает лишь 0,13% общего времени, в некоторых тестах эта величина достигает 3%.

Для эффективной работы в системах с виртуально адресуемым кэшированием необходимо пересмотреть некоторые алгоритмы. В работе [8] показаны некоторые изменения систем Mach и Chorus, позволяющие решить описываемые проблемы. Работа [24] содержит несколько способов предупреждения лишних операций поддержки корректности кэша, использование которых приводит к значительному росту производительности. Некоторые предложения весьма специфичны и подходят только для архитектуры PA-RISC корпорации Hewlett-Packard. На таких машинах просмотр буфера TLB производится параллельно с поиском адреса в кэше. Кэш поддерживает теги физического адреса. Это позволяет обнаруживать большинство случаев некорректности программно и применять более эффективные меры поддержки целостности.

15.14. Упражнения

1. Чем отличается наследование памяти в системах Mach и SVR4?
2. Что произойдет при попытке задачи A произвести запись страницы (см. пример, изображенный на рис. 15.2)?
3. Почему интерфейс внешнего менеджера памяти системы Mach является причиной низкой производительности?
4. Чем отличаются *объект vm* от *объекта памяти*?
5. Как поведет себя сетевой сервер разделения памяти в случае сбоя системы одного из клиентов? Что произойдет в результате отказа сервера?
6. Объясните, почему системе Mach не требуется поддержка массива элементов защиты страниц, аналогичного имеющемуся в SVR4.
7. Представьте, что производитель организует поддержку System V IPC в системе, основанной на ядре Mach. Каким образом в такой системе должен быть реализован интерфейс разделяемой памяти? Какие проблемы при этом придется решить?
8. Опишите отличия между вызовами vm_map системы Mach, mmap 4.4BSD и mmap SVR4, а также их сходства.
9. В разделе 15.8 упоминается защитный семафор, используемый аналогично набору семафоров системы System V в 4.4BSD. Как должен выделяться и управляться такой семафор: на уровне ядра или на уровне задачи? Схематично опишите реализацию семафора.
10. Почему правила замены страниц в системе Mach получили название *FIFO с возможностью второй попытки*?

11. Какими преимуществами обладает программная перезапись буфера TLB по сравнению с аппаратной реализацией этой задачи?
12. Представьте вхождение TLB, в котором хранится поддерживаемый аппаратно бит ссылки. Каким образом такой бит может быть использован ядром?
13. Ядро системы UNIX не является страничным. Что может привести к изменению вхождения TLB для страницы ядра?
14. Почему перезагрузка TLB в Mach считается трудоемкой операцией? Поясните это в сравнении с аналогичными процедурами в системах SVR4/MP и SVR4.2/MP.
15. Охарактеризуйте, каким образом SVR4/MP и SVR4.2/MP обрабатывают некорректность TLB, возникшую в результате записи страницы, имеющей атрибут копирования при записи.
16. К возникновению каких дополнительных проблем корректности TLB приводит поддержка легковесных процессов?
17. Объясните, почему технология отложенной перезагрузки является наиболее предпочтительной по сравнению с немедленной перезагрузкой в большинстве случаев? Когда следует использовать безотлагательную перезагрузку?
18. Требуется ли диспетчеру MMU с виртуально адресуемым кэшем буфер TLB? Какие вы видите преимущества и недостатки отсутствия TLB?
19. Поясните разницу между псевдонимом адреса и изменением отображения.
20. Каким образом ядро системы обеспечивает корректность TLB и виртуально адресуемого кэша во время выполнения вызова exec?

15.15. Дополнительная литература

1. Balan, R., and Golhardt, K., «A Scalable Implementation of Virtual Memory HAT Layer for Shared Memory Multiprocessor Machines», Proceedings of the Summer 1992 USEN1X Technical Conference, Jun. 1992, pp. 107–115.
2. Black, D. L, Rashid, R., Oolub, D, Hill, C, and Baron, R., «Translation Lookaside Buffer Consistency: A Software Approach», Proceedings of the Third Internationla Conference on Architectural Support for Programming Languages and Operating Systems, Apr. 1989, pp. 113–132.
3. Chao, C, Mackey, M., and Sears, B., «Mach on a Virtually Addresses Cache Architecture», Proceedings of the First Mach USENIX Workshop, Oct. 1990, pp. 31–51.
4. Cheng, R., «Virtual Address Cache in UNIX», Proceedings of the Summer 1987 USENIX Technical Conference, Jun. 1987, pp. 217–224.

5. Clegg, F.W., Ho, G. S.-F., Kusmer, S. R., and Sontag, J. R., «The HP–UX Operating System on HP Precision Architecture Computers», Hewlett-Packard Journal, Vol. 37, No. 12, 1986, pp. 4–22.
6. Draves, R. P., «Page Replacement and Reference Bit Emulation in Mach», Proceedings of the Second USENIX Mach Symposium, Nov. 1991, pp. 201–212.
7. Golub, D. B., and Draves, R. P., «Moving the Default Memory Manager Out of the Mach Kernel», Proceedings of the Second USENIX Mach Symposium, Nov. 1991, pp. 177–188.
8. Inouye, J., Konuru, R., Walpole, J., and Sears, B., «The Effects of Virtually Addressed Caches on Virtual Memory Design and Performance», Operating Systems Review, Vol. 26, No. 4, Oct. 1992, pp. 14–29.
9. Kane, G., «Mips RISC Architecture», Prentice-Hall, Englewood Cliffs, NJ, 1988.
10. Kleiman, S. R., «Vnodes: An Architecture for Multiple File System Types in Sun UNIX», Proceedings of the Summer 1986 USENIX Technical Conference, Jun. 1986, pp. 238–247.
11. Lee, R. B., «Precision Architecture», IEEE Computer, Vol. 21, No. 1, Jan. 1989, pp. 78–91.
12. McKusick, M. K., «A New Virtual Memory Implementation for Berkeley UNIX», Computing Systems, Vol. 8, No. 1, Winter 1995.
13. McNamee, D., and Armstrong, K., «Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement», Proceedings of the First Mach USENIX Workshop, Oct. 1990, pp. 17–29.
14. MIPS Computer Systems Inc., «MIPS R4000 Preliminary Users Guide», 1990.
15. Peacock, J. K., Saxena, S., Thomas, D., Yang, F., and Yu, W., «Experiences from Multithreading System V Release 4», Proceedings of the Third USENIX Symposium on Distributed and Multiprocessor Systems (SEDMS III), Mar. 92, pp. 77–91.
16. Pfister, G. F., Brantley, W. C., George, D. A., Harvey, S. L., Kleinfelder, W. J., McAuliffe, K. P., Melton, E. A., Norton, V. A., and Weiss, J., «The IBM Research Parallel Prototype (RP3): Introduction and Architecture», Proceedings of the 1985 International Conference on Parallel Processing, IEEE Computer Society, 1985, pp. 764–771.
17. Rashid, R. F., Tevanian, A., Young, M., Golub, D., Black, D., Bolosky, W., and Chew, J., «Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures», IEEE Transactions on Computing, Vol. 37, No. 8, Aug. 1988, pp. 896–908.
18. Rosenburg, B. S., «Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors», Eleventh ACM Symposium on Operating Systems Principles, Nov. 1987, pp. 137–146.

19. Subramanian, I ., «Managing Discardable Pages with an External Pager», Proceedings of the Second USENIX Mach Symposium, Nov. 1991, pp. 201–212.
20. Sun Microsystems, Inc., «Sun-3 Architecture: A Sun Technical Report», Aug. 1986.
21. Teller, P., Kenner, R., and Snir, M., «TLB Consistency on Highly Parallel Shared Memory Multiprocessors», Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, IEEE Computer Society, 1988, pp. 184–192.
22. Tevanian, A., Rashid, R. F., Young, M. W., Golub, D. B., «Thompson, M. R, Bolosky, W., and Sanzi, R., A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach», Technical Report CMU-CS-1-87, Department of Computer Science, Carnegie-Mellon University, Jul. 1987.
23. Thompson, M. Y., Barton, J. M., Jermoluk, T. A., and Wagner, J. C., «Translation Lookaside Buffer Synchronization in a Multiprocessor System», Proceedings of the Winter 1988 USENIX Technical Conference, Jan. 1988, pp. 297–302.
24. Wheeler, R., and Bershad, B. N., «Consistency Management for Virtually Indexed Caches», Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1992.
25. Young, M. W., Tevanian, A., Rashid, R. F., Golub, D. B., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R., «The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating Systems», Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Nov. 1987, pp. 63–76.

Глава 16
Ввод-вывод и драйверы устройств

16.1. Введение

Управление обменом данных между памятью и периферийными устройствами, такими как диски, принтеры и терминалы, осуществляется при помощи подсистемы ввода-вывода. Ядро взаимодействует с такими аппаратными компонентами через *драйверы устройств*. Драйвер управляет одним или несколькими устройствами и представляет собой интерфейс между устройством и остальной частью ядра системы. Такое разделение помогает скрыть особенности аппаратуры от ядра, которое обращается к устройствам, используя простой процедурный интерфейс.

Подробное изложение драйверов устройств выходит за рамки данной книги. Этой теме полностью посвящены многие другие фолианты, например [4] и [9]. Большинство производителей систем UNIX публикуют подробные описания, в которых можно найти информацию о создании драйверов под эти ОС [10]. В данной главе предлагается краткий обзор базовой структуры драйвера устройства UNIX. Большая часть описания посвящена интерфейсам системы SVR 4. Вы ознакомитесь с недостатками и преимуществами реализации драйверов в этой системе, а также с альтернативными подходами. В этой главе также рассказывается о подсистеме ввода-вывода, являющейся частью операционной системы, реализующей аппаратно-независимую обработку запросов ввода-вывода.

16.2. Краткий обзор

Драйверы устройств являются частью ядра и представляют собой набор структур данных и функций, управляющих одним или несколькими устройствами и взаимодействующих с остальной частью ядра посредством определенного интерфейса. Драйвер во многом отличается от внутренних компонентов ядра и является обособленным от них. Драйвер — это единственный модуль ядра, который может взаимодействовать с устройством. Драйверы часто создаются

сторонними производителями, как правило, создателями оборудования. Драйверы не взаимодействуют друг с другом, ядро может обращаться с ним только через средства интерфейса. Такой подход имеет несколько очевидных преимуществ:

- участки кода, зависящие от конкретного устройства, можно размещать в отдельных модулях;
- в систему легко добавлять новые устройства;
- производители в состоянии создавать драйверы без привлечения исходных текстов ядра;
- ядро системы видит все устройства одинаково и может обращаться к ним посредством одного и того же интерфейса.

На рис. 16.1 показана роль драйверов устройств в системе. Прикладные приложения взаимодействуют с периферийными устройствами через ядро, оперируя системными вызовами. Обработкой этих вызовов занимается подсистема ввода-вывода, которая, в свою очередь, использует интерфейс драйверов устройств для взаимодействия с необходимым оборудованием.

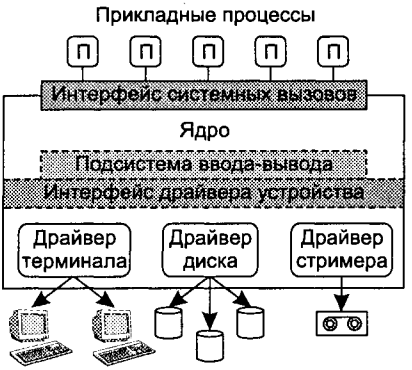


Рис. 16.1. Роль драйверов устройств в системе

Каждый уровень обладает четкой определенной задачей и ответственностью. Прикладным приложениям не нужно знать, взаимодействуют ли они с обычным файлом или устройством. Программа записи данных в файл должна обладать возможностью записи тех же данных в терминал или последовательную линию без внесения в себя каких-либо изменений или повторной компиляции. Следовательно, операционная система отвечает за целостное высокоуровневое представление себя для прикладных процессов.

Ядро передает все операции драйверов подсистеме ввода-вывода, которая отвечает за все аппаратно-зависимые действия. Подсистема ввода-вывода не обладает информацией о характеристиках отдельных устройств. Для нее устройства представляют собой абстракции высокого уровня, управляемые драйверами. Подсистема ввода-вывода отвечает за такие функции, как управление доступом, буферизация или присвоение устройствам имен.

За взаимодействия с устройством всецело отвечает драйвер этого устройства. Каждый драйвер управляет одним или несколькими сходными устройствами. Например, один дисковый драйвер может управлять несколькими дисками. Драйвер является единственным компонентом системы, который владеет информацией об аппаратных характеристиках устройства, таких как количество секторов, дорожек и головок диска или *скорости передачи в бодах* (baud rates) последовательной линии.

Драйвер воспринимает команды от подсистемы ввода-вывода, получаемые через интерфейс драйвера устройства. Он также принимает управляющие сообщения от самого устройства, в число которых входят уведомления о завершении, состоянии или ошибках. Как правило, устройство передает такие сообщения посредством генерации прерывания. Каждый драйвер имеет *обработчик прерываний*, который загружается ядром при возникновении определенного прерывания.

16.2.1. Аппаратная часть

Драйверы устройств по своей натуре являются аппаратно-зависимыми. Драйвер содержит коды, напрямую работающие с процессором машины. На рис. 16.2 показана упрощенная схема взаимодействия аппаратных устройств компьютера. Системная шина является высокоскоростной магистралью с широкой полосой пропускания. К ней присоединяется процессор (или CPU), устройство MMU и контроллеры устройств. К примеру, на машинах Intel 80486 применяются шины *ISA* (промышленная стандартная архитектура, Industry Standard Architecture) или *EISA* (расширенная промышленная стандартная архитектура, Extended Industry Standard Architecture)[1]. В некоторых компьютерах периферийные устройства присоединяются к отдельной шине ввода-

[1] Многие «486» машины также оборудованы шиной *PCI* (взаимосвязь периферийных компонентов, Peripheral Components Interconnect). — *Прим. авт.* Industry Standard Architecture — неофициальное название шины компьютеров IBM PC (Integrated Systems Architecture, архитектура интегрированных систем) с процессорами Intel x86, позволяющей добавлять различные устройства, вставляя их в гнезда платы расширений. Она использовалась еще в первом компьютере IBM PC/XT в 1981 году, в 1984 г. расширилась до 16-разрядной в IBM PC/AT. Отличается надежностью, совместимостью с различными периферийными устройствами, неплохим быстродействием (66,64 Мбит/с) и широкими возможностями. Процессоры 386 перевели ISA в класс устаревших (разница в разрядности), поскольку 32-разрядные версии ISA не отличаются стандартностью и используются платами расширения памяти и видеоадаптерами. В результате IBM разработала новый стандарт — шину MCA (Micro Channel Architecture, микроканальная архитектура). Он не получил распространения из-за несовместимости с адаптерами, разработанными для ISA, и вследствие требования IBM к изготовителям заплатить за использование шины ISA во всех выпущенных ранее компьютерах. VLB (VESA, VL-bus, VESA Local Bus), MCA или EISA можно встретить ныне только в старых компьютерах, а вот ISA и наиболее быстрая 32-разрядная PCI (1066,56 Мбит/с) стали стандартом после 1995 г. (в портативных ПК используется PCMCIA (или PC Card)). — *Прим. ред.*

вывода, например MASBUS или UNIBUS, которая, в свою очередь, подключена к системе через *адаптер*.

Мы можем рассматривать устройство как совокупность двух компонентов: электрическую часть, называемую контроллером или адаптером, и механическую часть, представляющую устройство само по себе. *Контроллер*, как правило, является печатной платой, устанавливаемой в компьютер и присоединенной к шине. В традиционных конфигурациях настольных компьютеров обычно имеются дисковый контроллер, видеокарта, карта ввода-вывода и, возможно, сетевая карта.

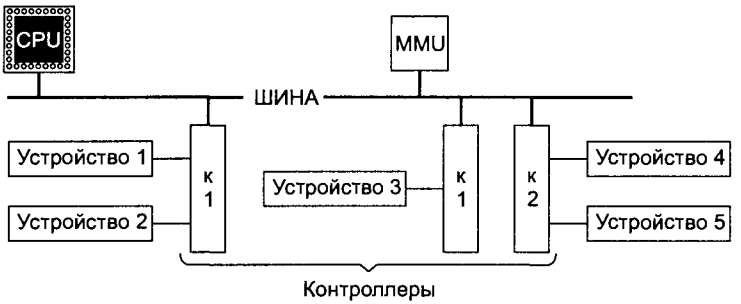


Рис. 16.2. Аппаратная архитектура обычной компьютерной системы

К каждому контроллеру может присоединяться одно или несколько устройств. Эти устройства чаще являются однотипными, но это необязательное требование. Например, контроллер SCSI (интерфейс малых вычислительных систем, Small Computer Systems Interface) может управлять работой таких устройств, как жесткие диски, дисководы, приводы компакт-дисков и ленточные накопители [1].

Контроллер обладает набором *управляющих регистров и регистров состояния* (Control and Status Registers, CSR). Каждое устройство может иметь один или несколько регистров CSR, функции которых полностью зависят от его архитектуры. Для подачи команды в устройство драйвер записывает данные в регистр, для получения информации о состоянии и ошибках он производит чтение из регистра. Эти регистры отличаются от регистров общего назначения. Запись в управляющий регистр приводит к выполнению некоторого действия устройства, например, инициализации дискового ввода-вывода или прогона бумаги в принтере. Чтение регистра состояния может иметь некоторые сторонние эффекты, например, очистку его содержимого. Следовательно, если драйвер дважды прочтет содержимое регистра, то он может получить неодинаковые результаты. Точно так же, если он попытается прочесть сразу после записи в регистр, считанные данные могут отличаться от только что внесенных в этот регистр[1].

[1] Один и тот же регистр может быть одновременно как управляющим, так и регистром состояния, и вследствие этого доступен для чтения и записи.

Пространство ввода-вывода компьютера включает в себя набор всех регистров устройств, а также буферы фреймов для устройств, отображаемых в память, таких как графические терминалы. Каждый регистр имеет четко определенный адрес в пространстве ввода-вывода. Такие адреса, как правило, присваиваются при загрузке системы, опираясь на параметры, указываемые в конфигурационном файле, который используется для первоначальной сборки системы. Каждому контроллеру присваивается диапазон адресов, в котором он выделяет пространство для каждого обслуживаемого устройства.

Существует два метода настройки пространства ввода-вывода системы. В некоторых архитектурах (к ним относится Intel 80x86) пространство ввода-вывода отделено от основной памяти и доступно при помощи специализированных команд ввода-вывода (например, inb или outb). В других архитектурах, например в Motorola 680x0, применяется подход под названием *ввод-вывод устройств, отображаемых в память* (memory-mapped device I/O). Регистры ввода-вывода отображаются в часть основной памяти, для чтения-записи регистров применяются обычные команды обращения к памяти.

Точно также существуют две метода передачи данных между ядром системы и устройством. Используемый метод зависит от конкретного устройства. Все устройства можно разделить на две категории, исходя из выбранного для них способа обмена данными: *программируемый ввод-вывод* (Programmed I/O, PIO) или *прямой доступ к памяти* (DMA). Устройства PIO передают данные побайтно, используя для этой цели центральный процессор машины. Если такое устройство готово принять или передать следующий байт, генерируется прерывание. Если устройство поддерживает DMA, ядро может предоставить ему сведения о местонахождении данных в памяти (их источнике или приемнике), размер передаваемых данных и другую необходимую информацию. Устройство производит обмен данными посредством прямого обращения к памяти, не привлекая для этого процессор системы. После завершения обмена устройство прерывает работу процессора, уведомляя его тем самым о готовности к новой операции.

Как правило, метод PIO используется медленными устройствами, такими как модемы, символьные терминалы и принтеры, в то время как дисковые накопители и графические терминалы являются устройствами DMA. В некоторых архитектурах, например SPARC, поддерживается метод DVMA (*прямой доступ к виртуальной памяти*, Direct Virtual Memory Access), позволяющий устройствам взаимодействовать напрямую с диспетчером MMU, передавая данные на виртуальные адреса. В этом случае устройства могут обмениваться друг с другом непосредственно, не используя основную память машины.

16.2.2. Прерывания устройств

Устройства вырабатывают прерывания с целью привлечения внимания центрального процессора. Обработка таких прерываний является аппаратно-

зависимой, но мы можем описать некоторые общие принципы их работы. Во многих системах UNIX определен набор *приоритетов прерываний устройств* (ipl). Количество поддерживаемых уровней в системах может быть неодинаковым. Низшим приоритетом является нуль. Фактически на этом уровне выполняются все прикладные процессы и большая часть кодов ядра. Высший приоритет зависит от конкретной реализации и, как правило, равняется 6, 7, 15 или 31. Если уровень ipl прерывания ниже по сравнению с текущим уровнем ipl системы, такое прерывание блокируется до тех пор, пока приоритет системы не станет выше ipl прерывания. Такой подход позволяет системе разграничить различные типы прерываний соответственно их значимости.

Каждое устройство генерирует прерывания с определенным уровнем ipl, как правило, все устройства, подключенные к одному устройству, имеют одинаковый ipl. При их обработке ядро сначала присваивает системе приоритет, равный ipl прерывания, что позволяет заблокировать последующие прерывания от того же устройства (а точнее, все прерывания, имеющие равный или более низкий уровень ipl). Более того, некоторые процедуры ядра повышают приоритет системы с целью временной блокировки прерываний. Например, процедура, управляющая очередью буферов дисковых блоков, поднимает уровень ipl с целью блокировки всех дисковых прерываний. В ином случае прерывание может произойти в тот момент времени, когда очередь находится в непротиворечивом состоянии, что приведет к проблемам дискового драйвера.

Для управления ipl ядро использует набор процедур. Например, spltty() применяется для увеличения уровня ipl до приоритета прерывания терминала. Процедура splx() снижает ipl до предыдущей сохраненной величины. Такие процедуры, как правило, реализованы в виде макроопределений с целью увеличения их эффективности.

При возникновении прерывания обычно загружается одна и та же процедура ядра, которая передает ядру некоторую информации о прерывании. Такая процедура производит сохранение контекста регистров, повышает приоритет системы до равного возникшему прерыванию и затем вызывает соответствующий обработчик прерывания. После завершения его выполнения управление вновь передается процедуре, которая восстанавливает приоритет системы, контекст регистров и заканчивает работу.

Каким образом ядро системы определяет обработчик, необходимый для возникшего прерывания? Это зависит от того, поддерживает ли система *векторные* (vectored, или опрашиваемые — polled) прерывания. В полностью векторных системах каждое устройство поддерживает уникальный *номер вектора прерываний*, который является индексом к *таблице векторов прерываний*. Элементы таблицы являются указателями на соответствующие обработчики прерываний.

Некоторые системы работают только с уровнями ipl. В принципе, они могут поддерживать и векторы прерываний, но существует вероятность, что один и тот же вектор будет назначен сразу нескольким устройствам. В этом случае ядру необходимо решать, какой из обработчиков прерываний следует загрузить. Ядро поддерживает связанный список для всех обработчиков, имеющих одинаковый уровень ipl (или одинаковый вектор прерываний). При возникновении прерывания процедура проходит по всей цепочке и опрашивает каждый драйвер. В свою очередь, драйвер проверяет, не создано ли это прерывание одним из его устройств. Если это так, он обрабатывает прерывание и возвращает процедуре данные об успешном завершении операции. В противоположном случае драйвер возвращает ошибку, а процедура обращается к следующему драйверу в списке.

Существует возможность комбинирования обоих методов. Системы, поддерживающие векторы прерываний, также могут обращаться к обработчикам через связанный список. Это дает возможность упрощения динамической загрузки драйвера устройства в работающей системе. Это также позволяет производителям замещать *драйверы* (override drivers) в целях отладки, устанавливая их в голове связанного списка. Переопределенные драйверы выборочно генерируют исключения в системе и обрабатывают определенные прерывания, а затем передают их драйверу, установленному по умолчанию.

Обработка прерываний является наиболее важной задачей системы. Выполнение обработчика прерывания более приоритетно по сравнению с любыми прикладными или системными программами. Так как обработчик останавливает все остальные процессы системы (кроме, конечно же, прерываний с более высоким приоритетом), он должен работать максимально быстро. В большинстве реализаций UNIX обработчики прерываний не могут быть переведены в режим сна. Если обработчику необходим заблокированный ресурс, он попытается получить его неблокирующим образом.

Описанные выше допущения влияют на действия обработчика. С одной стороны, он должен быть «коротким» и быстрым, настолько, насколько это возможно. Во-вторых, он должен выполнять достаточный объем работы, гарантирующий, что устройство не будет простаивать при высокой загрузке. Например, после завершения операции ввода-вывода контроллер диска генерирует в системе прерывание. Обработчик обязан уведомить ядро о результатах завершения операции. Он также должен произвести инициализацию следующей операции ввода-вывода, если имеется ожидающий этого запрос. С другой стороны, диск будет простаивать, пока ядро не получит управление и не начнет выполнение следующего запроса.

Несмотря на то, что описанные механизмы являются общими для большинства вариантов UNIX, они не универсальны. Например, разработчики Solaris 2.0 отказались от применения ipl, кроме ограниченного числа случаев. Система использует для обработки нити ядра и разрешает для таких нитей блокировку, если это необходимо (см. раздел 3.6.5).

16.3. Базовая структура драйвера устройства

В этом разделе вы увидите описание базовой структуры драйверов устройств UNIX с точки зрения их разработчика, в том числе описание интерфейса взаимодействия между драйвером и остальной частью ядра, а также здесь будет рассказано о представлении устройств и их драйверов подсистемой ввода-вывода.

16.3.1. Классификация устройств и их драйверов

Подсистема ввода-вывода управляет всей независимой от аппаратуры частью операций ввода-вывода. Ей требуется обеспечить высокоуровневое процедурное представление устройств. С точки зрения подсистемы ввода-вывода устройство представляет собой «черный ящик», поддерживающий стандартный набор операций. Внутри каждого устройства эти операции реализованы по-разному, однако подсистема ввода-вывода не знает об этом. В терминах объектно-ориентированного подхода (см. раздел 8.6.2) интерфейс взаимодействия с драйверами является абстрактным базовым классом, каждый драйвер представляет собой класс-наследник или его специфическую реализацию. На практике единый интерфейс взаимодействия не подходит для всех существующих устройств, так как они существенно различаются между собой по функциональности и методам доступа. В системах UNIX все устройства разделены на два типа — *блочные* и *символьные*. Для каждого типа определен собственный интерфейс взаимодействия.

Блочное устройство хранит данные и производит ввод-вывод блоками фиксированного размера, доступными в произвольном порядке. Обычно размер блока равняется 512 байтам, умноженным на 2 в степени, где степень больше либо равна нулю. В качестве примеров блочных устройств можно привести жесткие диски, флоппи-дисководы и приводы компакт-дисков. Файловая система UNIX способна храниться только на устройствах блочного типа. Ядро взаимодействует с такими устройствами через драйверы, используя структуры `buf`, в которых содержится вся необходимая информация об операциях ввода-вывода.

Как правило, блочные устройства обмениваются данными с областью памяти, называемой буферным кэшем (см. раздел 9.12). Каждый блок кэша обладает ассоциированной с ним структурой `buf`. В современных реализациях UNIX подсистема памяти поддерживает большее количество возможностей, чем обычный буферный кэш, который на их фоне теряет свою актуальность и, как правило, имеет небольшой объем. Из сказанного выше можно сделать вывод, что блочные устройства производят большинство операций ввода-вывода в страничную память машины. Несмотря на это они продолжают ис-

пользовать структуры buf для описания операций ввода-вывода. Такие структуры динамически ассоциируются с адресами данных в памяти и передаются драйверу. При чтении из блочного устройства в буфер или записи в него посредником выступает либо блок буферного кэша, либо область памяти, указанная в структуре buf. Более подробно читайте об этом в разделе 16.6.

Символьные устройства могут использоваться для хранения и передачи данных произвольного объема. Некоторые устройства этого типа умеют передавать информацию побайтно, вырабатывая каждый раз прерывание. Другие устройства способны поддерживать внутреннюю буферизацию. Ядро интерпретирует данные от символьных устройств как непрерывный поток байтов, доступный в последовательном режиме. Символьные устройства не в состоянии использовать произвольную адресацию и не поддерживают операцию поиска. Примерами устройств такого типа являются терминалы, принтеры, «мыши» и звуковые карты.

Не все устройства можно отнести к одной из вышеописанных категорий. В системах UNIX любое устройство, не обладающее свойствами блочного типа, классифицируется как символьное. Некоторые устройства вообще не производят никакого ввода-вывода. Например, аппаратный таймер является устройством, обеспечивающим через определенный интервал времени прерывание работы процессора (обычно с частотой 100 раз в секунду)[1]. Отображаемые в память дисплеи являются доступными в произвольном порядке, но они не считаются символьными устройствами. Некоторые блочные устройства, такие как диски, обеспечивают символьный интерфейс взаимодействия, так как он является более эффективным для определенных операций.

Драйвер не всегда управляет физическим устройством. Он может использоваться в качестве интерфейса доступа, поддерживающего дополнительные функции. Например, драйвер mem позволяет считывать из адресов физической памяти или записывать по ним. Устройство null обладает интересными свойствами: оно разрешает только запись в себя, производя удаление всех получаемых данных. Устройство zero является источником памяти, заполняемой нулями. Перечисленные выше устройства получили название *псевдоустройств*.

Важным преимуществом драйверов псевдоустройств является тот факт, что именно они позволяют сторонним производителям добавлять в ядро UNIX новые средства. Драйверы UNIX поддерживают общецелевую точку входа под названием ioctl. Эта процедура может быть вызвана с огромным количеством специфичных для определенного драйвера команд в качестве входного параметра. Это позволяет драйверам псевдоустройств обеспечивать на прикладном уровне богатый выбор способов вызова функций ядра, причем нет нужды изменять коды ядра.

[1] В архитектуре IBM PC выход микросхемы таймера обрабатывается аппаратным прерыванием 18,2 раза в секунду. — *Прим. ред.*

Современные системы UNIX поддерживают еще один класс устройств, называемый STREAMS. Устройства этого типа обычно используются для управления сетевыми интерфейсами и терминалами, а также заменяют собой устройства символьного типа для более ранних реализаций. С точки зрения сохранения обратной совместимости интерфейс драйверов STREAMS унаследован от символьных устройств (см. подробнее в разделе 16.3.3).

16.3.2. Вызов кодов драйвера

Ядро запускает драйвер устройства в нескольких случаях.

- **Настройка.** Ядро обращается к драйверу во время загрузки системы с целью проверки и инициализации устройства.
- **Ввод-вывод.** Подсистема ввода-вывода вызывает драйвер для чтения или записи данных.
- **Управление.** Пользователь может создать управляющий запрос к устройству, например, открытие или закрытие устройства или перемотка ленты в накопителе.
- **Прерывания.** По завершении операции или при изменении состояния устройства генерируется прерывание.

Функции настройки устройства отрабатывают только один раз при загрузке системы. Операции ввода-вывода и управления являются синхронными. Они запускаются в ответ на определенные запросы процесса и выполняются в его контексте. Процедура strategy блочного драйвера является исключением из правила (см. раздел 16.3.4). Прерывания — это асинхронные события в системе, так как ядро не может заранее знать о времени их возникновения. Их обработка не производится в контексте какого-либо определенного процесса.

Драйвер делится на две части, которые обычно называются *верхней и нижней половинами* (top half и bottom half). Верхняя «половина» драйвера содержит синхронные процедуры, в нижней располагаются процедуры асинхронного типа. Они могут обращаться к адресному пространству и области u вызывающего процесса, и даже умеют переводить процесс в режим сна, если это необходимо. Процедуры нижней «части» выполняются в системном контексте и, как правило, не имеют отношения к текущему процессу. В любом случае такие процедуры не обладают правом доступа к прикладному пространству и области u текущего процесса. Они не должны переходить в режим сна, поскольку это может привести к блокировке другого «невинного» процесса.

Обе составляющие драйвера должны обладать средствами синхронизации своих действий друг с другом. Если объект доступен обеим «половинам», то процедуры верхнего уровня должны произвести блокировку прерываний (повысить уровень ipl) при манипуляции таким объектом. В ином случае устройство может создать прерывание в тот момент, когда объект на-

ходится во внутренне противоречивом состоянии, что приведет к неожиданным результатам.

Кроме обращения ядра к драйверу, последний умеет сам загружать функции ядра для выполнения таких задач, как управление буфером, доступом или диспетчеризации событий. Эта часть интерфейса взаимодействия драйвера с ядром описана в разделе 16.7. Но сначала вы должны узнать о специфике интерфейсов блочных и символьных устройств.

16.3.3. Переключатели устройств

Переключатель устройств (device switch) представляет собой структуру данных, определяющую точки вхождения, поддерживаемые каждым устройством. Существует два типа переключателей: структура bdevsw — для блочных устройств и cdevsw — для символьных устройств. Ядро предоставляет отдельный массив для каждого типа переключателей, каждый драйвер имеет вхождение в соответствующем ему массиве. Если драйвер использует одновременно блочный и символьный интерфейсы, он будет иметь вхождения в обоих массивах.

Листинг 16.1 показывает типичные структуры данных переключателя.

Листинг 16.1. Блочные и символьные переключатели устройств

```
struct bdevsw {
   int (*d_open)();
   int (*d_close)();
   int (*d_strategy)();
   int (*d_size)();
   int (*d_xhalt)();
   ...
} bdevsw[];

struct cdevsw {
   int (*d_open)();
   int (*d_close)();
   int (*d_read)();
   int (*d_write)();
   int (*d_ioctl)();
   int (*d_mmap)();
   int (*d_segmap)();
   int (*d_xpoll)();
   int (*d_xhalt)();
   struct streamtab* d_str;
} cdevsw[];
```

Переключатель определяет абстрактный интерфейс доступа к устройствам. Каждый драйвер предоставляет специфическую реализацию этих функций. Подробное описание входных точек будет приведено в следующем разделе. Если ядру необходимо, чтобы устройство произвело какое-либо действие,

оно находит необходимый драйвер в таблице переключателя и вызывает соответствующую функцию драйвера. Например, для чтения данных с символьного устройства ядро загружает функцию d_read() соответствующего устройства. Если это драйвер терминала, то вызываемая функция называется ttread(). Более подробно см. об этом в разделе 16.4.6.

Драйверы устройств следуют стандартному соглашению об именовании функции переключателей. Каждый драйвер использует двухбуквенную аббревиатуру для описания самого себя. Эти буквы становятся префиксом всех функций драйвера. Например, дисковый драйвер может использовать префикс dk и именовать свои процедуры как dkopen(), dkclose(), dkstrategy() и dksize().

Устройство может поддерживать не все точки входа. Например, линейный принтер обычно не позволяет операцию чтения. Для таких точек входа драйвер может использовать общую процедуру nodev(), просто производящую выход с кодом ошибки ENODEV. Для некоторых точек входа драйвер вправе не определять никаких действий. Например, многие устройства не производят никаких действий при закрытии. Для их драйверов подходит процедура общего применения nulldev(), которая просто производит выход с кодом 0 (нормальное завершение операции).

Как уже упоминалось ранее, драйверы STREAMS обычно трактуются и доступны как драйверы символьных устройств. Они идентифицируются полем d_str, значение которого для обычных символьных устройств равняется NULL. Для драйверов STREAMS это поле указывает на структуру streamtab, содержащую указатели на специфичные для STREAMS данные и функции. Более подробно читайте о STREAMS в главе 17.

16.3.4. Входные точки драйвера

В этом разделе приводится описание функций драйвера, доступных через переключатель.

d_open()	Вызывается каждый раз при открытии устройства, может сделать устройство активным или инициализировать структуры данных. Устройства, требующие эксклюзивного доступа (такие как принтеры или накопители на магнитных лентах) могут устанавливать флаг при открытии и сбрасывать его при закрытии. Если флаг оказывается уже установленным, d_open() блокируется или завершается с ошибкой. Процедура является общей для символьных и блочных устройств
d_close()	Вызывается в том случае, если освобождается последняя ссылка на устройство, то есть не остается ни одного процесса, для которого это устройство открыто. Процедура вправе завершить работу устройства или перевести его в автономный режим. Накопитель на магнитных лентах может перемотать пленку. Процедура является общей для символьных и блочных устройств

d_strategy()	Общая точка входа для запросов на чтение и запись с устройства. Процедура получила такое название из-за возможности драйвера определять некоторую стратегию перераспределения ожидающих запросов с целью повышения производительности. Работает асинхронно; если устройство занято, процедура просто помещает запрос в очередь и завершает работу. После завершения ввода-вывода обработчик прерываний удаляет из очереди следующий запрос и начинает новую операцию ввода-вывода
d_size()	Используется дисковыми устройствами для определения размеров дисковых разделов
d_read()	Считывает данные с символьного устройства
d_write()	Записывает данные в символьное устройство
d_ioctl()	Общая точка входа управляющих операций символьных устройств. Каждый драйвер может определить набор команд, загружаемых посредством интерфейса ioctl. Одним из параметров таких функций является cmd, целое число, соответствующее выполняемой команде. Параметр arg указывает на специфический для команды набор аргументов. Это удобная точка входа, поддерживающая богатый набор операций с устройством
d_segmap()	Отображает память устройства в адресном пространстве процесса. Используется символьными устройствами, отображаемыми в память при применении системного вызова mmap
d_mmap()	Применяется только в случае поддержки процедуры d_segmap(). Если d_segmap равняется NULL, системный вызов mmap по отношению к символьному устройству вызывает операцию spec_segmap(), которая, в свою очередь, вызывает d_mmap(). Проверяет корректность указанного смещения устройства и завершает работу с возвратом соответствующего виртуального адреса
d_xpoll()	Опрашивает устройство с целью проверки наступления определенного события. Может использоваться для проверки готовности устройства для чтения или записи без блокировки, если произошла ошибка и т. д.
d_xhalt()	Завершает работу устройства, управляемого драйвером. Вызывается при завершении работы системы либо при выгрузке драйвера из ядра

Структуры переключателя немного отличаются друг от друга в различных реализациях UNIX. Например, в некоторых вариантах переключатель блочного устройства содержит дополнительные функции, такие как d_ioctl(), d_read() и d_write(). Другие версии могут поддерживать функции инициализации или реакции на сброс шины.

Все перечисленные процедуры (кроме d_xhalt() и d_strategy()) принадлежат к верхнему уровню. Процедура d_xhalt() вызывается при завершении работы, вследствие чего не может воспользоваться контекстом прикладного процесса и даже прерываниями. В любом случае эта процедура не переводится в режим сна.

Операция d_strategy() является специализированной сразу же по нескольким причинам. Она часто запускается с целью чтения или записи буферов,

не относящихся к вызывающему ее процессу. Например, процесс пытается получить свободный буфер. Для этой цели он находит в списке свободных буферов первый по счету «грязный» буфер и вызывает процедуру strategy для выгрузки его содержимого на диск. После вызова операции записи процесс запрашивает следующий буфер (считая, что он свободен) и начинает его использовать. В дальнейшем процесс не интересуется состоянием записываемого буфера и не ожидает окончания операции записи. Более того, дисковые операции ввода-вывода часто являются несинхронными (как в только что приведенном примере) и драйвер не должен блокировать вызывающий процесс.

Из вышесказанного можно заключить, что операция d_strategy() интерпретируется как процедура нижнего уровня. Она инициализирует операцию ввода-вывода и завершает работу, не ожидая окончания ввода-вывода. Если при получении запроса устройство окажется занято, d_strategy() просто добавит этот запрос во внутреннюю очередь устройства и завершит выполнение. Через некоторое время из кода прерывания будут вызваны другие процедуры нижнего уровня, которые удалят запрос из очереди и выполнят его. Если вызывающему процессу необходимо ждать завершения ввода-вывода, он производит это вне рамок процедуры d_strategy().

Точки входа драйвера для обработки прерываний и инициализации, как правило, недоступны посредством таблицы переключателя. Вместо этого они указываются в главном файле конфигурации, используемом при сборке ядра системы. Такой файл содержит вхождения для каждого контроллера и драйвера. Во вхождении отражается также такая информация, как уровни ipl, номер вектора прерываний и базовый адрес драйвера CSR. Специфика содержимого и формата файла зависят от реализации UNIX.

В системе SVR4 определено две процедуры инициализации для каждого драйвера — init и start. Каждый драйвер регистрирует эти процедуры в массивах io_init[] и io_start[] соответственно. Код первоначальной загрузки системы загружает все функции init до инициализации ядра и все функции start после завершения его инициализации.

16.4. Подсистема ввода-вывода

Подсистема ввода-вывода представляет собой часть ядра, управляющую аппаратно-независимой частью операций ввода-вывода и взаимодействующую с драйверами устройств для обработки аппаратно-зависимых операций. Подсистема ввода-вывода также отвечает за именование устройств и их защиту, предоставляя прикладным приложениям единый интерфейс доступа ко всем устройствам системы.

16.4.1. Старший и младший номера устройств

Идентификация и обращение к устройствам определяется *пространством имен* устройств. В системе UNIX существуют три различных пространства имен устройств. Аппаратное пространство идентифицирует устройства по контроллерам, к которым они присоединены, а также логическому номеру устройства в контроллере. Ядро применяет для именования устройств их нумерацию. Пользователям же необходимо предоставить простое и понятное пространство имен. Для этой цели используются полные имена файлов. Подсистема ввода-вывода определяет семантику пространств имен для ядра и пользователя и осуществляет связь между ними.

Ядро идентифицирует каждое устройство по *типу* (блочное или символьное), а также по паре номеров, получивших название старшего и младшего номера устройства. *Старший номер устройства* идентифицирует тип устройства или, что будет более точным, его драйвер. *Младший номер устройства* идентифицирует определенный экземпляр устройства. Например, все диски могут иметь старший номер 5, но каждый из них в отдельности будет обладать индивидуальным младшим номером. Блочные и символьные устройства имеют отдельные независимые наборы старших номеров. Таким образом, номер 5 блочного устройства может указывать на дисковый драйвер, в то время как тот же номер для символьного устройства принадлежит линейным принтерам.

Старший номер является индексом к таблице соответствующего переключателя устройств. Если в рассматриваемом выше примере ядру необходимо загрузить операцию open дискового драйвера, то для этого оно находит в массиве bdevsw[] вхождение 5 и вызывает определенную для него функцию d_open(). Как правило, старший и младший номера устройства комбинируются в единой переменной типа dev_t. Старшие биты этой переменной содержат старший номер, а младшие — младший соответственно. Для выделения номера применяются макроопределения getmajor() и getminor(). Для рассматриваемого примера исходный код открытия дискового драйвера выглядит примерно так:

```
(*bdevsw[getmajor(dev).d_open])(dev, ...);
```

Ядро передает процедуре d_open() номер устройства в качестве входного параметра. Драйвер устройства поддерживает внутренние таблицы, при помощи которых преобразует младший номер устройства в определенные регистры CSR или номера порта контроллера. Драйвер извлекает младший номер из dev и использует его для обращения к необходимому устройству.

Существует возможность настройки нескольких старших номеров для одного драйвера устройства. Такая возможность оказывается удобной, если драйвер обслуживает сразу несколько устройств различного типа, производящих некоторые общие действия. Например, драйвер накопителя на магнитных лентах может использовать один старший номер для выбора режима

автоматической перемотки и другой — для неавтоматического режима. Если устройство одновременно имеет символьный и блочный интерфейс доступа, то оно использует отдельные вхождения в обеих таблицах переключателя и, следовательно, обладает как минимум двумя старшими номерами.

В ранних реализациях UNIX переменная типа dev_t имела 16-разрядное поле, каждые 8 битов которого указывали номер устройства. Это являлось причиной ограничения максимального количества устройств младшего типа в драйвере старшего типа до 256 единиц, что представляет значительное неудобство для некоторых систем. В дополнение к этому драйверы могли использовать несколько старших номеров устройств, соответствующих одному и тому же устройству. Драйверы также использовали сразу несколько имен старших устройств при управлении устройствами различного типа.

Еще одной проблемой является рост таблиц переключателя, которые могут иметь большой объем в том случае, если содержат все возможные вхождения, включающие описания устройств, не подключенных к системе или с незагруженными для них драйверами. Эта проблема возникает из-за того, что производителям драйверов нет нужды в настройке таблицы переключателя для каждой возможной конфигурации их устройств, вследствие чего в таблицу попадает вся информация.

В системе SVR4 было произведено некоторое количество изменений в целью решения вышеописанной проблемы. Тип dev_t является 32-разрядным, первые 14 битов используются для старшего номера, остальные 18 — для младшего. В SVR4 также было представлено понятие внутренних и внешних номеров устройств. *Внутренние номера устройств* идентифицируют драйвер и хранятся в виде индексов для доступа к переключателю. *Внешние номера устройств* формируют видимое пользователем представление устройств и хранятся в поле i_rdev индексного дескриптора специального *файла устройства* (см. раздел 16.4.2).

Во многих системах, в том числе и архитектуре Intel 80x86, внутренние и внешние номера устройств одинаковы. В системах, поддерживающих автонастройку, таких как AT&T 3B2, номера отличаются друг от друга. В таких системах массивы bdevsw[] и cdevsw[] формируются динамически при загрузке системы и содержат вхождения только для устройств, сконфигурированных в системе. Ядро поддерживает массив под названием MAJOR[], индексируемый по старшим внешним номерам устройств. Каждый элемент массива хранит соответствующий старший внутренний номер.

Связь между внешними и внутренними номерами осуществляется по типу «много к одному». Для преобразования номеров ядром предусмотрены два макроопределения etoimajor() и itoemagor(). Макрос itoemajor() должен применяться последовательно для создания всех возможных старших номеров. То же самое и для младших номеров. Например, если драйвер имеет два внешних старших номера с 8 устройствами для каждого номера, они будут

проецироваться на внутренние младшие номера в диапазоне от 0 до 15 для одного внутреннего старшего номера.

Макросы getmajor() и getminor() возвращают внутренние номера устройств. Для получения внешних номеров применяются getemajor() и geteminor().

16.4.2. Файлы устройств

Пара <major, minor> (младший номер, старший номер) является простым и эффективным методом представления пространства имен устройств для ядра системы. Однако на прикладном уровне такой подход не слишком удобен, поскольку пользователи системы не хотят запоминать пары номеров, обозначающие каждое устройство. Более важен тот факт, что пользователи желают применять одни и те же приложения и команды для чтения и записи данных как в обычные файлы, так и в устройства. Наиболее очевидным решением является такое пространство имен, в котором устройства представлены как обычные файлы.

Система UNIX поддерживает единый интерфейс доступа к файлам и устройствам при помощи такого понятия, как *файл устройства* (device file). Это специальный файл, не располагаемый в каком-либо произвольном месте файловой системы и ассоциируемый с определенным устройством. По соглашению, все файлы устройств находятся в каталоге /dev или его подкаталогах.

С точки зрения пользователя файл устройства ничем не отличается от обычного файла. Пользователь может открывать или закрывать такой файл, читать или писать данные и даже производить позиционирование по заданному смещению (однако, последнее поддерживается лишь небольшим количеством устройств). Командный интерпретатор умеет перенаправлять потоки stdin, stdout или sttderr в файл устройства. Их содержимое преобразуется в определенные действия устройства, представленного собственным файлом устройства. Например, запись данных в файл /dev/lpr приводит к печати этих данных на линейный принтер.

«Изнутри» файл устройства существенно отличается от обычного файла. Он не имеет блоков данных, хранимых на диске, но обладает постоянным индексным дескриптором в той файловой системе, где он расположен (обычно в корневой файловой системе). Поле di_mode индексного дескриптора файла устройства показывает, что такой файл имеет тип либо IFBLK (для блочных устройств), либо IFCHR (для символьных устройств). Вместо списка номеров блоков индексный дескриптор содержит поле di_rdev, в котором хранятся старшие и младшие номера представляемых этим файлом устройств. Это позволяет ядру осуществить преобразование прикладного номера устройства (пути) в его внутренний номер (пару <minor, major>). Механизм преобразования будет описан в следующем разделе.

Файл устройства не может быть создан традиционным способом. Файлы такого типа вправе быть объявлены только суперпользователями, для чего применяется следующий привилегированный системный вызов:

```
mknod (path, mode, dev);
```

где path — полное имя специального файла, параметр mode указывает на тип этого файла (IFBLK или IFCHR) и привилегии, а dev представляет собой комбинацию из старшего и младшего номеров устройств. Вызов mknod создает специальный файл и инициализирует поля di_mode и di_rdev индексного дескриптора из входных параметров.

Унификация пространств имен файлов и устройств дала массу преимуществ. Для ввода-вывода устройств используется тот же набор системных вызовов, что и для ввода-ввода файлов. Программисты могут создавать приложения, не вникая в подробности, является ли ввод или вывод программы обычным файлом или устройством. Пользователи видят систему единообразно, и для обращения к устройствам вправе использовать понятные символьные обозначения.

Еще одним важным преимуществом единого пространства имен является управление доступом и защита. Некоторые операционные системы, такие как DOS, предоставляют неограниченный доступ к устройствам всем пользователям, в то время как операционные системы мэйнфреймов не разрешают прямой доступ к устройствам. Ни одна из этих схем не является приемлемой. Унификация пространств имен файловой системы и устройств в UNIX позволила расширить механизм защиты файлов на устройства. В каждом файле устройства установлены права доступа на «чтение/запись/выполнение» для владельца, группы и остальных. Эти права инициализируются и изменяются стандартными методами (как для обычных файлов). Традиционно некоторые устройства, в том числе диски, могут быть доступны напрямую только привилегированному пользователю, в то время как другие устройства, накопители на магнитных лентах, доступны всем пользователям.

16.4.3. Файловая система specfs

Современные системы UNIX включают в себя форму интерфейса vnode/vfs [7], позволяющего сосуществование в одном ядре нескольких файловых систем. С каждым открытым файлом ассоциируется объект ядра под названием *vnode*. Интерфейс определяет для каждого объекта vnode набор абстрактных операций. В каждой файловой системе поддерживается собственная реализация этих функций. Поле v_op объекта vnode указывает на вектор указателей таких функций. Например, объект vnode файла ufs (файловой системы UNIX) хранит ссылку на вектор ufsops, содержащий указатели к различным функциям ufs, таким как ufslookup(), ufsclose() и ufslink(). Более подробно об интерфейсе vnode/vfs читайте в разделе 8.6.

Таким системам необходим некий специализированный метод управления файлами устройств. Файл устройства находится в корневой файловой системе, которая (с целью упрощения обсуждения) может являться системой ufs. Таким образом, объект vnode для такого файла является объектом ufs и указывает на вектор ufsops. Все операции над файлом могут быть обработаны при помощи функций ufs.

Вместе с тем это не совсем корректный подход. Файл устройства не является обычным файлом системы ufs. Все операции над таким файлом должны привести к соответствующим действиям устройства, производимым обычно через переключатель устройства. В любом случае требуется найти способ отображения всех обращений к файлу на соответствующее ему устройство.

В SVR4 для этой цели используется специализированная файловая система под названием *specfs*. В этой системе все операции vnode реализованы в виде обращения к переключателю устройства и вызова необходимой функции. Объект vnode системы specfs поддерживает закрытую структуру данных, называемую snode (в действительности vnode является частью snode). Подсистема ввода-вывода должна гарантировать, что при открытии файла устройства запрашивается ссылка на объект specfs vnode, и все дальнейшие операции с этим файлом направляются в него.

Для того чтобы проследить работу системы, возьмем в качестве примера открытие пользователем файла /dev/lp. Каталог /dev находится в корневой файловой системе, имеющей тип ufs. Системный вызов open преобразует полное имя посредством нескольких вызовов функции ufs_lookup(), сначала с целью обнаружения vnode для dev и затем — для lp. После этого функция ufs_lookup() получает объект vnode для lp и обнаруживает, что тип рассматриваемого файла — IFCHR. Затем функция выделяет из индексного дескриптора старший и младший номера устройства и передает их процедуре specvp().

В файловой системе specfs все объекты snode поддерживаются в таблице хэширования, формируемой на основе номеров устройств. Процедура specvp() производит обращение к таблице хэширования и в случае отсутствия snode создает новые объекты snode и vnode. Объект snode содержит поле s_realvp, в котором функция specvp() сохраняет указатель на vnode файла /dev/lp. Поле v_op указывает на вектор операций specfs (таких как spec_read() и spec_write()), которые, в свою очередь, вызывают операции, являющиеся входными точками драйвера. Результирующая конфигурация представлена на рис. 16.3.

Перед завершением выполнения функция open запускает операцию VOP_OPEN объекта vnode, которая вызывает spec_open() в случае файла устройства. Функция spec_open() инициирует выполнение процедуры драйвера d_open(), производящей необходимые действия по открытию устройства. Термин snode относится к теневому объекту (shadow node). Объект системы specfs скрывает «настоящий» объект vnode и перехватывает все операции по отношению к нему.

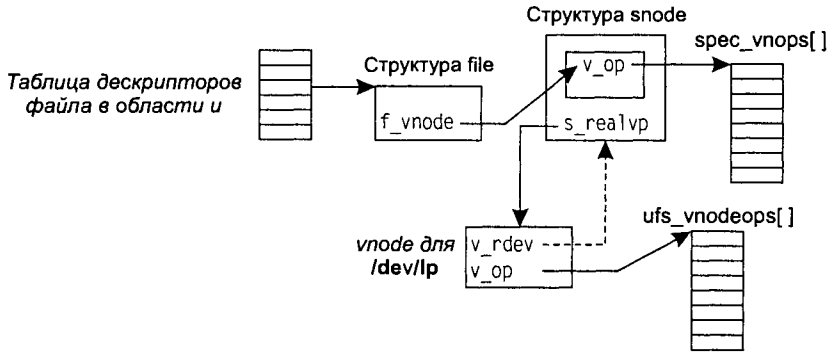


Рис. 16.3. Структуры данных после открытия /dev/lp

16.4.4. Общий объект snode

Описание файловой системы spefs, данное выше, является не совсем полным и нас удовлетворяющим. Оно предполагает, что между файлами устройств и самими устройствами установлена связь «один к одному». На практике это не совсем верно, так как одно устройство может быть представлено сразу несколькими драйверами (поля di_rdev файла устройств имеют одинаковое значение). Такие файлы могут принадлежать как одной, так и нескольким файловым системам.

Этот подход является причиной возникновения нескольких сложностей. Например, операция устройства close должна отработать только после закрытия последнего дескриптора файла устройства. Представьте, что одно и то же устройство открыто двумя процессами, использующими различные файлы. Ядро должно уметь отслеживать подобные ситуации и вызывать операцию close только после закрытия обоих файлов.

Еще одна проблема касается адресации страниц. В системе SVR4 имя страницы в памяти определяется объектом vnode, являющимся владельцем этой страницы, и смещением страницы внутри файла. Для страниц, ассоциированных с устройством (таких как отображаемые в память буферы фреймов или дисковые блоки, предполагающие «неструктурированный» (raw) доступ), такое имя является неоднозначным в том случае, если несколько файлов ссылаются на одно и то же устройство. Два процесса, обращающиеся к устройству через различные файлы, могут создать две копии одной и той же страницы в памяти, что приведет к проблемам корректности.

При присвоении нескольких имен файлов для одного устройства можно разбить все операции над устройствами на две группы. Большинство операций не зависит от имени файла, используемого для обращения к устройству. Такие операции могут быть доступны через общий объект. В то же время существует некоторое количество операций, зависящих от имени файла устройства. Например, каждый такой файл может иметь различных владельцев и неодинаковые права доступа. Для таких операций важно отслеживать дан-

ные «настоящего» объекта vnode файла устройства и производить их перенаправление на такой объект.

В файловой системе specfs есть понятие *общего объекта snode* (common snode) для обоих типов операций. На рис. 16.4 показана схема структур данных этого объекта. Каждое устройство имеет единственный snode, создаваемый при первом обращении к устройству. В системе также поддерживается по одному объекту snode на каждый файл устройства. Объекты всех файлов, представляющих одно и то же устройство, разделяют общий объект snode и ссылаются на него через поле s_commonvp.

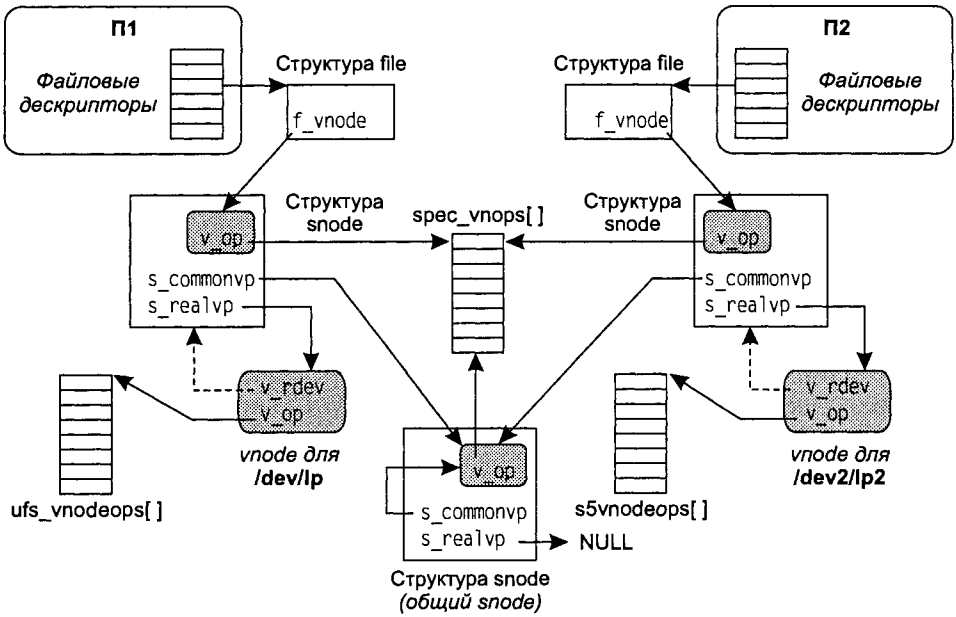


Рис. 16.4. Общий объект snode

При первом открытии файла определенного устройства ядро создает объект snode и общий объект snode. Если через некоторое время файл откроет другой пользователь системы, он будет разделять эти объекты. Если пользователь откроет другой файл, представляющий то же устройство, ядро создаст новый объект snode, который будет иметь ссылку на общий объект snode в поле s_commonvp. Общий объект snode не ассоциируется с файлом устройства напрямую, следовательно, значение его поля s_realvp равно NULL. Поле s_commonvp такого объекта содержит ссылку на самого себя.

16.4.5. Клонирование устройств

Во многих случаях пользователь открывает один из экземпляров устройства, не желая при этом оперировать определенным младшим номером. Например,

сетевые протоколы часто реализуются в виде драйверов устройств. Если несколько активных соединений работают под одним и тем же протоколом, каждое из них имеет различные младшие номера устройства. Для пользователя, желающего создать соединение, безразлично, какой для этого будет выбран младший номер. Для такой цели подходит любой свободный номер устройства. Несмотря на то, что это приведет к циклу обнаружения свободного номера, данная задача не является заботой пользователя. Он может указать системе на необходимость задействования любого подходящего младшего номера, который и будет найден ядром.

Описанная возможность достигается при помощи *клонирования устройств* (device cloning). Если пользователь открывает файл устройства, ядро производит инициализацию snode и вызывает процедуру spec_open(). Эта процедура, в свою очередь, вызывает необходимое для драйвера вхождение d_open() и передает ему указатель на номер устройства, хранимый в поле s_dev объекта snode. Единственной возможностью поддержки клонирования является резервирование одного или нескольких младших номеров специально для этой цели. Если пользователь открывает файл устройства под таким номером, процедура d_open() создает дополнительный младший номер для новой копии устройства. Процедура изменяет поле s_dev объекта snode, записывая в него данные о новом младшем номере (через указатель), и обновляет внутренние структуры данных. В этом случае несколько процессов могут обращаться к устройству посредством указания резервного младшего номера. Драйвер возвратит отличный от имеющихся младший номер (а, следовательно, и другую логическую копию устройства), каждому такому процессу.

В системе SVR4 большинство драйверов, поддерживающих клонирование (например, сетевые протоколы и псевдотерминалы), реализованы как драйверы STREAMS. В SVR4 обеспечена дополнительная функциональность устройств STREAMS при помощи специализированного *драйвера клонирования* (clone driver). Этот драйвер представляет собой набор файлов устройств по одному на каждое устройство STREAMS, поддерживающее клонирование. Старшие номера таких устройств совпадают с номером *устройства клонирования*. Младшие номера устройств клонирования равны старшим номерам реальных устройств.

Приведем пример. Драйвер клонирования имеет старший номер 63. Файл устройства /dev/tcp может представлять все потоки TCP (протокола управления передачей, transmission control protocol). Если старший номер устройства драйвера TCP равен 31, то файл /dev/tcp будет обладать старшим номером 63 и младшим номером 31. Когда пользователь откроет /dev/tcp, ядро выделит для файла объект snode и вызовет spec_open(). Процедура spec_open() запустит операцию d_open драйвера клонирования (реализованную в виде процедуры clnopen()), передав ей в качестве параметра указатель на номер устройства (то есть указатель на поле s_dev объекта snode).

Операция clnopen() выделяет младший номер (в нашем примере — 31) и использует его как индекс к массиву cdevsw[] для обнаружения драйвера TCP.

Затем вызывается операция d_open этого драйвера, ей передается указатель на номер устройства и флаг CLONEOPEN. В рассматриваемом примере эти действия приводят к вызову функции tcpopen(). Если последней предъявляется флаг CLONEOPEN, происходит генерация незанятого младшего номера устройства и запись его в snode. Это дает возможность получить уникальное TCP-соединение без потребности в знании об используемом младшем номере.

16.4.6. Ввод-вывод символьных устройств

Подсистема ввода-вывода играет лишь малую роль в организации ввода-вывода символьных устройств. Большую часть операций выполняет драйвер устройства. Если прикладной процесс первый раз открывает символьное устройство, ядро создает объект snode и общий объект snode для него, а также структуру file, указывающую на snode. При вызове функции read ядро получает дескриптор файла для обращения к структуре file, через которую производит доступ к объекту vnode файла (являющемуся частью snode устройства). Ядро выполняет некоторую проверку, например, на открытие файла только «на чтение». Затем вызывается операция VOP_READ объекта vnode, что в конечном итоге приводит к вызову spec_read().

Функция spec_read() проверяет тип объекта vnode и находит, что речь идет о символьном устройстве. Тогда она просматривает таблицу cdevsw[], взяв в качестве индекса старший номер устройства (хранимый в поле v_rdev). Если устройство относится к STREAMS, для выполнения операции вызывается strread(). Для символьных устройств запускается процедура d_read() соответствующего устройства, которой передается структура uio, содержащая все параметры операции чтения, такие как адрес приемника в прикладном адресном пространстве и число считываемых байтов.

Так как операция d_read() является синхронной, она может заблокировать вызвавший ее процесс, если данные не оказались получены сразу же. После получения данных обработчик прерываний будит процесс для копирования байтов в его адресное пространство. Для этой цели d_read() вызывает функцию ядра uiomove(). Эта функция проверяет наличие прав пользователя на запись данных в указываемое место. Если этого не сделать, то пользователь по неосторожности или, наоборот, злоумышленно может перезаписать текстовый сегмент или даже адресное пространство ядра. После завершения передачи данных ядро возвращает количество считанных байтов.

16.5. Системный вызов poll

Системный вызов poll позволяет объединять ввод-вывод с набора дескрипторов. Представьте серверную программу, открывающую несколько сетевых соединений, каждое из которых представлено файлом устройства. Это требует

использования уникального дескриптора для каждого соединения. Если программе нужно получить входящее сообщение от конкретного соединения, вызывается операция чтения, где в качестве параметра указывается соответствующий дескриптор. Системный вызов read заблокирует сервер до тех пор, пока не получит ожидаемые данные, после чего возобновит работу и передаст данные программе.

Разберем другой вариант развития примера: сервер теперь будет ожидать сообщение, поступившее с любого соединения. В этом случае вызов read не будет эффективен, так как чтение одного дескриптора может привести к блокировке программы, несмотря на то, что на других соединениях имеются данные. В такой ситуации серверу необходимо применять системный вызов poll, позволяющий одновременно ожидать событий в заданном диапазоне дескрипторов и завершать выполнение после наступления такого события. Синтаксис вызова следующий:

```
poll (fds, nfds, timeout);
```

где параметр fds указывает на массив размера nfds, имеющий следующую структуру:

```
struct pollfd {
   int fd; /* дескриптор файла */
   short events; /* интересующие события */
   short revents; /* возвращаемые события */
}
```

Для каждого дескриптора поле events ссылается на события, интересные для вызывающего процесса. Параметр revents относится к событиям, которые реально имели место. Обе переменные представляют собой битовые маски. Типы предопределенных событий включают в себя POLLIN (данные можно прочесть без блокировки), POLLOUT (данные могут быть записаны без блокировки), POLLERR (в устройстве или потоке произошла ошибка), POLLHUP (разрыв связи в потоке — hang-up). Следовательно, в стандартном варианте вызов poll производит проверку готовности устройства на ввод-вывод или получает состояние ошибки.

Системный вызов poll проверяет все указанные в параметрах дескрипторы. Если происходит любое событие из входящих в список интересных, вызов немедленно завершает выполнение после просмотра всех дескрипторов. В противоположном случае процесс блокируется до тех пор, пока не наступит одно из ожидаемых событий. На выходе из функции поле revents каждой структуры pollfd будет содержать событие, произошедшее для указанного дескриптора (если таковое имело место). Функция poll также завершает выполнение после промежутка времени, равного параметру timeout (время указывается в миллисекундах). Если параметр timeout равен 0, функция poll прекращает работу незамедлительно. Если timeout равняется INFTIM или -1, функция закончит работу только после возникновения интересующего события (либо при пре-

рывании системного вызова). Возвращаемое значение poll описывает число произошедших событий, 0 — если имел место выход по исчерпанию времени или -1, если выполнение завершилось неудачно по другой причине.

В приведенном выше примере сервер может вызвать функцию poll, указав флаг POLLIN для каждого дескриптора. Если вызов возвратит число больше 0, сервер сделает вывод о наличии хотя бы одного сообщения на каком-то соединении и проверит структуры pollfd. Затем сервер прочтет сообщение из соответствующего дескриптора, обработает его и повторно вызовет poll для нового опроса.

16.5.1. Реализация вызова poll

Несмотря на то, что дескрипторы, передаваемые системному вызову poll, могут ссылаться на любые файлы, как правило, он используется по отношению к файлам символьных или STREAMS-устройств. По этой причине описание работы вызова будет сфокусировано именно на вышеперечисленных типах файлов. Одним из интересных применений poll является блокировка процесса таким образом, чтобы его пробуждение произошло после наступления одного или нескольких определенных событий. Для этого ядро использует две структуры данных, pollhead и polldat. Структура pollhead ассоциируется с файлом устройства и содержит очередь структур polldat. Каждая структура polldat идентифицирует заблокированный процесс, а также события, по которым он был заблокирован. Процесс, блокируемый по нескольким устройствам, обладает набором структур polldat по одной на каждое устройство. Они взаимосвязаны между собой, как это показано на рис. 16.5.

Системный вызов poll просматривает указанные ему дескрипторы в цикле и запускает операции VOP_POLL ассоциированных с этими дескрипторами объектов vnode. Синтаксис их вызова следующий:

```
error = VOP_POLL (vp, events, anyyet, &revents, &php);
```

где параметр vp является указателем на vnode, events — битовая маска опрашиваемых событий, а enyyet — количество ожидаемых событий, уже обнаруженных системным вызовом poll по отношению к другим дескрипторам. Операция возвращает в revents набор произошедших событий, а параметр php содержит указатель на структуру pollhead.

В случае символьного устройства операция VOP_POLL реализована в виде процедуры spec_poll(), которая производит обращение к таблице cdevsw[] и вызывает процедуру драйвера d_xpoll(), осуществляющую проверку, не ожидает ли устройство указанного события. В этом случае происходит обновление маски revents и выход. Если устройство не ожидает каких-либо событий и параметр anyyet равняется нулю, процедура возвращает указатель на структуру pollhead устройства. Драйверы символьных устройств, как правило, выделяют структуру pollhead для каждого обслуживаемого ими младшего номера устройства.

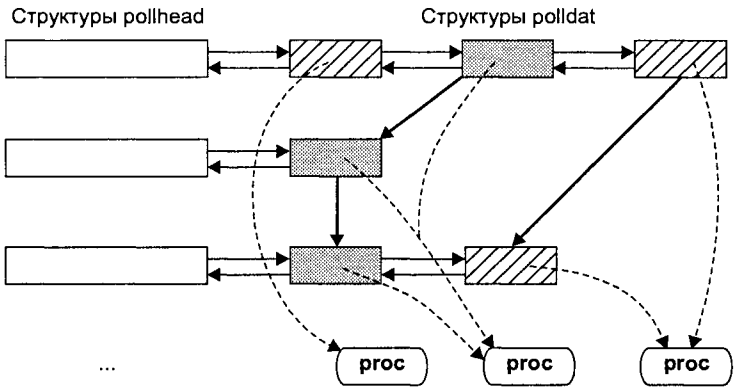


Рис. 16.5. Реализация вызова poll

После возврата функции VOP_POLL системный вызов poll проверяет значения переменных revents и anyyet. Если обе переменные равны нулю, ни одно интересующее событие не ожидается устройством. В этом случае вызов poll получает из параметра php указатель на структуру poolhead, размещает структуру polldat и добавляет ее в очередь pollhead. Он сохраняет в структуре polldat указатель на структуру proc и маску событий устройства, а также соединяет ее с другими структурами polldat процесса.

Если устройство возвращает ненулевое значение переменной revents , это означает, что событие уже ожидается устройством и вызову poll нет необходимости блокировать процесс. В этом случае функция удаляет все структуры polldat из очередей pollhead и освобождает их. Затем она увеличивает значение anyyet на количество событий, установленных в revents. При опросе следующего устройства драйвер обнаружит, что параметр anyyet отличен от нуля и не станет инициировать структуру pollhead.

Если ни одно из указанных событий не ожидается ни одним устройством, вызов poll блокирует процесс. Информация об ожидаемых событиях при этом поддерживается драйверами. Когда происходит одно из таких событий, драйвер вызывает функцию pollwakeup(), передает ей событие и указатель на структуру pollhead устройства. Вызов просматривает очередь polldat структуры pollhead и будит все процессы, находящиеся в режиме ожидания события. Для каждого процесса также просматривается цепочка polldat с целью удаления и освобождения структур polldat из очереди pollhead.

Операция опроса должна быть реализована в каждой операционной системе или устройстве. Некоторые файловые системы, такие как ufs или s5fs, используют для этой цели процедуру ядра fs_poll(), которая производит копирование флагов из параметра events в revents и после этого завершает свою работу. Это приводит к возврату poll без блокировки процессов. Такой подход, как правило, применяется блочными устройствами. Устройства STRAMS используют для поставленной цели процедуру strpoll(), опрашивающую любой обычный поток.

16.5.2. Системный вызов select ОС 4.3BSD

В 4.3BSD реализован системный вызов select, сходный по своим возможностям с poll. Вызов select может применяться для ожидания нескольких событий. Синтаксис операции приведен ниже:

```
select (nfds, readfds, writefds, exceptfds, timeout);
```

где readfds, writefds и exceptfds — указатели на наборы дескрипторов, устанавливаемых для событий чтения, записи и выполнения соответственно. В системе 4.3BSD каждый набор дескрипторов является массивом целых чисел размера nfds. Ненулевые элементы массива определяют дескрипторы, за событиями которых наблюдает вызывающий процесс. Например, процесс может ожидать готовности дескрипторов 2 и 4 начать чтение. Для этого в переменной readfs устанавливаются наборы элементов 2 и 4 и сбрасываются элементы в остальных наборах.

Параметр timeout указывает на структуру timeval, содержащую информацию о максимальном времени ожидания наступления события. Если время ожидания равно 0, вызов завершает работу сразу после проверки дескриптора. Если сам параметр timeout установлен в значение NULL, вызов будет заблокирован до наступления события для указанного дескриптора. При завершении работы select изменяет наборы дескрипторов, предоставляя тем самым информацию о тех из них, где имело место ожидаемое событие. Возвращаемая величина select равняется общему числу готовых к вводу-выводу дескрипторов.

Вызов select поддерживается большинством современных систем UNIX. Он может быть реализован как в виде системного вызова, так и библиотечной процедуры. В различных вариантах UNIX наборы дескрипторов могут выглядеть неодинаково, однако наиболее часто для этого применяется битовая маска. Системы поддерживают приведенный ниже набор POSIX-совместимых макросов для управления наборами дескрипторов, что позволяет скрыть подробности реализации от пользователей системы.

```
FD_SET (fd, fdset); /* добавляет дескриптор fd в набор */
FD_CLR (fd, fdset); /* удаляет дескриптор из fdset */
FD_ISSET (fd, fdset); /* проверяет наличие fd в fdset */
FD_ZERO (fdset); /* очищает все дескрипторы набора fdset */
```

Размер набора дескрипторов, принятый по умолчанию, определяется константой FD_SETSIZE и равняется в большинстве систем (в том числе и в SVR 4) значению 1024.

Реализация вызова select во многом похожа на таковую для poll. Каждый интересующий дескриптор должен указывать на открытый файл, в противоположном случае функция возвращает ошибку. Обычные файлы или блочные устройства всегда готовы для ввода-вывода; таким образом, вызов select

применим только для символьных устройств. Для каждого дескриптора ядро вызывает процедуру d_select (являющуюся частью точки входа d_xpoll системы BSD) соответствующего драйвера символьного устройства с целью проверки готовности устройства дескриптора. Если дескриптор «не готов», процедура запоминает данные о выбранном процессом событии в отношении этого дескриптора. После наступления такого события драйвер должен разбудить процесс, который произведет повторную ревизию всех дескрипторов.

Задача реализации вызова select в системе 4.3BSD усложняется тем, что драйверы могут вести запись данных лишь об одном процессе. Если один и тот же дескриптор выбирается сразу же несколькими процессами, возникает коллизия. Отслеживание и обработка таких ситуаций может привести к дополнительным пробуждениям процессов.

16.6. Блочный ввод-вывод

Ввод-вывод блочных устройств требует большего по сравнению с символьными устройствами операций участия подсистемы ввода-вывода. Блочные устройства делятся на два типа в зависимости от того, содержат ли они файловую систему UNIX или являются неформатированными (иначе «необработанными», raw) устройствами. Устройства второго типа доступны лишь напрямую через файлы устройств. Несмотря на то что и к форматированным устройствам можно обращаться через файлы устройств, как правило, доступ к ним происходит через операции ввода-вывода для размещенных на них файлов[1]. В блочных устройствах может возникать несколько различных событий, в том числе:

- чтение или запись обычного файла;
- прямое чтение из файла устройства и запись в него;
- обращение к отображаемым в память файлам;
- выгрузка-загрузка страниц памяти из устройства свопинга.

На рис. 16.6 показаны различные стадии обработки операции чтения блока (алгоритм записи практически идентичен). Во всех случаях для инициализации чтения ядро системы использует механизм *страничных исключений* (page fault). Обработчик передает страницу из объекта vnode, ассоциированного с блоком. Объект vnode, в свою очередь, вызывает для чтения блока процедуру d_strategy() драйвера устройства.

[1] Вообще говоря, принято разделять все устройства UNIX на блочные (главное достоинство которых — произвольный доступ и буферизация) и неструктурированные (символьные), таким образом, неформатированные устройства скорее относятся к символьному типу, либо нужно говорить об обладании ими символьным интерфейсом. Возможно, это и имел в виду автор. — *Примеч. ред.*

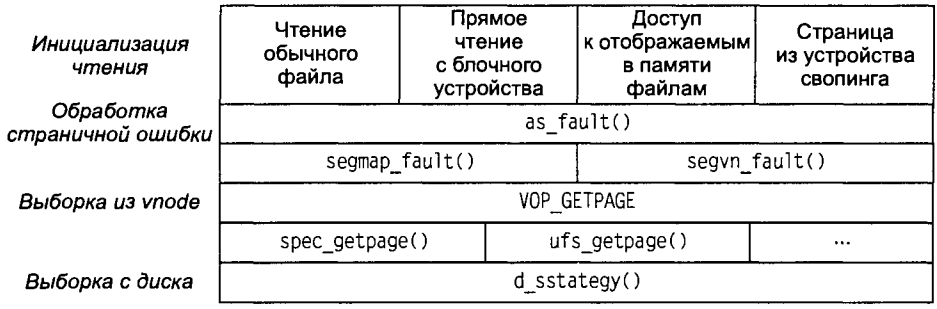


Рис. 16.6. Различные методы инициализации чтения из блочного устройства

Файл может располагаться на разных носителях, например на локальном жестком диске или сменных носителях типа гибких или компакт-дисков, а также на другой машине в сети. В последнем случае ввод-вывод осуществляется с помощью сетевых драйверов, которые часто реализуются в виде устройств STREAMS. В этом разделе будет рассказано только о файлах, размещаемых на локальных жестких дисках. Изложение принципов работы блочных устройств начинается с описания структуры buf, после чего будет представлен анализ различных вариантов осуществления доступа подсистемы ввода-вывода к блочным устройствам.

16.6.1. Структура buf

Структура buf используется как интерфейс взаимодействия между ядром и драйвером блочного устройства. Если ядру необходимо осуществить чтение или запись, вызывается процедура d_strategy() драйвера, которой передается указатель на структуру buf. Эта структура содержит всю информацию, необходимую для операций ввода-вывода, в том числе:

- старший и младший номера устройства;
- номер начального блока данных в устройстве;
- число передаваемых байтов (должно быть кратно размеру сектора);
- месторасположение данных (источника или приемника) в памяти;
- флаги, указывающие на тип операции (запись или чтение), а также на ее синхронность;
- адрес процедуры завершения, вызываемой обработчиком прерывания.

По окончании ввода-вывода обработчик прерывания записывает информацию о состоянии в структуру buf и передает ее процедуре завершения. Данные о состоянии завершения содержат:

- флаги, указывающие на завершение ввода-вывода и успешность операции;
- код ошибки, если операция оказалась неудачной;
- счетчик оставшихся (не переданных) байтов.

Структура buf также требуется буферному кэшу для хранения управляющей информации о кэшируемом блоке. В современных системах UNIX, таких как SVR4, буферный кэш нужен только для блоков метаданных файлов (содержащих индексные дескрипторы или блоки косвенной связи, см. раздел 9.2.2). Он кэширует последние использованные блоки, ожидая, что именно они с наибольшей вероятностью потребуются повторно (см. о принципе *локальности ссылок* в разделе 13.2.6). Структура buf ассоциируется с каждым таким блоком и содержит следующие дополнительные поля, предназначенные для управления кэшированием:

- указатель на vnode файла устройства;
- флаги, указывающие на занятость или «свободность» буфера, а также «грязность» буфера (наличие изменения);
- флаг aged (см. ниже);
- указатели на положение буфера в списке свободных буферов;
- указатели для размещения буфера в таблице хэширования. Аргументом хэш-функции являются vnode и номер блока.

Флаг aged требует более подробного объяснения. При освобождении «грязного» буфера ядро производит его перемещение в конец списка свободных буферов. Постепенно такой буфер оказывается в начале списка, если к нему не произойдет обращение раньше. После того как буфер достигнет начала списка, процесс может попытаться воспользоваться им повторно и увидит, что его содержимое необходимо сначала сбросить на диск. Процесс устанавливает для такого буфера флаг aged и запускает операцию записи. Флаг показывает, что буфер уже находился в списке свободных буферов. Такой буфер должен быть использован повторно раньше других буферов, попавших в данный список впервые, так как к этому буферу не происходило обращения в течение длительного периода времени. Следовательно, после завершения операции записи обработчик прерывания освободит буфер с флагом aged и сразу поместит его в начало списка свободных буферов.

16.6.2. Взаимодействие с объектом vnode

Ядро обращается к дисковому блоку при помощи указания vnode и смещения внутри этого объекта. Если vnode представляет специальный файл устройства, то смещение является физическим и отсчитывается от «начала» устройства. Если объект vnode представляет обычный файл, его смещение является логической величиной и базируется началом файла.

Из этого следует, что обращение к обычному файлу можно произвести двумя методами: либо по его объекту vnode и смещению в файле, либо по объекту vnode устройства и физическому смещению. Второй метод применяется при осуществлении прямого доступа к устройству. Такое обращение может привести к порождению двух псевдонимов одного и того же блока, что

повлечет появление двух копий блока в памяти. Для предупреждения подобных ситуаций прямой доступ к устройству ограничен и разрешен только до монтирования файловой системы.

Так как каждый блок ассоциируется с vnode (файла или устройства), ядро осуществляет все операции блочного ввода-вывода через объекты vnode (кроме случая неформатированного ввода-вывода, о котором можно прочесть в разделе 16.6.4). Объект vnode поддерживает для этой цели две операции, VOP_GETPAGE (получение страницы) и VOP_PUTPAGE (сброс страницы на диск). Вызов этих операций приводит к запуску spec_putpage() и spec_getpage() в случае файлов устройств, функций ufs_putpage() и ufs_getpage() для файла ufs и так далее.

Представленный выше механизм гарантирует сохранение целостности при обращении нескольких процессов к одному и тому же файлу различными способами. В частности, один процесс может отобразить файл в памяти, в то время как другой процесс обращается к этому же файлу через вызовы read и write. Для того чтобы ядро системы обладало непротиворечивым представлением о таком файле, оба метода осуществляют доступ к нему через vnode.

Для примера опишем работу функции ufs_getpage(), которая сначала проверяет наличие страницы в памяти посредством обращения в глобальной таблице хэширования, используя vnode и смещение в качестве аргумента. Если страница не была обнаружена в памяти, функция вызывает процедуру ufs_bmap() для преобразования логического номера блока файла в физический номер блока на диске. Затем функция запрашивает страницу памяти, в которую будет считываться блок данных и ассоциирует с ним структуру buf. Функция получает номер устройства диска из индексного дескриптора (который доступен через объект vnode). На последнем этапе работы происходит вызов процедуры d_strategy() дискового драйвера для непосредственно чтения (процедуре передается указатель на buf) и переход процесса в режим сна до завершения операции. По окончании ввода-вывода обработчик прерываний разбудит процесс. Функция ufs_getpage() осуществляет также некоторые дополнительные действия, такие как упреждающее чтение, если это необходимо.

Если блок не содержит данных файла, то он ассоциирован с объектом vnode файла устройства. В этом случае для чтения блока загружается функция spec_getpage(). Она также производит проверку наличия блока в памяти. Если блок не обнаружен, функция загружает операцию чтения с диска. В отличие от случая обычных файлов, операции spec_getpage() не нужно делать преобразование логического номера блока в физический, так как указываемые номера блоков уже относятся к конкретному устройству.

16.6.3. Способы обращения к устройствам

Как уже говорилось ранее, блочный ввод-вывод осуществляется в целом ряде различных случаев. Обсудим каждый из них более подробно.

Свопинг

Каждая страница в страничной памяти обладает ассоциированной с ней структурой page. Эта структура содержит поля для хранения указателя на vnode и смещение, которые вместе образуют имя страницы. Подсистема виртуальной памяти инициализирует эти поля при первом переносе страницы в память.

Демон pagedaemon периодически выгружает все «грязные» (то есть содержащие изменения) страницы на диск. Он выбирает страницы для записи исходя из правила (см. в разделе 13.5.2 описание алгоритма *давно использованных страниц*), согласно которому наиболее часто используемые страницы остаются в памяти. Существуют и некоторые другие операции ядра, сбрасывающие страницы на диск, такие как выгрузка в область свопинга адресного пространства процесса целиком или вызов функции fsync.

Для сброса страницы на диск ядро получает объект vnode из структуры page и запускает для него операцию VOP_PUTPAGE. Если страница относится к файлу устройства, это приведет к вызову функции spec_putpage(), которая получит номер устройства из объекта vnode и вызовет для этого устройства процедуру d_strategy().

Если нужно сбросить на диск страницу обычного файла, то реализация этой задачи будет зависеть от конкретной файловой системы. Например, для записи на диск страниц файлов ufs применяется функция ufs_putpage(). Она вызывает ufs_bmap() для получения физического номера блока и затем вызывает процедуру d_strategy() устройства (номер которого содержится в индексном дескрипторе, доступном через объект vnode). Функция ufs_putpage() также производит некоторые оптимизирующие действия, такие как *кластеризация* (clustering) (сбор смежных измененных страниц и одновременная их запись в одном запросе ввода-вывода).

Ввод-вывод файлов, отображаемых в память

Процесс может отображать файл или какую-либо его часть в сегмент своего адресного пространства, используя для этой цели системный вызов mmap. При загрузке программы посредством exec происходит отображение ее текстов и данных в адресное пространство процесса. Более подробно о файлах, отображаемых в память, читайте в разделе 14.2. После создания отображения процесс может попытаться прочесть страницу, не находящуюся в памяти (либо не отображенную в *таблице аппаратного преобразования адресов, HAT*). В этом случае произойдет *страничное исключение* (page fault). Так как страницы файлов, отображаемых в памяти, относятся к сегментам vnode (seg_vn), для обработки ошибки вызывается процедура segvn_fault(). Она запускает операцию VOP_GETPAGE объекта vnode файла, имеющего указатель на закрытые структуры данных сегмента.

Точно также при изменении процессом страницы, отображенной в режиме *разделения* (shared mapping), необходимо произвести ее запись в соответствующий файл. Это действие наиболее часто производится при выгрузке страницы процессом pagedaemon, как это описывалось в предыдущем подразделе.

Ввод-вывод обычных файлов

В системе SVR4 чтение и запись обычных файлов реализовано через драйвер seg_map. Например, при вызове системной функции read ядро использует дескриптор файла для определения структуры file, из которой, в свою очередь, получает данные об объекте vnode файла. Затем ядро вызывает операцию VOP_READ объекта vnode, реализованную в виде функции, зависимой от конкретной файловой системы. Для примера выберем файловую систему ufs. В этом случае применяется функция ufs_read(), которая производит чтение в указанной ниже последовательности.

1. Вызывает segmap_getmap() с целью создания отображения ядра для нужной записи (байтовой области) файла. Функция возвращает адрес, по которому произошло отображение данных в пространстве ядра.
2. Вызывает uiomove() для передачи данных файла в прикладное пространство. Адрес источника данных берется из информации, полученной на предыдущем шаге.
3. Вызывает segmap_release() для открепления отображения. Драйвер seg_map кэширует такие отображения в порядке LRU, предполагая, что недавно использованные страницы наиболее вероятно будут запрошены повторно.

Если страница не оказалась в памяти или ядро не обладает корректным преобразованием адреса для нее, функция uiomove() вырабатывает страничную ошибку. Обработчик исключения определяет, что данная страница относится к сегменту seg_map, и вызывает segmap_fault() для загрузки ее в память. Функция segmap_fault() запускает операцию VOP_GETPAGE объекта vnode, которая и считывает страницу с диска, если это необходимо.

Прямой ввод-вывод блочных устройств

Пользователь обладает возможностью прямого обращения к блочным устройствам, если обладает соответствующими привилегиями. Для этого применяются системные вызовы read или write по отношению к файлу устройства. Ядро получает дескриптор файла для доступа к структуре file, из которой выделяет данные об объекте vnode файла. Затем ядро запускает операцию VOP_READ или VOP_WRITE объекта vnode, что в свою очередь приводит к вызову функций spec_read() или spec_write() соответственно. Эти функции производят действия во многом аналогичные алгоритму работы функций чтения и записи файловой системы ufs. Они также вызывают segmap_getmap(), uiomove() и segmap_release(). Следовательно, прямой ввод-вывод приводит к возникновению страничных исключений (и необходимости сброса страниц на диск), как и в предыдущих случаях.

Процесс может произвести отображение блочного устройства в своем адресном пространстве при помощи системного вызова mmap. В этом случае

чтение из отображаемых адресов приведет к страничным исключениям, которые будут обработаны драйвером seg_vn. Процедура segvn_fault() запустит операцию VOP_GETPAGE объекта vnode, что закончится в данном случае вызовом spec_getpage(). Функция spec_getpage() вызовет процедуру d_strategy() устройства, если страница не окажется в памяти. При потребности записи в блочное устройство будет использована функция spec_putpage().

16.6.4. Неформатированный ввод-вывод блочных устройств

При использовании системных вызовов read или write происходит двойное копирование данных: между прикладным пространством и ядром, а также между ядром и диском. При этом ядру необходимо кэшировать данные, что нисколько не тревожит обычные приложения. Однако если приложение производит обмен большими объемами данных между памятью и диском, и его правила работы с памятью не зависят от кэширования, такой подход является неэффективным.

Одной из альтернатив применения read и write является использование вызова mmap для создания отображения данных в адресном пространстве. Несмотря на то, что этот способ позволяет избавиться от одной операции копирования, семантика обращения отличается от принятой в стандартных вызовах чтения и записи. Более того, системная функция mmap является относительно новой и может не поддерживаться всеми реализациями ОС. В системах UNIX имеется альтернативная вызову mmap возможность, получившая название *неформатированного ввода-вывода* (raw I/O), которая позволяет осуществлять небуферизированный доступ к блочным устройствам. Неформатированный ввод-вывод также влечет за собой отказ от дополнительного копирования данных, что существенно способствует общей производительности. Неформатированный ввод-вывод поддерживается многими реализациями систем, в том числе и не обладающими вызовом mmap.

В этом случае блочное устройство должно обладать неформатированным или символьным интерфейсом доступа. Соответственно, такое устройство должно иметь вхождение в переключателе символьных устройств. Неформатированный ввод-вывод осуществляется через вызовы read или write применительно к символьным устройствам, что приводит к вызову d_read() или d_write() устройства. Эти процедуры напрямую вызывают функцию ядра physiock(), работающую по следующей схеме:

1. Функция проверяет параметры ввода-вывода, такие как не превышение «конца» устройства.
2. Выделяет из списка свободных буферов структуру buf.
3. Вызывает as_fault() для загрузки по исключению страниц пользовательского процесса, участвующих в операции.

4. Блокирует страницы прикладного процесса в памяти (делает их невыгружаемыми).
5. Вызывает процедуру d_strategy() ассоциированного блочного устройства. Символьный драйвер передает функции physiock() номер устройства в качестве входного параметра.
6. Переходит в режим сна до момента завершения ввода-вывода.
7. Разблокирует страницы прикладного процесса.
8. Возвращает результат выполнения операции (счетчик переданных байтов, состояние ошибки и т. д.).

16.7. Спецификация DDI/DKI

Несмотря на то, что драйверы устройств являются частью ядра системы, как правило, они создаются производителями оборудования. При этом разработчики могут вообще не обращаться к исходным кодам ядра, поскольку между ядром и драйвером реализован процедурный интерфейс взаимодействия, основанный на переключателях устройств. При разработке драйвера UNIX производитель создает реализацию этого интерфейса, включая функции переключателя, обработчик прерываний, а также функции настройки и инициализации устройства. Вхождения драйвера добавляются в соответствующие конфигурационные файлы системы (такие как файл conf.c, в котором содержатся таблицы bdevsw[], cdevsw[]), после чего производится сборка ядра и связывание драйвера с набором объектных файлов ядра, предоставленных разработчиком операционной системы.

В предыдущих разделах мы делали упор на этой части интерфейса драйверов. Однако наше описание пока является неполным, так как рассказывает только о вызовах, производимых ядром по отношению к драйверам. Драйвер устройства также может производить вызов нескольких функций ядра для доступа к службам передачи данных, выделения и синхронизации памяти и др. Более того, в ядре системы сосуществуют и работают одновременно сразу несколько драйверов, созданных независимо друг от друга, немалую роль в этом играет то, что один драйвер не мешает работе остальных драйверов или ядра системы.

Для решения проблемы урегулирования разногласий множества независимых друг от друга драйверов должен быть четко определен интерфейс взаимодействия между драйвером и ядром. С этой целью в системе SVR4 была представлена реализация *спецификации DDI/DKI* (Device-Driver Interface/Driver-Kernel Interface, интерфейс «устройство-драйвер»/«драйвер-ядро») [12], формализующая все необходимое взаимодействие.

Логически интерфейс разбивается на несколько разделов, организованных по типу UNIX man pages[1].

- **Раздел 1** определяет данные, которые должен содержать драйвер. Метод обращения ядра к этой информации является зависимым от реализации и от поддержки конфигурации устройства ядром.
- **Раздел 2** объявляет процедуры точек входа. Он содержит функции драйвера, используемые в переключателях устройств, а также процедуры обработки прерываний и инициализации.
- **Раздел 3** определяет процедуры ядра, загружаемые драйвером.
- **Раздел 4** описывает структуры данных ядра, используемых драйвером.
- **Раздел 5** содержит определения #define ядра, которые могут потребоваться драйверу.

Функционально интерфейс делится на три части.

- **Драйвер-ядро** (driver-kernel). Самая большая составляющая. Содержит точки входа драйвера и процедуры поддержки ядра.
- **Драйвер-аппаратура** (driver-hardware). Этот узел описывает процедуры взаимодействия между драйвером и устройством. Такие процедуры являются аппаратно-зависимыми, однако большинство из них определяется спецификацией DDI/DKI.
- **Драйвер-загрузка** (driver-boot). Эта связка отвечает за то, каким именно образом драйвер встраивается в ядро системы. Она не описывается в спецификации DDI/DKI, но присутствует во множестве руководств производителей по программированию устройств (например, [10]).

Стандарт также содержит описание большого числа утилитарных функций общего назначения, предоставляющих такие возможности, как работа с символами и строками. Эти функции не являются частью интерфейса DDI/DKI.

Каждой функции интерфейса присвоен *уровень привязки* (commitment level). Функции первого уровня останутся в следующих версиях спецификации

[1] В каждой системе UNIX имеется интерактивное справочное руководство — manual pages («man» pages), в котором вся информация сгруппирована по разделам (системные вызовы, пользовательские команды, форматы файлов и т. д.). Каждый раздел имеет свой номер. Каждая страница (page) оформлена по единым правилам (включает название, описание, синтаксис, возвращаемые значения, ошибки, предостережения, ограничения, файлы, ссылки и др.). Нужная страница описания выводится по команде «man command». Спецификация DDI/DDK больше не переиздавалась, но и сейчас ее можно приобрести в магазинах e-commerce (Device Driver Reference Unix Svr4.2: Unix Svr4.2; By Hines, Robert M. (editor); Hines, Robert M.; Unix System Laboratories (corporate author); Wilcox, Spence (editor); Published by Prentice Hall (Sd) (Jun 1, 1993); ISBN: 0130426318). Для тех ОС, которые используют принципы CVR4.2, существуют соответствующие руководства. Для Digital UNIX это Writing Device Drivers: Reference; Digital Equipment Corporation, 1996. Подходящая к своей документации с особой тщательностью Sun Microsystems выделяет в каждой версии своих Man pages секцию 9 с описанием команд системного администрирования, в которой содержатся описания процедур и структур DDI/DKI. Более полная версия включается в Device Driver Developers Kit (DDK). — *Примеч. ред.*

DDI/DKI и могут быть изменены только для обратной совместимости. Иначе говоря, программы, созданные с использованием функций первого уровня, будут совместимы с будущими реализациями системы SVR4. Однако поддержка процедур второго уровня ограничена тремя годами. Для каждой такой процедуры указывается дата появления в спецификации. По истечении трех лет процедура может быть изменена любым способом или вообще удалена из спецификации.

Некоторые возможности функций первого уровня определены в документе как принадлежащие ко второму уровню. С течением времени процедура может быть полностью перенесена на второй уровень (например, как это произошло с процедурой rminit(), описание которой находится в разделе 16.7.2)[1]. При этом дата появления процедуры на втором уровне считается датой ее создания. Таким образом, такая функция будет поддерживаться спецификацией еще как минимум три года.

16.7.1. Общие рекомендации

В спецификациях DDI/DKI содержится набор рекомендаций, помогающих разработчикам обеспечивать должный уровень переносимости кодов между реализациями SVR4.

- Драйверы не должны обращаться к системным структурам данных напрямую. В частности, это относится к области u. В ранних версиях UNIX такое обращение часто являлось необходимым и применялось для чтения информации о базовых адресах, счетчиках передаваемых байтов, кодов ошибки или завершения и т. д. Это приводило к тому, что драйвер сильно зависел от структуры области u. При внесении в нее изменений драйвер необходимо было перестраивать заново. В SVR4 произошла кардинальная перекройка взаимодействия драйверов и ядра. Теперь для этой цели могут использоваться структуры данных, определенные в *разделе 4*.
- При обращении к структурам, описанным в разделе 4, драйвер не вправе использовать поля, не описанные в спецификации. Такие поля могут не поддерживаться в последующих версиях стандарта.
- Драйверы не имеют права определять массивы структур раздела 4. Последующие изменения размера структур могут привести к невозможности применения таких массивов в дальнейших реализациях системы. Исключением из этого правила являются структуры iovec и uio.
- Некоторые поля структур являются масками флагов. Драйверам не разрешено передавать таким полям значения напрямую, а только лишь устанавливать или сбрасывать соответствующие флаги. Это правило появилось из-за того, что в определенных реализациях могут оказаться флаги, не предусмотренные спецификацией.

[1] В этом разделе не описание, а отсылка, см. сноску. — *Прим. ред.*

- Структуры данных, недоступные для приложений, не включены в раздел 4, однако могут упоминаться при описании использующих их процедур. Драйверам запрещено обращаться к какой-либо части таких структур. Они могут использовать их исключительно по ссылкам, передавая указатели на эти структуры соответствующим процедурам ядра.
- Для чтения или изменения структур данных, описанных в разделе 4, драйверы должны по мере возможности использовать только функции раздела 3. Такой подход позволяет защитить драйвер от потери совместимости при внесении изменений в структуры данных.
- Драйвер должен содержать файл ddi.h, размещаемый после всех системных *включаемых* файлов, но до специфических для драйвера *включаемых* файлов. Причиной появления данного требования является реализация большого числа функций спецификации в виде макроопределений. Файл ddi.h переопределяет такие макросы, заставляя драйверы использовать не их, а вызовы функций, формирующие соответствующие процедуры. Это позволяет упростить задачу переноса драйвера на другие версии системы. Включение специфических для драйвера файлов после ddi.h гарантирует, что он будет использовать только интерфейс DDI/DKI.
- Все закрытые процедуры и внутренние переменные должны иметь тип static.

16.7.2. Функции раздела 3

В разделе 3 спецификации DDI/DKI описаны функции ядра, используемые драйверами устройств. Процедуры могут быть разделены на несколько функциональных групп.

- **Синхронизация и таймер.** Процедуры sleep() и wakeup() описаны в разделе 2.5.1. Функция delay() блокирует процесс на указанное количество времени. Процедуры timeout() и untimeout() позволяют производить планирование процессов (см. раздел 5.2.1).
- **Управление памятью.** Процедуры kmem_alloc() и kmem_free() отвечают за выделение памяти ядром. Процедуры rminit(), rmalloc() и rmfree() управляют картами ресурсов. Работа этих функций описана в главе 12[1]. Функции physmap() и physmapfree() применяются для отображения физических адресов в виртуальные.

[1] К сожалению, автор привел примеры применения в разделе 12.3 только функций rmalloc и rmfree, но не rminit. Следует восполнить упущение, сказав, что функция rminit инициализирует карту ресурсов и связывает ее с указанным именем. Ее синтаксис следующий: void rminit (map_struct, size, addr, name, mapsize); где map_struct — указатель на инициализируемую структуру, size и addr — количество элементов в карте ресурсов и адрес первого элемента, name — имя карты ресурсов, а mapsize — максимальное количество фрагментов. — *Прим. ред.*

- **Управление буфером.** Процедура getblk() размещает буфер, функция brelse() используется для его освобождения. Драйвер вызывает biowait() для ожидания завершения ввода-вывода. Для пробуждения процесса и освобождения буфера обработчик прерываний вызывает biodone().
- **Операции с номерами устройств.** Функции getmajor() и getminor() выделяют внешний старший и младший номера из dev_t. Функции itoemajor() и etoiminor() применяются для преобразования внутреннего номера во внешний и обратно.
- **Прямой доступ к памяти.** Набор аппаратно-зависимых функций, поддерживающих DMA. Спецификация содержит описания функций для IBM PC-AT-совместимых архитектур.
- **Обмен данными.** Функция uiomove() копирует данные между адресным пространством ядра и пространством прикладного процесса, либо между двумя участками внутри пространства ядра. Эта функция поддерживает ввод-вывод методом сборки-рассоединения (scatter-gather I/O). Например, функция может собирать данные из нескольких прикладных буферов в единый буфер ядра (см. рис. 16.7). Для описания параметров передачи функция использует структуру uio. Процедуры copyin() и copyout() применяются при обмене данными между буфером драйвера и прикладным буфером. Такие аппаратно-зависимые процедуры, как inb() и outb(), перемещают данные из пространства ввода-вывода и обратно на архитектурах, не поддерживающих отображаемый ввод-вывод (например, в Intel x86).

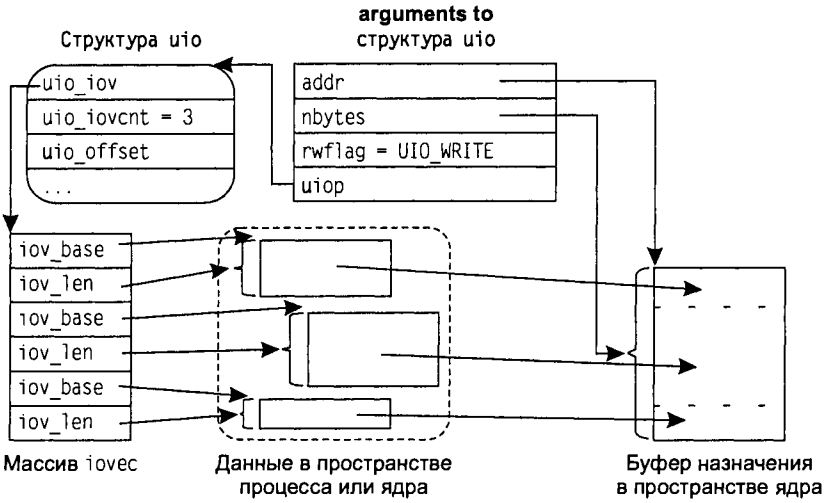


Рис. 16.7. Обмен данными при помощи uiomove()

- **Опрос устройств.** Процедуры опроса устройств включают в себя phalloc() и phfree(), используемые для выделения структур pollhead, и pollwakeup() для пробуждения опрашивающих процессов.
- **STREAMS.** Набор процедур поддержки драйверов устройств STREAMS. О них подробнее читайте в главе 17.
- **Сервисные процедуры.** В интерфейсе DDI/DKI поддерживается набор функций работы со строками, таких как strcopy() и strlen(), манипулирования байтами, таких как bcopy(), bcmp() и bzero(), функции обработки ошибок, такие как ASSERT() и cmn_err(), а также другие полезные функции, например max() и min().

16.7.3. Другие разделы

В разделе 1 говорится о необходимости определения для каждого драйвера префикса, который будет использован во всех его глобальных функциях и структурах данных. Префикс указывается в зависящем от реализации конфигурационном файле и позволяет ядру идентифицировать точки входа драйвера. Например, дисковый драйвер может иметь префикс dk, а его функции будут именоваться соответственно dkopen(), dkclose() и т. д. В этом разделе также указывается на необходимость определения глобальной переменной под названием *prefix*devflag (где *prefix* — это префикс драйвера) и описания флагов, которые могут быть установлены в этой переменной. Ниже перечислены некоторые возможные флаги.

D_DMA	Драйвер поддерживает прямой доступ к памяти
D_TAPE	Драйвер управляет накопителем на магнитных лентах
D_NOBRKUP	Драйвер понимает списки страниц, следовательно, ядру не нужно разбивать операцию передачи нескольких страниц на несколько запросов

Дополнительно рассматривается структура *prefix*info, которая требуется для драйверов STREAMS.

Раздел 2 описывает точки входа драйвера (о которых можно прочесть в начале этой главы). Сюда входят все функции переключателя, а также процедуры обработки прерываний и функции инициализации *prefix*init() и *prefix*start().

Раздел 4 определяет структуры данных, разделяемые между ядром и драйверами. Эти структуры включают в себя buf (см. раздел 16.6.1), структуры uio и iovec (см. раздел 8.2.5). Остальные структуры используются библиотекой STREAMS или аппаратно-зависимым интерфейсом прямого доступа к памяти.

Раздел 5 содержит определения #define, необходимые для ядра системы. Они включают значения кодов ошибок errno, сообщения STREAMS и номера сигналов.

16.8. Поздние версии SVR4

Изначальная реализация системы SVR4 претерпела множество изменений и модификаций, что привело к появлению набора новых возможностей. В систему SVR4/MP встроена поддержка многопроцессорных машин, в SVR4.1/ES добавлены расширенные средства защиты, в SVR4.2 появилась поддержка динамической загрузки драйверов[1]. Все перечисленные средства повлияли на интерфейс взаимодействия между драйвером и ядром, что стало причиной предъявления дополнительных требований к разработчикам драйверов.

16.8.1. Драйверы для многопроцессорных систем

Обычные однонитевые драйверы не могут корректно работать в ядре, поддерживающем симметричную многопроцессорную обработку (SMP). Эти драйверы рассчитаны на то, что они обладают эксклюзивным доступом к структурам данных. Защита таких структур построена на блокировке прерываний, что было эффективно только на однопроцессорных машинах [6]. Для нормальной работы в многопроцессорных средах необходима тщательная переработка однонитевых драйверов. Большинство глобальных данных необходимо защищать при помощи многопроцессорных примитивов синхронизации (МП-защита). Возможно, что потребуется защищать и регистры устройств, так как они могут оказаться доступными одновременно другим нитям драйвера.

Требование поддержки параллельности драйверов ничем не отличается от требований к остальной части ядра. Более подробно о параллельной работе читайте в разделе 7.10. Для реализации параллельности ядру необходимо экспортировать средства многопроцессорной синхронизации драйверам устройств. Кроме этого, возможно, потребуется изменить или полностью заменить функции, не обладающие МП-защитой, либо как-то ограничить их применение.

В системе SVR4/MP произведены некоторые изменения интерфейса DDI/DKI. Во-первых, в него добавлен набор функций, позволяющий создателям драйверов использовать новые возможности синхронизации системы. В SVR4.2/MP поддерживаются три типа объектов многопроцессорной блокировки: простая блокировка, блокировка чтения-записи и блокировка сна. Эти объекты не рекурсивны, поэтому драйвер может войти в клинч при попытке получения уже удерживаемого объекта блокировки. Если объект простой блокировки или объект блокировки чтения-записи не может быть получен незамедлительно, вызывающий процесс способен перейти в состояние блокировки или ждущего цикла, в зависимости от реализации. В системе SVR4/MP также поддерживаются переменные синхронизации, сходные по функциям с *условными переменными*, описанными в разделе 7.7.

---

[1] Некоторые производители поддерживали динамическую загрузку драйверов и до появления SVR4. Например, корпорация Sun Microsystems добавила такую возможность в SunOS 4.1. Система OSF/1 также обладала средством динамической загрузки драйверов.

В раздел 3 был добавлен набор функций для выделения различных объектов синхронизации и работы с ними. В SVR4/MP также появились некоторые ограничения на уже существующие функции. В большинстве случаев эти ограничения не позволяют вызов функции во время удержания какого-либо одного, либо всех типов объектов блокировки.

В некоторых случаях в SVR4/MP была произведена замена функций на их эквиваленты, поддерживающие многопроцессорную обработку. При этом несколько изменился и интерфейс функций. Например, функция timeout() была заменена функцией itimeout(). Она имеет дополнительный параметр, в котором указывается уровень приоритета, с которым необходимо загрузить функцию. В систему добавлена новая функция dtimeout(), которая вызывает данную функцию на определенном процессоре. Функция timeout() сохранена из соображений обратной совместимости, но перенесена на второй уровень спецификации (дата переноса — 08.10.1991). Следовательно, необходимость поддержки этой функции утратила актуальность 8 октября 1994 года.

В SVR4/MP также был добавлен новый флаг D_MP параметра *prefix*devflag драйвера. Установка этого флага указывает на поддержку драйвером МП-защиты. Если флаг сброшен, ядро обрабатывает все операции такого драйвера последовательно. Например, ядро может ассоциировать с драйвером объект глобальной блокировки и запрашивать его каждый раз при вызове любой процедуры драйвера.

16.8.2. Изменения в SVR4.1/ES

В операционную систему SVR4.1/ES были включены расширенные средства защиты ядра UNIX. Основным изменением интерфейса взаимодействия с драйверами стало добавление трех флагов в переменную *prefix*devflag. Эти флаги перечислены ниже.

D_NOSPECMACDATA	Драйвер не выполняет обязательную проверку управления доступом при передаче данных и не изменяет значение параметра времени доступа в индексных дескрипторах
D_INITPUB	Устройства, управляемые такими драйверами, могут быть доступны непривилегированным процессам. Этот флаг может быть изменен вызовом системы защиты
D_RDWEQ	Обращение к устройству требует неукоснительного следования правилу управления доступом

16.8.3. Динамическая загрузка

В традиционных системах UNIX ядро компилируется и связывается статически. После загрузки системы ядро не может быть изменено (кроме случаев незначительных модификаций, производимых отладчиками). Такой подход

является негибким и непрактичным. Он провоцирует людей подключать все возможные модули и драйверы во время сборки системы, даже если впоследствии они не будут использованы. Это приводит к появлению бесполезной части ядра и увеличению его общего размера. Ядро системы, как правило, не использует страничный механизм для самого себя, вследствие чего занимает большой «кусок» оперативной памяти машины. Все изменения ядра, такие как обновление версии драйвера или добавление нового драйвера, требуют перестройки ядра и перезагрузки системы. Это может быть небольшой проблемой для обычной рабочей станции, но не подходит для крупных коммерческих систем, для которых важна стабильность и непрерывность работы. Это также увеличивает время, затрачиваемое на создание драйверов, так как принуждает к постоянной отладке и перезагрузке драйверов.

Современные системы UNIX поддерживают динамическую загрузку модулей ядра. Это означает, что модуль (набор объектов) может быть добавлен или удален во время функционирования ядра. Для этого необходим загрузчик времени выполнения (runtime loader), который производит окончательное присвоение и связывание адресов при загрузке модуля. Такой подход имеет несколько очевидных преимуществ. Система может быть загружена с ядром небольшого объема, содержащего только самые необходимые модули. Другие модули будут добавляться позже по мере необходимости и удаляться, если они больше не нужны. Для обновления модуля достаточно отключить его текущую версию в ядре, выгрузить его и затем загрузить новую версию. При этом не нужно перезагружать всю систему.

Динамическая загрузка может быть использована для различных типов модулей. Ниже представлены модули, которые удовлетворяют требованиям загрузки времени выполнения в системе SVR4.2:

- ✦ драйверы устройств;
- ✦ главный адаптер шины (Host Bus Adapter)[1] и драйверы контроллеров;
- ✦ модули STREAMS;
- ✦ файловые системы;
- ✦ другие модули, в том числе содержащие общие коды, разделяемые между несколькими динамически связываемыми модулями.

Механизмы динамической загрузки для различных типов модулей сходны между собой, однако в этом разделе книги мы поговорим только о драйверах устройств. Загрузка драйвера в работающее ядро сводится к проведению нескольких действий.

1. Присвоение и связывание идентификаторов (symbols) драйвера. За это отвечает загрузчик времени выполнения.
2. Инициализация драйвера и устройства.

[1] Главный адаптер шины (Host Bus Adapter, HBA), осуществляет взаимодействие устройств SCSI с шиной и центральным процессором. — *Прим. ред.*

3. Добавление драйвера в таблицы переключателя устройств, что дает возможность ядру обращаться к процедурам переключателя.
4. Установка обработчика прерываний, что обеспечивает реагирование драйвера на прерывания устройства.

Выгрузка драйвера также требует принятия некоторых мер, в число которых входит освобождение памяти, используемой драйвером, выполнение операций завершения работы драйвера и устройства, отключение обработчика прерываний и удаление всех ссылок на драйвер из прочей части ядра.

Система SVR4.2 поддерживает набор средств, решающих все перечисленные выше задачи. С появлением этой ОС в спецификацию DDI/DKI были добавлены новые процедуры[1].

prefix_load()

Процедура раздела 2, ее реализация должна быть предоставлена драйвером. Процедура производит инициализацию драйвера, вызывается ядром системы при необходимости загрузки драйвера. Ее задачи сходны с действиями, производимыми start и init, так как эти функции не используются при динамической загрузке драйвера. Процедура выделяет память для закрытых данных и инициализирует различные структуры данных. Затем происходит вызов процедуры mod_drvattach() для установки обработчика прерываний. Последним этапом работы процедуры *prefix*_load() является инициализация всех устройств, ассоциированных с драйвером.

prefix_unload()

Процедура раздела 2 предоставляется драйвером. Загружается ядром для проведения «уборки» драйвера при его выгрузке. Как правило, она производит действия, противоположные задачам процедуры *prefix*_unload(). Она вызывает mod_drvattach() для отключения обработчика прерываний драйвера, освобождает занимаемую им память и производит все необходимые действия по завершению работы драйвера и устройств.

mod_drvattach()

Это процедура раздела 3, поддерживаемая ядром. Выполняет установку обработчика прерываний драйвера и разрешает прерывания от устройств драйвера. Должна вызываться из *prefix*_load() с единственным параметром, содержащим указатель на структуру *prefix*attach_info драйвера. Эта структура определяется и инициализируется инструментами настройки ядра при проведении его конфигурирования. Структура является прозрачной для драйвера, который не имеет права обращаться к любому ее полю.

---

[1] Помните, что во всех процедурах *prefix* заменяется префиксом конкретного драйвера устройства.

mod_drvdettach()

Раздел 3, поддерживается ядром. Запрещает поступления прерываний драйверу и отключает их обработчик. Должна вызываться из *prefix*_unload() с указателем на *prefix*attach_info в качестве входного параметра.

Макроопределения надстроек

Спецификация DDI/DKI поддерживает набор определений, генерирующих коды надстроек (wrapper code) загружаемых модулей. Для каждого типа модулей существует персональный макрос:

MOD_DRV_WRAPPER	для драйверов устройств
MOD_HDRV_WRAPPER	для драйверов главного адаптера шины
MOD_STR_WRAPPER	для модулей STREAMS
MOD_FS_WRAPPER	для файловых систем
MOD_MISC_WRAPPER	для других модулей

Для каждого из них определено пять параметров. Например, синтаксис MOD_DRV_WRAPPER выглядит следующим образом:

```
MOD_DRV_WRAPPER (prefix, load, unload, halt, desc);
```

где prefix — префикс драйвера, load и unload — имена процедур *prefix*_load() и *prefix*_unload(), halt — имя процедуры драйвера *prefix*_halt(), если таковая существует. Параметр desc является строкой символов, описывающих модуль. Код надстройки производит вызов *prefix*_load() при загрузке драйвера и вызов *prefix*_unload() при его выгрузке. Надстройки макроса определяются в файле moddefs.h.

Существуют два метода загрузки драйверов в ядро системы SVR4.2 [11]. Пользователь (обычно это системный администратор) может загрузить или выгрузить драйвер принудительно, используя системные вызовы modload и moduload. Альтернативным вариантом является автоматическая загрузка драйвера при первом обращении к нему. Например, ядро загружает модуль STREAMS (см. раздел 17.3.5) после того, как тот будет помещен в поток. Если драйвер неактивен в течение времени, превышающего изменяемый параметр DEF_UNLOAD_DELAY, то он становится кандидатом на выгрузку. Ядро вправе выгрузить такой драйвер автоматически в случае недостатка свободной памяти в системе. Значение DEF_UNLOAD_DELAY может быть изменено драйвером, для этого необходимо указать его в конфигурационном файле.

Динамическая загрузка SVR4.2 обладает многими возможностями и преимуществами, но при этом несвободно от недостатков. Она должна поддерживаться самим драйвером. По этой причине старые версии драйверов не могут использовать это средство. В [8] описывается другая версия динами-

ческой загрузки, не обладающая указанным недостатком. Она обеспечивает прозрачную загрузку драйвера при первом обращении к нему и (дополнительно) выгрузку драйвера после удаления последней ссылки на него. При загрузке системы в массивы cdevsw[] и bdevsw[] добавляется процедура open для всех старших номеров устройств, не имеющих определенных для них драйверов. При первом обращении к устройству процедура загружает драйвер, после чего вызывает процедуру открытия, принадлежащую этому драйверу. При запуске процесса происходит обновление таблицы переключателя, поэтому последующие операции открытия приведут к вызову процедуры open самого драйвера. Средство *автоматической выгрузки* работает по тому же принципу: если в конфигурационном файле драйвера указан флаг autounload, то со вхождением переключателя связывается специальная процедура close. Она производит вызов процедуры close драйвера и затем проверяет закрытие всех устройств с младшими номерами. После этого процедура выгружает драйвер.

Реализация динамической загрузки, описанная в [8], прозрачна для драйвера, так как использует его процедуры init и start вместо специализированных процедур инициализации и запуска. Средство хорошо работает с драйверами, удовлетворяющими определенным требованиям. Драйвер не должен иметь функций или переменных, на которые ведут прямые ссылки ядра или других драйверов. При обработке последнего вызова close драйвер обязан производить полную «уборку» системы от своих следов, что включает в себя освобождение памяти ядра, аннулирование состояний ожидания и т. д.

16.9. Перспективы

Одним из ограничений базовой структуры драйверов устройств UNIX является ограниченность поддержки разделения кодов между драйверами. Каждый конкретный класс устройств, например контроллеры Ethernet или жесткие диски, может формироваться множеством поставщиков. Для каждого устройства может быть разработан уникальный контроллер или чип. Однако лишь часть кодов является «привязанной» к контроллеру или устройству ASIC[1]. Больший их объем зависит только от класса устройства и специфики конкретной операционной системы или архитектуры процессора.

Каждый производитель устройств поставляет собственные драйверы к устройствам. Для всех них необходимо реализовать все точки входа, составляющие высокоуровневый интерфейс взаимодействия с устройством. Любой драйвер включает в себя не только ASIC-зависимые инструкции, но и высокоуровневый, не привязанный к ASIC код. В результате различные

[1] *ASIC* — сокращение от Application-Specific Integrated Circuit (интегральная схема, зависимая от приложения).

драйверы одного и того же класса устройств размножают количество служащих одним и тем же целям функций. Это не только приводит к лишним трудозатратам на создание драйвера, но и неэффективно увеличивает размер ядра.

UNIX предлагает несколько вариантов решения вышеописанной проблемы. Библиотека STREAMS дает возможность создавать драйверы символьных устройств в виде модулей. Любой поток складывается из нескольких модулей, каждый из которых производит какую-либо специфическую операцию с данными. Драйверы умеют разделять коды на модульном уровне, раз несколько потоков обладают правом разделять общий модуль.

Устройства SCSI обладают еще одной возможностью, позволяющей совместное использование кодов драйверов. Контроллер SCSI управляет различными типами устройств, которые могут быть не только жесткими дисками, но и флоппи-дисководами, накопителями на магнитных лентах, аудиокартами, приводами компакт-дисков и т. д. Каждый SCSI-контроллер обладает специфическим набором средств и, следовательно, требует написания для него отдельного драйвера. Конкретный тип устройства также может иметь различную семантику и тоже требовать собственного драйвера. Отсюда следует, что при m типах контроллеров и n типах устройств необходимо создать n×m драйверов.

Более предпочтительным видится разбиение кода на две части, зависимую от устройства и зависимую от контроллера. Для каждого типа устройства будет иметься один модуль, а для отдельного типа устройства — свой драйвер. Такой подход требует создания четко сформулированного интерфейса взаимодействия между двумя составляющими. Необходимо создать стандартный набор команд, понимаемых всеми контроллерами и используемых каждым модулем устройства.

Существуют несколько стандартов, целью появления которых было создание драйверов по вышеописанной схеме. В системе SVR4 реализован интерфейс PDI (Portable Device Interface, интерфейс переносимых устройств), специфицирующий следующие компоненты:

- набор функций раздела 2, которые необходимо реализовать для каждого главного адаптера шины;
- набор функций раздела 3, выполняющих общие задачи, необходимые для устройств SCSI, такие как выделение командных и управляющих блоков;
- набор структур данных раздела 4, используемых функциями предыдущего раздела.

В системе Digital UNIX [3] поддерживается интерфейс со сходным разбиением на уровни под названием *SCSI CAM* (общий метод доступа, Common Access Method) [2]. Наиболее популярным в мире персональных компьютеров стал интерфейс *ASPI* (интерфейс периферийных устройств SCSI Adaptec, Adaptec SCSI Peripheral Interface).

В Mach 3.0 [5] подсистема ввода-вывода расширила разделение уровней устройств. В этой системе коды устройств оптимизированы за счет возможности разделения каждого класса устройств. Любой класс обладает независимыми от конкретного устройства модулями, реализующими общие коды. Все автономные операции перенесены на прикладной уровень, что позволило уменьшить размеры ядра. В системе также обеспечивается прозрачность местонахождения устройства при помощи средств межпроцессного взаимодействия (IPC). Такой подход позволяет обращаться к устройствам, находящимся на других машинах. Однако интерфейс системы Mach 3.0 несовместим с инфраструктурой драйверов UNIX.

16.10. Заключение

В этой главе были описаны базовые понятия драйверов устройств в системах UNIX, а также взаимодействие подсистемы ввода-вывода с устройствами. Ближе к концу вы узнали о спецификации DDI/DKI системы SVR4, позволяющей производителям создавать драйверы устройств, легко переносимые на различные реализации SVR4. Кроме этого читатель познакомился с некоторыми современными возможностями драйверов, такими как поддержка многопроцессорной обработки, динамическая загрузка и интерфейсы разделения кодов между драйверами.

16.11. Упражнения

1. Почему для любых действий с драйверами устройств в UNIX принято использовать таблицы переключателей?
2. Чем отличается DMA от DVMA?
3. В каких случаях предпочтительнее производить опрос устройств, не дожидаясь получения прерываний от них?
4. Почему псевдоустройства типа mem или null реализованы как драйверы устройств?
5. Назовите примеры устройств, не используемых для ввода-вывода. Какие функции интерфейса они могут поддерживать?
6. Назовите примеры устройств, для которых не совсем подходит традиционный интерфейс взаимодействия с драйверами, принятый в UNIX. Какие части интерфейса являются чужеродными для таких устройств?
7. Почему процедуры верхней «половины» должны защищать структуры данных от процедур нижней «половины»?
8. Перечислите преимущества, которые дает ассоциирование специального файла с каждым устройством.

9. Каким образом в файловой системе specfs реализована обработка ситуации, когда с одним устройством ассоциируется сразу несколько файлов?
10. Зачем нужен общий объект snode?
11. Какие средства должны быть реализованы в драйвере для поддержки клонирования устройств? Назовите примеры устройств, обладающих такой возможностью.
12. Укажите отличия в интерпретации ввода-вывода символьных устройств по сравнению с блочными устройствами и файлами.
13. В чем заключается разница между функциями poll и select? Опишите, как можно реализовать первую из них в виде библиотечной функции на основе второй и наоборот. Какие при этом могут возникнуть проблемы?
14. Что имеется в виду под поддержкой устройством доступа к отображениям в памяти? Для каких типов устройств такая возможность дает некоторые преимущества?
15. Спецификация DDI/DKI запрещает прямой доступ к структурам данных и требует использования для этой цели процедурного интерфейса. Почему? Назовите преимущества и недостатки применения вызовов функций для обращения к полям структур данных.
16. Какие главные проблемы возникают при разработке драйвера, поддерживающего многопроцессорную обработку?
17. Опишите преимущества динамически загружаемых драйверов. О каких возможных проблемах необходимо знать разработчику такого драйвера?

16.12. Дополнительная литература

1. American National Standard for Information Systems, «Small Computer System Interface-2 (SCSI-2)», X3.131–199X, Feb. 1992.
2. American National Standard for Information Systems, «SCSI-2 Common Access Method: Transport and SCSI Interface Module», working draft, X3T9.2/90–186, rev. 3.0, Apr. 1992.
3. Digital Equipment Corporation, «Guide to Writing Device Drivers for the SCSI/CAM Architecture Interfaces», Mar. 1993.
4. Egan, J. I., and Texeira, T. J., «Writing a UNIX Device Driver», John Wiley & Sons, 1988.
5. Forin, A., Golub, D., and Bershad, B. N., «An I/O system for Mach 3.0», Technical Report CMU-CS-91-191, School of Computer Science, Carnegie Mellon University, Oct. 1991.

6. Gould, E., «Device Drivers in a Multiprocessor Environment», Proceedings of the Summer 1992 USENIX Technical Conference, Jun, 1992, pp. 357–360.
7. Kleiman, S. R., «Vnodes: An Architecture for Multiple File System Types in Sun UNIX», Proceedings of the Summer 1986 USENIX Technical Conference, Jun. 1986, pp. 238–247.
8. Konnerth, D., Bartel, E., and Adler, 0., «Dynamic Driver Loading for UNIX System V», Proceedings of the Spring 1990 European UNIX Users' Group Conference, Apr. 1990, pp. 133–138.
9. Pajari, G., «Writing UNIX Device Drivers», Addison-Wesley, Reading, MA, 1992.
10. Sun Microsystems, «Writing Device Drivers», Part No. 800-5117-11,1993.
11. UNIX System Laboratories, «Device Driver Programming — UNIX SVR4.2», UNIX Press, Prentice-Hall, Englewood Cliffs, NJ, 1992.

Глава 17
Подсистема STREAMS

17.1. Введение

Традиционные базовые средства драйверов устройств имеют несколько недостатков. Во-первых, ядро взаимодействует с драйвером на очень высоком уровне (через точки входа), что требует от последнего обработки почти всего объема ввода-вывода. Драйверы устройств очень часто создаются независимо от поставщиков самих устройств. На самом деле привязанной к конкретному устройству является лишь часть кода драйвера, оставшаяся часть реализует обработку высокого уровня. В результате разные драйверы дублируют одни и те же средства, что приводит к увеличению размера ядра и грозит конфликтами.

Еще одним узким местом традиционного подхода является буферизация. Интерфейсы блочных устройств обладают, бесспорно, полезной поддержкой выделения буферов и управления ими. Однако для символьных устройств не существует подобной унифицированной схемы. Их интерфейс изначально был рассчитан на работу с медленными устройствами, осуществляющими чтение или запись символ за символом, такими как телексы или низкоскоростные последовательные линии. Отсюда следует, что для устройств символьного типа реализована минимальная поддержка буферизации, ответственность за которую переложена на каждое отдельное устройство. Это привело к созданию некоторых *временных* буферов и схем управления памятью, примером которых являются clists, используемые традиционными драйверами терминалов. Специализация подобных механизмов сопровождается неэффективным использованием памяти и дублированием кодов.

Также заметим, что интерфейс взаимодействия с драйверами предоставляет приложениям ограниченные возможности. Ввод-вывод символьных устройств требует применения системных вызовов read или write, которые интерпретируют данные как поток байтов FIFO. Вызовы не поддерживают отслеживания границ сообщения, разделения данных на обычные и управляющую информацию, а также назначение сообщениям различных уровней приоритета. Не существует каких-либо средств для управления потоками. Для решения этой проблемы в каждом драйвере реализуются собственные *временные* механизмы.

Наиболее ярким примером необходимости внесения изменений в концепцию драйверов стали сетевые устройства. Как известно, сетевые протоколы поделены на несколько уровней. Передача данных основана на пакетах (или сообщениях). На каждом уровне протоколы производят с ними некоторые действия и затем переправляют их на следующий уровень. Протоколы умеют различать обычные и служебные данные. Сетевые уровни имеют некоторые взаимозаменяемые части, поэтому один протокол может комбинироваться другими протоколами, принадлежащими иным уровням. Такая сложная сетевая модель требует модульной структуры, поддерживающей разделение на уровни и позволяющей получать готовые драйверы посредством комбинирования нескольких независимых модулей.

Подсистема STREAMS решает большинство вышеописанных проблем. Она предоставляет разработчикам драйверов возможность создавать программы на основе модульной структуры. Интерфейс взаимодействия подсистемы с другими компонентами основан на сообщениях и содержит такие средства, как управление буферами, потоками, а также диспетчеризацию по приоритету. Подсистема поддерживает наборы многоуровневых протоколов посредством объединения последних в стек так, чтобы они образовывали *конвейер* (pipeline). Коды программ стали разделяемыми благодаря тому, что каждый поток состоит из набора модулей, которые могут совместно использоваться различными драйверами. STREAMS предоставляет дополнительные средства обмена данными и способы разграничения управляющей информации и обычных данных для приложений прикладного уровня.

Подсистема STREAMS была разработана Д. Ритчи (Dennis Ritchie) и на сегодняшний день поддерживается большинством производителей UNIX. Именно эта архитектура стала превалирующей основой для создания сетевых драйверов и протоколов. Кроме того, в SVR4 подсистема STREAMS заменила собой традиционные драйверы терминалов, а также механизм *каналов*. В этой главе кратко описана структура и реализация подсистемы STREAMS и произведен анализ ее преимуществ и недостатков.

17.2. Краткий обзор

Поток (stream) — это полнодуплексный канал вычислений и обмена данными между драйвером в пространстве ядра и процессом в прикладном пространстве. *STREAMS* — набор системных вызовов, ресурсов и прикладных процедур ядра, используемых для создания, применения и демонтирования потоков. Подсистема STREAMS также является базовой структурой для создания драйверов устройств. Она определяет набор правил и рекомендаций для разработчиков драйверов и предоставляет механизмы и утилиты, позволяющие создавать драйверы на основе модулей.

На рис. 17.1 показан обычный поток. Он полностью находится в адресном пространстве ядра, все его операции также реализованы на уровне ядра. Поток состоит из *головного интерфейса* (stream head), *оконечного драйвера* (driver end)[1], а также нескольких *модулей*, располагаемых между ними (модули могут отсутствовать). Головной интерфейс взаимодействует с прикладным уровнем и позволяет приложениям обращаться к потоку через системные вызовы. Оконечный драйвер взаимодействует непосредственно с устройством (это также может быть драйвер *псевдоустройства*, в таком случае происходит взаимодействие с другим потоком). Модули производят необходимые действия над данными.

Каждый модуль содержит две очереди — *очередь чтения* (read queue) и *записи* (write queue). Головной интерфейс потока и драйвер также обладают такими парами очередей. Поток передает данные путем помещения их в сообщения. Очереди записи отправляют сообщение в *нисходящем направлении* (downstream), от приложения к драйверу. Очереди чтения передают сообщения драйвера приложению, то есть в *восходящем направлении* (upstream). Несмотря на то что большинство сообщений создается либо головным интерфейсом потока либо драйвером, промежуточные модули также умеют генерировать сообщения и посылать их в обоих направлениях.

Каждая очередь может взаимодействовать со следующей очередью потока. Например, на рис. 17.1 очередь записи модуля 1 может отправлять сообщения на очередь записи модуля 2 (но не наоборот). Очередь чтения модуля 1 имеет право посылать сообщения головному интерфейсу потока. Очередь вправе также взаимодействовать с парной очередью. Таким образом, очередь чтения модуля 2 может передавать сообщения очереди записи того же модуля, которая затем отправит их в нисходящем направлении. Очереди не требуется знать, с кем она производит общение: с головным интерфейсом потока, его оконечным драйвером или другим промежуточным модулем.

Описание основ подсистемы STREAMS показывает очевидные преимущества такого подхода. Создание каждого модуля является независимым действием, осуществляемым не обязательно одними и теми же разработчиками. Модули могут выбираться и объединяться различными способами, примерно так, как происходит комбинирование команд в каналах интерпретаторов UNIX.

На рис. 17.2 показан пример построения нескольких потоков из небольшого набора компонентов. К примеру, разработчик сетевого программного обеспечения добавляет в свой продукт поддержку стека протоколов TCP/IP. Если использовать для этой цели подсистему STREAMS, он может создать модули TCP, UDP и IP. Другие поставщики сетевых интерфейсных карт вольны создавать отдельные драйверы для сетевых карт Ethernet и Token Ring.

[1] Также называемого downstream end, в отличие от upstream end (головной интерфейс). — *Прим. ред.*

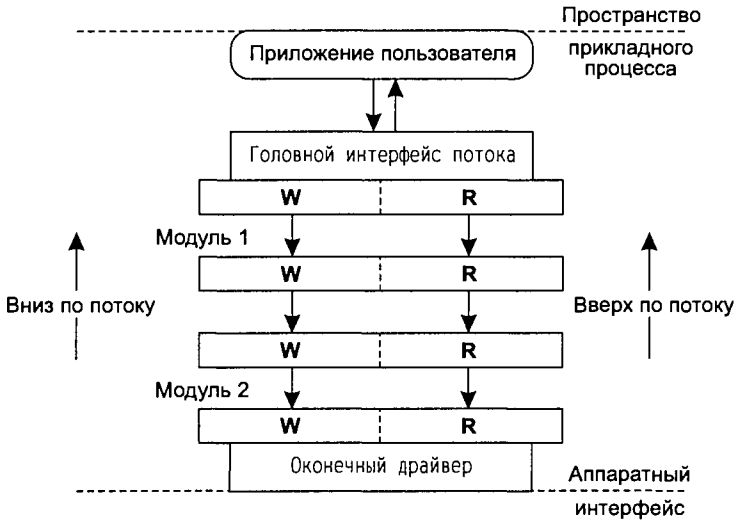


Рис. 17.1. Обычный поток: W — запись, R — чтение

После разработки отдельных модулей можно произвести их динамическую настройку, сформировав несколько типов потоков. На рис. 17.2, *а* показан поток TCP/IP, соединенный с Token Ring. На рис. 17.2, *б* представлено уже другое сочетание модулей, в котором поток UDP/IP взаимодействует с драйвером карты Ethernet.

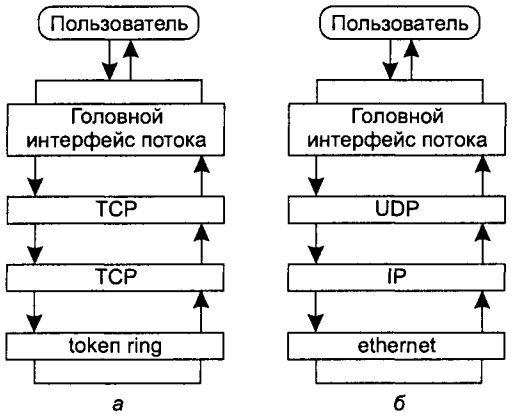


Рис. 17.2. Многократно используемые модули: *а* — вариант 1; *б* — вариант 2

Подсистема STREAMS поддерживает *мультиплексирование* (multiplexing). Мультиплексирующий драйвер может присоединяться к нескольким потокам как «сверху», так и «снизу». Существуют три типа мультиплексоров: разветвления по входу, выходу и двунаправленные. Мультиплексор разветвления *по входу* (fan-in) может присоединяться к нескольким потокам выше

него. Мультиплексор разветвления *по выходу* (fan-out) замыкается на несколько потоков ниже его. Двунаправленные мультиплексоры поддерживают несколько соединений в обоих направлениях.

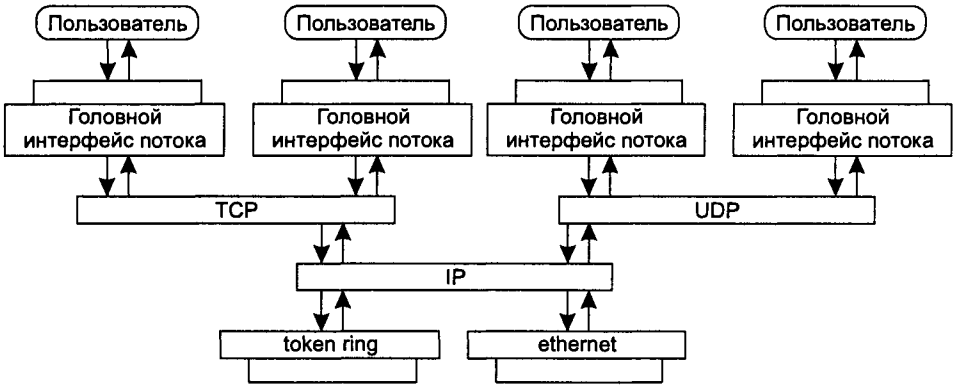


Рис. 17.3. Мультиплексирование потоков

Если модули TCP, UDP и IP создать в виде мультиплексирующих драйверов, то мы сможем с их помощью уплотнить потоки в один сложный объект, поддерживающий несколько путей перемещения данных. На рис. 17.3 показано одно из возможных решений. Драйверы TCP и UDP осуществляют мультиплексирование по входу, в то время как драйвер IP является двунаправленным мультиплексором. Это позволяет приложениям создавать различные типы соединений и предоставляет пользователям возможность обращаться к любой конкретной комбинации протоколов и драйверов. Мультиплексирующие драйверы должны уметь корректно управлять всеми соединениями и передавать данные через верхнюю или нижнюю части потока так, чтобы они попали в соответствующую очередь.

17.3. Сообщения и очереди

Подсистема STREAMS использует обмен сообщениями как единственную форму коммуникаций. Посредством сообщений производится передача всех данных между приложениями и устройством. Внутри сообщения также находится управляющая информация для драйвера или модуля. Драйверы и модули создают сообщения с целью информирования пользователя, друг друга, а также уведомляют в них об ошибочных состояниях или нестандартных событиях. Очередь вправе обрабатывать входящее сообщение несколькими различными способами. Сообщение может быть передано другой очереди без изменения, либо после некоторых действий. Очередь может произвести диспетчеризацию обработки сообщения, то есть имеет возможность отложить все операции над ним. Очередь также может передать сообщение своей паре,

то есть переслать его в обратном направлении. Очередь даже имеет право удалить поступившее сообщение.

В этом разделе мы рассмотрим структуру сообщений, очередей и модулей, а также функции, выполняемые ими.

ПРИМЕЧАНИЕ: РАСШИРЕННЫЕ БАЗОВЫЕ ТИПЫ ДАННЫХ

В системе SVR4 приняты большие по сравнению со стандартными размеры основных типов данных. Например, в SVR3 тип данных dev_t имел размер 16 битов, в то время как в SVR4 он составляет уже 32 бита. Новые типы данных получили название Расширенных базовых типов (Extended Fundamental Types, EFT). Во многие структуры SVR 4 были добавлены дополнительные поля, отсутствовавшие в SVR3. Если такие изменения касаются публичных структур или интерфейсов, это приводит к появлению проблемы обратной совместимости[1]. Драйверы и модули, написанные для ранних версий системы, не могут взаимодействовать друг с другом. Как одно из решений этой проблемы, в SVR4 предусмотрены настройки, позволяющие собрать ядро системы со старыми типами переменных. При компиляции системы без EFT поля структур изменяются таким образом, чтобы гарантировать корректную работу старых версий драйверов. Некоторые поля помещаются в иные структуры, в зависимости от настроек компилятора. В отдельных случаях определенные структуры используются только при одной опции компилятора. В последующих версиях системы поддержка устаревших типов данных может быть отменена окончательно. В этой главе мы предполагаем, что система использует новые типы данных.

17.3.1. Сообщения

Простейшее сообщение состоит из трех объектов: структуры msgb (или типа mblk_t), структуры datab (типа dblk_t) и буфера данных. Сообщение, включающее в себя несколько частей, может быть получено объединением перечисленных триплетов объектов вместе (рис. 17.4). В структуре msgb поля b_next и b_prev соединяют сообщение с очередью, а поле b_cont служит связующим звеном для разных частей одного сообщения. Поле b_datap указывает на соответствующую структуру datab.

Структуры msgb и datab содержат информацию о буфере данных. Поля db_base и db_lim структуры datb указывают на начало и конец буфера. Полезные данные могут занимать лишь часть буфера, поэтому поля b_rptr и b_wptr структуры msgb ссылаются на начало и конец области действительных данных буфера. Для выделения буфера и инициализации полей b_rptr и b_wptr на указание начала буфера (db_base) используется процедура allocb(). При записи модулем данных в буфер происходит увеличение значения b_wptr (поле сверяется с db_lim). Чтение данных модулем из буфера сопровождается изменением значения b_rptr (при этом контролируется, не происходит ли чтение после b_wptr), то есть происходит удаление данных из буфера.

[1] Backward, возможность использования новых программ для старой системы, в отличие от upward. Из-за таких проблем SunOS SVR4 в преобразованиях для именования устройств Sun отказалась от использования EFT в Solaris 7. Полная прямая и обратная совместимость для инсталляции Solaris на SunOS 4.1.1 достигается отказом от использования утилит QuickCheck или Backup Copilot. — *Прим. ред.*

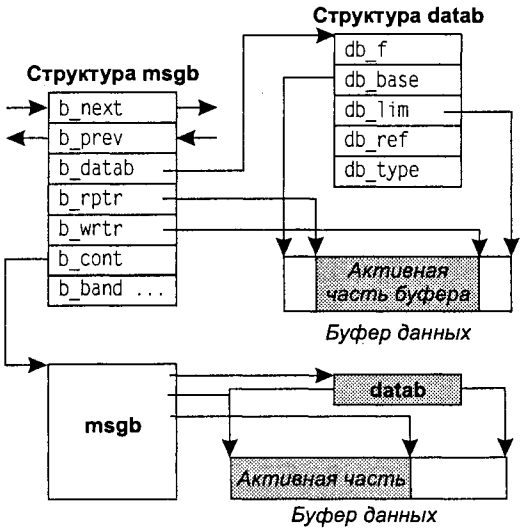


Рис. 17.4. Сообщение STREAMS

Поддержка составных сообщений дает очевидные преимущества. Сетевая модель является многоуровневой. Протоколы каждого уровня добавляют собственный заголовок или «концевик» в сообщение. При перемещении сообщения в нисходящем направлении каждый уровень может добавлять к нему новый заголовок или заключительную часть, присоединяя в начало или конец сообщения новый блок данных. При этом протоколам более высокого уровня не обязательно знать о таких добавках протоколов нижних уровней, либо резервировать свободное место для данных при создании сообщения. Если сообщение прибывает из сети и движется в потоке вверх, каждый протокол удаляет из него соответствующий заголовок или «концевик» в обратном порядке, обновляя значения полей b_rptr и b_wptr.

Поле b_band описывает уровень приоритета сообщения и используется при диспетчеризации (см. раздел 17.4.1). Каждая структура datab имеет поле db_type, которое содержит один или несколько типов сообщений, определенных подсистемой STREAMS. Это позволяет присваивать сообщениям приоритеты и обрабатывать их в той или иной последовательности в зависимости от их типа. Более подробно о типах сообщения читайте в разделе 17.3.3. Поле db_f содержит информацию, используемую при размещении сообщений (см. раздел 17.7). Поле db_ref хранит счетчик ссылок, предназначенный для виртуального копирования (раздел 17.3.2).

17.3.2. Виртуальное копирование

Структура datab включает в свой состав счетчик ссылок, помещаемый в поле db_ref. Одна структура datab может быть разделена между несколькими структурами msgb, что позволяет совместное использование данных буфера. Это

свойство структур сделало возможным *виртуальное копирование* (virtual copying) данных. На рис. 17.5 представлен пример совместного использования структуры datab двумя сообщениями. Оба сообщения разделяют ассоциированный с datab буфер, однако для каждого из них поддерживаются собственные значения смещения чтения и записи.

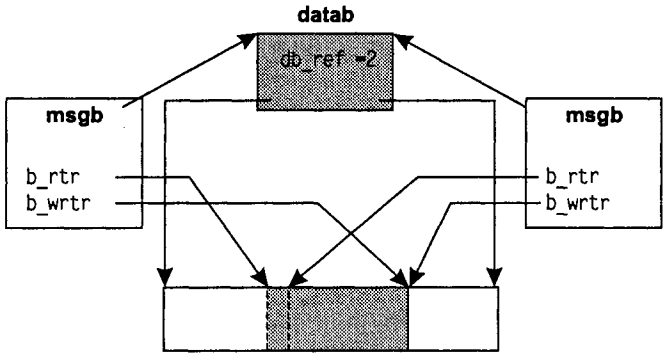


Рис. 17.5. Два сообщения, разделяющие между собой структуру datab

Как правило, разделяемые буферы используются в режиме чтения, поскольку независимая операция записи в такой буфер может влиять на результат другой записи. Однако такие правила должны реализовываться внутри драйверов или модулей самостоятельно. Подсистема STREAMS не отвечает за операции чтения или записи модулей в разделяемые буферы.

Одним из примеров виртуального копирования является протокол TCP/IP. Уровень TCP обеспечивает надежную доставку и, следовательно, должен гарантировать получение сообщения приемником. Если получатель не отправляет подтверждение о приходе сообщения в течение определенного интервала времени, отправитель повторяет передачу. Для этого необходимо хранить копии всех сообщений до поступления подтверждения. Физическое копирование каждого сообщения является избыточным, поэтому протокол TCP применяет механизм виртуального копирования. После получения сообщения, которое необходимо передать далее в нисходящем направлении, модуль TCP вызывает процедуру подсистемы STREAMS под названием dupmsg(), которая создаст еще одну структуру msgb, имеющую ссылку на ту же структуру данных datab. Это приводит к созданию двух логических сообщений, каждое из которых будет обладать ссылкой на одни и те же данные. Модуль TCP отошлет одно из сообщений в поток, придержав при себе другое.

При отправке сообщения и освобождении msgb не производится освобождение структуры datab и буфера, так как счетчик ссылок сообщения не равен нулю. После получения подтверждения модуль TCP произведет демонтаж второй структуры msgb. Это повлечет обнуление счетчика ссылок и последующее освобождение подсистемой STREAMS структуры datab и ассоциированного с нею буфера данных.

17.3.3. Типы сообщений

В STREAMS определено несколько типов сообщений. Каждое сообщение должно относиться к одному из таких типов. Для идентификации типа служит поле db_type структуры datab. Тип сообщения показывает цель его применения и влияет на приоритет диспетчеризации. Сообщения по своему типу могут быть поделены на обычные и высокоприоритетные. Сообщения последнего типа помещаются в очередь и обрабатываются вперед обычных сообщений. Более подробно о приоритетах читайте в разделе 17.4.2.

В системе SVR4.2 определены следующие типы обычных сообщений [10]:

M_BREAK	Отправляется в нисходящем направлении, используется для отправки команды break драйвером на устройство
M_CTL	Управляющий запрос модуля
M_DATA	Получение или отправка обычных данных системными вызовами
M_DELAY	Запрос задержки реального времени на выводе
M_IOCTL	Управляющее сообщение, создается для потока командами ioctl
M_PASSFP	Передает указатель файла
M_PROTO	Управляющее сообщение протокола
M_RSE	Зарезервировано
M_SETOPTS	Передается в восходящем направлении модулями или драйверами для настройки головного интерфейса потока
M_SIG	Доставляется в восходящем направлении модулями или драйверами, запрашивает головной интерфейс потока об отправке пользователю сигнала

Ниже перечислены высокоприоритетные типы сообщений.

M_COPYIN	Отправляется в восходящем направлении. Передает запрос головному интерфейсу потока на копирование данных для ioctl
M_COPYOUT	Отправляется в восходящем направлении. Также просит произвести копирование данных для ioctl
M_ERROR	Посылается в восходящем направлении. Оповещает об уровне ошибки
M_FLUSH	Запрос модуля на сброс очереди
M_HANGUP	Отправляется в восходящем направлении. Устанавливает состояние разрыва связи (hangup) для головного интерфейса потока.
M_IOCACK	Подтверждение ioctl, отправляется в восходящем направлении
M_IOCNACK	Отрицательный ответ ioctl, пересылается в восходящем направлении
M_IOCDATA	Отправляет данные ioctl в восходящем направлении
M_PCPROTO	Высокоприоритетная версия M_PROTO
M_PCSIG	Высокоприоритетная версия M_SIG
M_PCRSE	Зарезервировано

M_READ	Уведомление о чтении. Отправляется в нисходящем направлении
M_START	Продолжение приостановленного вывода устройства
M_STARTI	Продолжение приостановленного ввода устройства
M_STOP	Приостановка вывода
M_STOPI	Приостановка ввода

Поддержка различных типов сообщений дает возможность модулям отслеживать специализированные требования сообщения без просмотра его содержимого. Для составных сообщений первая структура указывает тип всего сообщения. Из этого правила имеется единственное исключение. Если приложение использует высокоуровневый служебный интерфейс, такой как *Transport Provider Interface* (интерфейс поставщиков транспорта, TPI), то сообщения данных такого приложения содержат единственный блок M_PROTO, после которого следует один или несколько блоков M_DATA внутри того же сообщения.

17.3.4. Очереди и модули

Модули являются блоками, составляющими поток. Каждый модуль обладает двумя объектами-очередями — очередью чтения и записи. На рис. 17.6 приведена структура данных queue, включающая следующие важные поля:

q_qinfo	Указатель на структуру qinit (см. ее описание далее)
q_first, q_last	Указатели, позволяющие поддерживать двунаправленный связанный список сообщений, ожидающих отложенной обработки в очереди
q_next	Указатель на следующую очередь нисходящего или восходящего потока
q_hiwait, q_lowait	Верхний и нижний пределы количества данных, удерживаемых в очереди. Используются для управления потоком данных (см. раздел 17.4.3).
q_link	Указатель, связывающий очередь со списком очередей на диспетчеризацию
q_ptr	Указатель на структуру данных, содержащую закрытые данные очереди

Поле q_qinfo указывает на структуру qinit, наследующую процедурный интерфейс взаимодействия с очередью. На рис. 17.7 показаны структуры данных, доступные через q_qinfo. Каждая очередь должна поддерживать четыре процедуры: put, service, open и close. Они являются единственными функциями других объектов STREAMS, необходимыми для организации взаимодействия с очередью. Структура module_info описывает верхнюю и нижнюю границы вместимости очереди, установленные по умолчанию, размеры пакетов и другие параметры очереди. Некоторые из этих полей повторяются в структуре queue. Это дает возможность динамически перезаписывать параметры посредством изменения их значений в структуре queue, при этом значения по

умолчанию, хранимые в module_info, остаются неизменными. Объект module_stat не используется подсистемой STREAMS напрямую. В каждом модуле могут быть реализованы собственные методы сбора статистики.

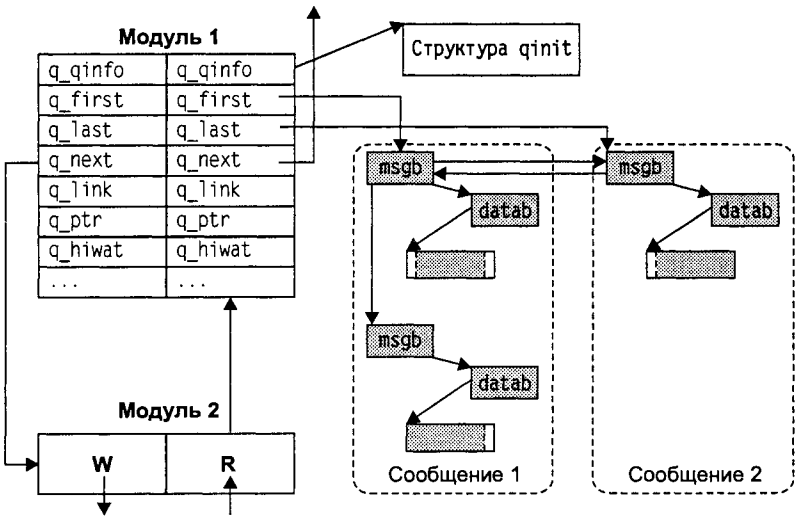


Рис. 17.6. Структура queue

В следующих разделах вас ожидает более подробное описание процедур очередей. Процедуры open и close вызываются синхронно процессом, открывающим или закрывающим поток. Процедура put производит немедленную обработку сообщения. Если сообщение невозможно обработать сразу же, процедура put помещает его в *очередь сообщений очереди*. Затем при загрузке процедуры service будет произведена обработка задержанных сообщений.

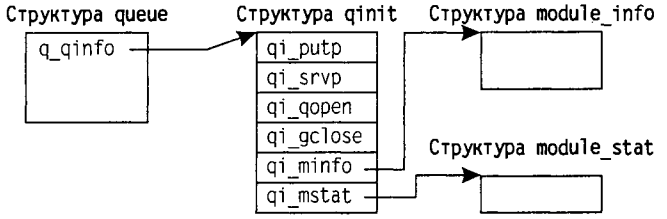


Рис. 17.7. Объекты, доступные через поле q_qinfo

Каждая очередь должна содержать реализацию процедуры put[1], однако реализация service не является обязательной. Если эта процедура не поддерживается, то put не может отложить обработку сообщений и должна производить необходимые действия над ними немедленно и пересылать сообщение следующему модулю. В простейшем случае очередь не имеет процедуры

[1] Исключая мультиплексоры (см. раздел 17.8.3).

service, а процедура put производит передачу сообщения в следующую очередь без какой-либо обработки.

ПРИМЕЧАНИЕ

Во избежание некоторой терминологической путаницы следует предупредить, что слово «очередь» в данном случае относится как к объекту очереди, так и к очереди сообщений, содержащихся в ней. В этой книге термин «очередь» применяется для обозначения объекта очереди, а термин «очередь сообщений» обозначает связанный список сообщений в очереди.

17.4. Ввод-вывод потока

Поток производит ввод-вывод посредством передачи сообщений от одной очереди к другой. Прикладной процесс записывает данные в устройство при помощи системных вызовов write или putmsg. Головной интерфейс потока создает сообщение (структуры msgb, datab и буфер) и производит копирование данных в него. Затем сообщение посылается в нисходящем направлении в следующую очередь. Через некоторое число перемещений с одного уровня на другой сообщение достигнет драйвера. Данные из устройства поступают не синхронно, драйвер перенаправляет их наверх в головной интерфейс потока. Процесс получает данные через системные вызовы read или getmsg. Если в головном интерфейсе потока не окажется доступных данных, процесс будет заблокирован.

Очередь передает сообщение в следующую очередь потока при помощи вызова putnext(). Эта функция идентифицирует следующую очередь через поле q_next и запускает для нее процедуру put. Очередь никогда не вызывает put напрямую, так как поле q_next является для очереди внутренним и может быть реализовано в следующих версиях системы другим способом. Очередь может посылать сообщения в обратном направлении посредством передачи их своей паре. Например, очередь чтения может сделать это через вызов

```
WR(q)->put (WR(q), msgp);
```

Ввод-вывод потока является асинхронным. Единственным местом блокировки операции ввода-вывода может быть только головной интерфейс потока. Процедуры put и service модуля и драйвера не являются блокирующими. Если процедура put не в состоянии отправить данные в следующую очередь, она помещает сообщение в собственную очередь, из которой оно может быть впоследствии запрошено процедурой service. Если процедура service произвела удаление сообщения из очереди, но не в силах произвести его обработку незамедлительно, она возвратит такое сообщение обратно и повторит попытку позже.

Процедуры put и service дополняют друг друга. Процедура put необходима в том случае, если обработка сообщения не вправе быть отложена. Например, драйвер терминала должен производить немедленный вывод (эхо) всех

получаемых символов, в противоположном случае пользователь будет считать, что произошел некий сбой. Процедура service выполняет менее важные действия, такие как каноническая обработка входящих символов.

Ни одна из этих процедур не имеет права вызывать какие-либо другие блокирующие процедуры. Следовательно, подсистема STREAMS предоставляет собственные средства для проведения таких операций, как выделение памяти. Например, для размещения структуры сообщения используется процедура allocb(). Если по какой-то причине невозможно завершить это действие (например, процедура не смогла найти свободные структуры msgb, datab или буфер), вместо блокировки произойдет выход с ошибкой. Затем вызывающий модуль загрузит процедуру bufcall (), передав ей указатель на функцию возврата. Процедура bufcall() добавит запрос вызывающего модуля в список очередей, для которых необходимо выделить дополнительную память. При появлении возможности предоставить память подсистема STREAMS вызовет функцию возврата, которая (как правило) запускает процедуру service потока для повторного вызова allocb().

Асинхронные операции являются центральной нитью, на которую нанизана подсистема STREAMS. На стороне очереди чтения драйвер получает данные через прерывания устройства. Процедуры put этой части потока выполняются на уровне прерываний и поэтому не могут быть заблокированы. В принципе, подсистема может позволить производить блокировку при записи данных, но разработчики отключили такую возможность из соображений ее симметричности и простоты.

Диспетчеризация процедур service происходит в системном контексте, а не в контексте процесса, инициировавшего передачу данных. Следовательно, блокировка процедуры service приведет к переводу в режим сна совершенно не относящегося к этому процесса. Например, если прикладной процесс интерпретатора будет заблокирован до завершения передачи посторонних для него данных, результат может оказаться совершенно неприемлем. Невозможность блокировки процедур put и service не дает возникнуть означенной проблеме.

Процедуры put и service должны синхронизироваться друг с другом при обращении к общим структурам данных. Процедура чтения put может быть вызвана из обработчика прерываний, поэтому существует возможность прерывания ею выполнения процедуры service, либо процедур записи put. Дополнительная блокировка требуется для многопроцессорных систем, так как они ориентированы на одновременное выполнение вызовов на разных процессорах [3], [9].

17.4.1. Диспетчер STREAMS

Если процедуре put необходимо задержать обработку данных, то она производит вызов putq() для переноса сообщения в очередь и затем вызывает qenable() для диспетчеризации очереди на обработку. Процедура qenable()

помечает очередь флагом QENAB и добавляет ее в конец списка очередей, ожидающих диспетчеризации. Если флаг QENAB уже оказался установленным, процедура qenable() не выполняет никаких действий. Последним этапом работы qenable() является установка глобального флага qrunflag, указывающего на то, что очередь ожидает диспетчеризации.

Диспетчеризация (scheduling) в подсистеме STREAMS реализована в виде процедуры runqueues() и не имеет никакого отношения к планированию процессов[1] UNIX. Ядро вызывает runqueues() при попытке процесса начать операцию ввода-вывода или произвести управление потоком. Такой подход позволяет завершать выполнение многих действий до переключения контекста. Ядро также вызывает runqueues() перед возвратом в прикладной режим после контекстного переключения.

Процедура runqueues() проверяет, нужно ли произвести диспетчеризацию какого-либо потока. Если это необходимо, то она вызывает процедуру queuerun(), которая просматривает список диспетчера и вызывает процедуры service для каждой очереди, содержащейся в нем. Процедура service может попытаться произвести действия сразу со всеми сообщениями в очереди, что будет более подробно описано в следующем разделе.

На однопроцессорных системах ядро гарантирует, что все процедуры service, вызванные диспетчером, будут выполнены до переключения в прикладной режим. Поскольку в тот же момент времени в системе может быть запущен произвольный процесс, процедуры service должны выполняться в системном контексте и не обладать правом доступа к адресному пространству текущего процесса.

17.4.2. Уровни приоритетов

Многие сетевые протоколы поддерживают понятие экстренных данных (out-of-band) [7], являющихся важной, специфичной для протокола управляющей информацией, которую необходимо обрабатывать раньше обычных данных. Такие критически срочные данные отличаются от рассматриваемых здесь высокоприоритетных сообщений, определяемых своим типом. Например, протокол Telnet предлагает механизм Synch, позволяющий перехватить управление процессом посредством отправки срочного сообщения. Как правило, конец экстренных данных помечается в потоке специальным маркером.

Подсистема STREAMS интерпретирует срочные сообщения как обычные сообщения, реагирование на которые зависит от конкретного протокола. Но STREAMS также предлагает средство под названием *уровней приоритетов* (priority bans), которое позволяет модулям присвоить сообщениям приоритеты и обрабатывать их в зависимости от уровня значимости. Некоторые протоколы могут использовать иерархию приоритетов для реализации различных классов сообщений данных.

[1] Также scheduling. — *Прим. ред.*

Уровни приоритетов назначаются только сообщениям обычных типов. Каждому сообщению присваивается весовое значение в диапазоне от 0 до 255. По умолчанию сообщение имеет приоритет 0. Большинство протоколов используют значения 0 и 1. Высокоприоритетные сообщения (например, имеющие тип M_PCROTO) могут не иметь приоритета, так как априори считаются более спешными по сравнению со всеми стандартными сообщениями.

Внутри каждой очереди подсистема STREAMS поддерживает отдельные очереди сообщений по количеству всех используемых приоритетов. Для этой цели используется набор структур qband, по одной на каждый уровень срочности. Выделение структур qband происходит динамически по мере необходимости. После вызова putq() подсистема STREAMS добавляет сообщение в хвост списка, соответствующего его приоритету (и, если это нужно, выделяет новую структуру qband). Если процедура service запрашивает сообщение через вызов getq(), то подсистема STREAMS возвратит в поджидающую очередь наиболее приоритетное из них.

Таким образом, процедура service производит обработку всех ожидающих высокоприоритетных сообщений в первую очередь, после чего начинает обрабатывать обычные сообщения, исходя из заданного для них приоритета. Внутри каждого уровня сообщения обрабатываются в порядке FIFO.

17.4.3. Управление потоком данных

Простейший вариант управления потоком (flow control) — это отсутствие какого-либо управления. Представьте поток, в котором все модули поддерживают только процедуру put. При продвижении данных внутри потока каждая очередь совершает над ними какие-то действия и передает следующей очереди через вызов putnext(). Когда данные достигают конечной точки, драйвер незамедлительно отправляет их в устройство. Если устройство не готово принять данные, драйвер удаляет такое сообщение.

Несмотря на то, что описанный метод обработки информационных потоков подходит для некоторых устройств (например, устройства null), для большинства приложений неприемлема потеря данных из-за несогласия аппаратной части. К сожалению, любое устройство системы не может быть постоянно готовым к принятию данных. Значит, удовлетворение требований приложений приводит к необходимости блокировки одного или нескольких компонентов потока и корректного разрешения ситуации без блокирования процедур put и queue.

Управление потоком данных является дополнительным средством очереди. Очередь, обеспечивающая управление потоком, взаимодействует с ближайшими модулями, также умеющими координировать данные. Если очередь не обладает таким средством, то она и не поддерживает процедуру service. В таких модулях вся обработка сообщений происходит через процедуру put незамедлительно, после чего сообщения передаются следующей очереди. При этом модуль не использует свою очередь сообщений.

В очереди, поддерживающей управление потоком, устанавливаются нижний и верхний барьеры (watermarks), ограничивающие общее количество данных, помещаемых в очередь. Значения этих параметров копируются из структуры module_info (инициализируемой статически при компиляции модуля) и могут быть в дальнейшем изменены через сообщения ioctl.

На рис. 17.8 очереди A и B обладают средствами управления потоком, в то время как очередь Б не имеет такого средства. При поступлении сообщения в очередь A запускается процедура put этой очереди. Она производит все необходимые действия с данными и вызывает putq(). Процедура putq() добавляет сообщение, полученное очередью A, в собственную очередь сообщений и помещает очередь A в список обслуживаемых очередей. Если появление сообщения в A приводит к превышению верхней границы, процедура устанавливает флаг, указывающий на переполнение очереди.

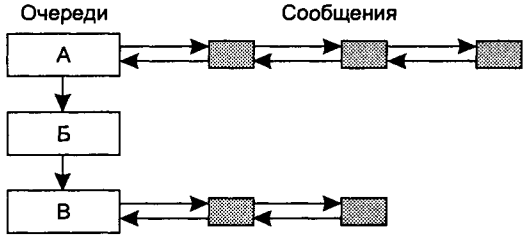


Рис. 17.8. Управление потоком данных между двумя очередями

Через некоторое время диспетчер STREAMS выбирает очередь A и вызывает для нее процедуру service. Эта процедура запрашивает сообщения из очереди в порядке FIFO. После обработки сообщений она вызывает canput() для проверки готовности следующей очереди, поддерживающей управление потоком, принять сообщение. Процедура canput() просматривает указатели q_next до тех пор, пока не найдет нужную очередь. В рассматриваемом примере это B. Затем процедура проверяет состояние очереди и возвращает TRUE, если очередь способна принять дополнительные сообщения, или FALSE, когда очередь переполнена. Процедура service очереди A в этих двух случаях будет вести себя по-разному (см. листинг 17.1).

Листинг 17.1. Обработка сообщений процедурой service

```
if (canput(q->q_next))
   putnext (q, mp);
else
   putbq (q, mp);
```

Если canput() возвращает TRUE, очередь A вызывает процедуру putnext(), которая отправляет сообщение в очередь Б. Эта очередь не поддерживает управление потоком, поэтому производит безотлагательную обработку сообщения и передает его очереди B (для которой раньше была сделана проверка на готовность принять еще одно сообщение).

Когда canput() возвращает FALSE, очередь A вызывает процедуру putbq() для возврата сообщения в его очередь сообщений. Процедура service возвратится без помещения себя на обслуживание.

Через некоторый промежуток времени очередь B обработает свои сообщения, тем самым добавив свободного места до значения большего, чем нижняя граница. Если в очереди появляется свободное место, подсистема STREAMS автоматически проверяет, не заблокирована ли предыдущая очередь, поддерживающая управление потоком (в нашем примере очередь A). Если это так, подсистема отправит ее на обслуживание. Указанная операция получила название *восстановления доступности* (back-enabling) очереди.

Управление потоком подразумевает корректность как обязательную составляющую модуля. Это означает, что все сообщения с одинаковым приоритетом должны интерпретироваться модулем эквивалентно. Если процедура put отправляет сообщения в очередь на обслуживание процедуры service, она должна сделать это одинаково для всех сообщений. В противоположном случае не сохранится определенная последовательность сообщений, что может привести к некорректным результатам.

При вызове процедуры service должны обрабатываться все сообщения в очереди, кроме случаев сбоев при размещении или при переполнении очереди, обладающей поддержкой управления потоком. Иначе «ломается» механизм управления потоком, и такая очередь может быть никогда не обработана.

Управление потоком не работает с высокоприоритетными сообщениями. Процедура put, помещающая в очередь обычные сообщения, должна производить аналогичные действия для высокоприоритетных сообщений тотчас. Если сообщение такого типа необходимо поместить в очередь, то оно окажется в начале очереди перед сообщениями обычных типов. Для высокоприоритетных сообщений также применяется порядок обработки FIFO.

17.4.4. Оконечный драйвер

Присоединенный к конечной точке потока драйвер является модулем, обладающим некоторыми отличиями от обычных модулей. Во-первых, оконечный модуль должен быть готов к возникновению прерываний. Это требует от драйвера наличия обработчика прерываний, известного ядру системы. Устройство генерирует прерывания после получения входящей информации. Драйвер должен собрать такие данные в сообщение и отправить их по потоку вверх. Если драйвер получает сообщение, идущее от головного интерфейса потока в нисходящем направлении, он обязан извлечь данные из этого сообщения и отправить их на устройство.

Как правило, разработчики драйверов добавляют в них поддержку управления потоком, так как этого требует большинство устройств. Во многих случаях (особенно для входящих данных) драйверы не обрабатывают сообщения, если они не могут управлять загрузкой. Следовательно, если драйвер не в со-

стоянии разместить буфер или его очереди переполнены, то он просто удаляет входящие и исходящие сообщения без оповещения об этом. Ответственность за удаление пакетов полностью лежит на приложениях. Высокоуровневые протоколы, такие как TCP, гарантируют надежную доставку путем сохранения копии каждого сообщения до тех пор, пока оно не достигнет приемника. Если получатель не шлет подтверждение о прибытии сообщения за определенный период времени, протокол посылает такое сообщение повторно.

Оконцовывающий драйвер также отличается от обычного модуля методами открытия и инициализации. Драйверы STREAMS открываются системным вызовом open, в то время как модули *помещаются* в потоки вызовами ioctl. Более подробно об этом читайте в разделе 17.5.

17.4.5. Головной интерфейс потока

Головной интерфейс потока отвечает за обработку системных вызовов. Он также является единственной частью потока, в которой операции ввода-вывода могут блокировать вызывающий процесс. Несмотря на то, что каждый модуль потока имеет собственные процедуры put, service, open и close, все головные интерфейсы разделяют общий набор процедур, являющихся внутренними для подсистемы STREAMS.

Процесс записывает данные в поток при помощи системных вызовов write или putmsg. Системный вызов write позволяет вести запись только обычных данных и не гарантирует соблюдение требуемых правил. Он полезен для приложений, представляющих потоки в виде последовательностей байтов. Системный вызов putmsg позволяет процессу создать управляющее сообщение и сообщение с обычными данными за один прием. STREAMS объединит их в единое сообщение, у которого первая структура datab будет иметь тип M_PROTO, а следующая — тип M_DATA.

В обоих случаях головной интерфейс потока производит копирование данных и прикладного адресного пространства в сообщения STREAMS и затем вызывает canput() для проверки наличия свободного места в потоке (отсутствия переполнения следующего модуля или драйвера). Если поток не переполнен, процедура посылает данные в нисходящем направлении посредством вызова putnext() и затем возвращает управление вызывающему процессу. Если canput() возвращает FALSE, вызывающий процесс блокируется головным интерфейсом до тех пор, пока последний не будет сделан доступным следующим модулем, управляющим потоком.

Следовательно, после завершения функций write или putmsg данные могут еще не успеть достигнуть устройства. Вызывающему процессу гарантируется копирование данных в ядро и либо достижение устройства, либо помещение их в очередь модуля или драйвера.

Процесс читает данные из потока при помощи функций read или getmsg. Системный вызов read применяется только для чтения обычных данных.

Модуль может отправлять сообщение M_SETOPTS головному интерфейсу потока, указывая ему на интерпретацию сообщений M_PROTO как обычных данных. После этого вызов read читает содержимое как сообщений M_DATA, так и M_PROTO. В любом случае, системный вызов read не интересуется границами сообщений, а также не возвращает информацию об их типах. Как правило, этот вызов используется приложениями, интерпретирующими потоки как последовательности байтов.

Как бы то ни было, если данные доступны головному интерфейсу потока, ядро производит их извлечение из сообщения, копирование в прикладное адресное пространство и возвращает управление вызывающему процессу. Если в головном интерфейсе отсутствуют ожидающие сообщения, ядро блокирует процесс до прибытия сообщения. Один и тот же поток могут пытаться читать одновременно несколько процессов. Если они не находят ни одного ожидающего сообщения, все процессы будут заблокированы. При прибытии сообщения ядро передаст его одному из этих процессов. Интерфейс не определяет, какой из процессов получит сообщение.

Когда сообщение достигает головного интерфейса потока, ядро проверяет наличие ожидающего процесса. Если такой процесс существует, ядро копирует сообщение в его адресное пространство и затем пробуждает этот процесс, который, в свою очередь, возвращается из вызова read или getmsg. Если ни один процесс не ожидает принятия сообщения, ядро помещает его в очередь головного интерфейса потока. При переполнении очереди сообщений все последующие сообщения будут заноситься в очередь предыдущего модуля, обладающего средством управления потоком, и так далее по нисходящей.

17.5. Конфигурирование и настройка

В этом разделе описывается последовательность конфигурирования драйвера или модуля STREAMS в системе, а также создание и настройка потока в работающем ядре. Конфигурирование в STREAMS состоит из двух фаз. Сначала (на стадии построения ядра) модули и драйверы STREAMS должны быть соединены с ядром системы, а соответствующие процедуры ядра должны обнаружить их. Второй фазой является создание и настройка файлов устройств для того, чтобы разрешить приложениям обращаться к потокам как к обычным файлам. Настройка STREAMS является динамической и происходит при открытии потока и помещении в него модулей.

17.5.1. Конфигурирование модуля или драйвера

Модули и драйверы STREAMS обычно создаются поставщиками систем независимыми от ядра. Модули и драйверы должны быть соединены с другими частями ядра, при этом ядро получает информацию о методах доступа к ним. Подсистема STREAMS обладает полным набором средств для этой цели.

Каждый модуль STREAMS должен поддерживать три структуры данных настроек — module_info, qinit и streamtab. На рис. 17.9 показаны содержимое и взаимосвязи этих структур. Структура streamtab является единственным объектом, видимым снаружи модуля. Остальные структуры описаны как статические и, следовательно, не видны извне модуля.

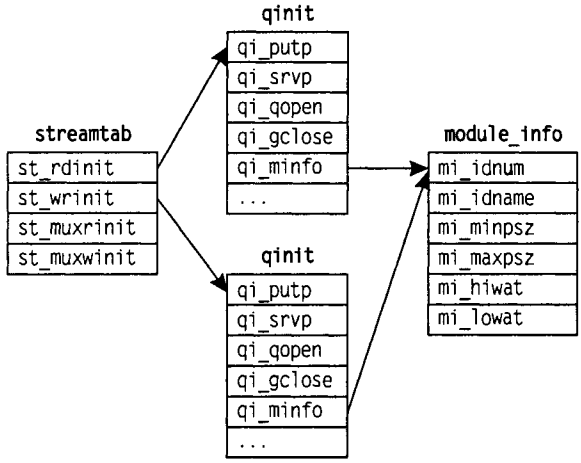


Рис. 17.9. Структуры данных, участвующие в конфигурировании модуля или драйвера

Структура streamtab содержит указатели на две структуры qinit, одну для очереди чтения и одну для очереди записи. Остальные поля используются только мультиплексирующими драйверами для хранения указателей на дополнительные пары очередей. Структура qinit содержит указатели на набор функций (open, close, put и service), формирующих процедурный интерфейс доступа к очереди. Процедуры open и close являются общими для модуля и определяются только для очереди чтения. Все очереди обладают процедурой put, однако процедуру service поддерживают только те очереди, которые имеют свойство управления потоком. Структура qinit также включает в себя указатель на структуру module_info очереди.

Структура module_info хранит параметры модуля, задаваемые по умолчанию. При первом открытии модуля эти параметры копируются в структуру очереди. Пользователь может изменить их значения посредством вызова функций ioctl. Каждая очередь имеет собственную структуру module_info, либо единственная структура будет разделяться двумя очередями.

Остальные настройки модулей и драйверов неодинаковы. Во многих системах UNIX для конфигурирования модулей STREAMS используется таблица fmodsw[]. Каждое вхождение таблицы (см. рис. 17.10, *а*) состоит из имени модуля и указателя на структуру streamtab модуля. Модули идентифицируются и различаются между собой по именам. Имя модуля должно совпадать со значением в поле mi_idname структуры module_info, несмотря на то, что подсистема STREAMS не обязывает к этому.

Драйверы устройств STREAMS идентифицируются через таблицу переключателя символьных устройств. Каждое вхождение cdevsw имеет поле d_str, которое содержит NULL в случае обычных символьных устройств. Если вхождение относится к устройству STREAMS, то поле d_str будет содержать адрес структуры streamtab драйвера (см. рис. 17.10, *б*). Для завершения конфигурирования драйвера необходимо создать файлы устройств, имеющие старшие номера, равные индексу драйвера в массиве cdevsw[] (кроме клонов, о которых будет рассказано в разделе 17.5.4). Драйверы STREAMS должны обрабатывать прерывания устройств и поэтому требуют некоторого дополнительного механизма установки обработчиков этих прерываний в ядро системы. Эта процедура является системно-зависимой.

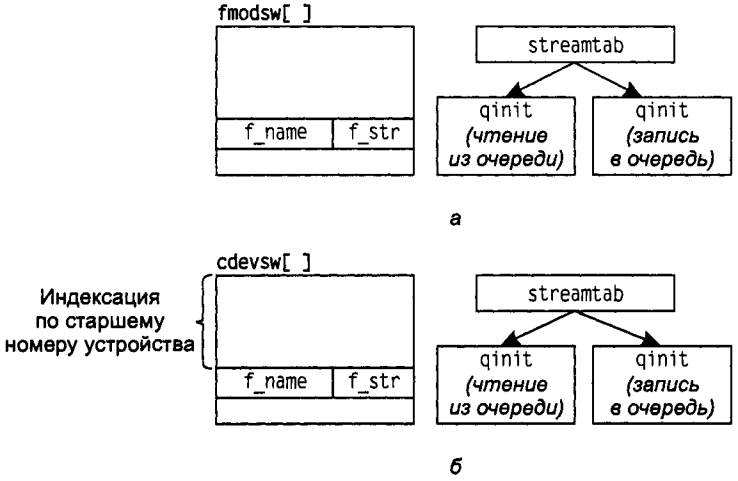


Рис. 17.10. Конфигурация модулей и драйверов:
а — модули STREAMS; *б* — драйверы STREAMS

После конфигурирования драйвер или модуль становится доступным для использования приложениями после загрузки ядра. В следующих подразделах вы увидите, как это происходит.

17.5.2. Открытие потока

Пользователь открывает поток открытием соответствующего ему файла устройства. При первом открытии конкретного устройства STREAMS ядро преобразует полное имя файла и определяет, что данный файл относится к символьному устройству. Ядро вызывает процедуру specvp(), которая производит выделение и инициализацию объекта snode и общего объекта snode для этого файла (см. раздел 16.4.4). Затем загружается операция VOP_OPEN объекта vnode (объект vnode, ассоциированный с общим объектом snode, является vnode потока), которая вызывает функцию spec_open(). Эта функция

просматривает массив cdevsw[], используя в качестве ключа старший номер устройства, и находит, что данное устройство относится к STREAMS (так как поле d_str!=NULL). Затем вызывается процедура stropen(), которой передаются указатели на vnode и на номер устройства, а также флаги открытия и привилегий. Если stropen() вызывается для нового потока, то функция производит указанную ниже последовательность действий.

1. Выделяет пару очередей для головного интерфейса потока.
2. Выделяет и инициализирует структуру stdata, представляющую собой этот интерфейс.
3. Устанавливает для очередей головного интерфейса потока указатели на объекты strdata и stwdata, являющиеся структурами qinit (для очереди чтения и записи соответственно), содержащими общие функции головного интерфейса.
4. Сохраняет указатель на vnode в поле sd_vnode структуры stdata.
5. Помещает указатель на головной интерфейс потока в объект vnode (в поле v_stream).
6. Сохраняет указатель на структуру streamtab драйвера (получаемый из вхождения cdevsw) и поле sd_streamtab головного интерфейса потока.
7. Устанавливает закрытый указатель структуры stdata для очередей головного интерфейса потока (d_str).
8. Вызывает qattach() для настройки оконечного драйвера потока.
9. Помещает все *автоматически помещаемые* (autopush) модули устройства в стек потока при помощи вызова qattach(). Об этом читайте в следующем разделе.

На рис. 17.11 показаны все структуры потока после возврата функции stropen().

Функция qattach() производит присоединение модуля или драйвера к головному интерфейсу, при этом функция выполняет перечисленные ниже операции.

1. Выделяет пару очередей и присоединяет их к головному интерфейсу потока снизу.
2. Находит структуру streamtab через массив cdevsw[] драйвера или массив fmodsw[] модуля.
3. Из структуры streamtab получает структуры qinit чтения и записи и использует для их инициализации поле q_qinfo пары очередей.
4. Вызывает процедуру open модуля или драйвера.

Представьте ситуацию, что рассматриваемый нами поток открыл еще один пользователь. Это могло быть сделано двумя способами, либо через тот же файл устройства, либо через другой файл, имеющий такие же старшие и младшие номера (обрабатывается общим объектом snode). Ядро проверяет

поле v_stream vnode общего объекта snode и находит, что оно не равно NULL, а содержит указатель на структуру stdata потока. Это показывает, что поток уже открыт. В данном случае процедура stropen() всего лишь вызывает процедуры open драйвера потока и каждого модуля в этом потоке с целью информирования об открытии другим процессом того же потока.

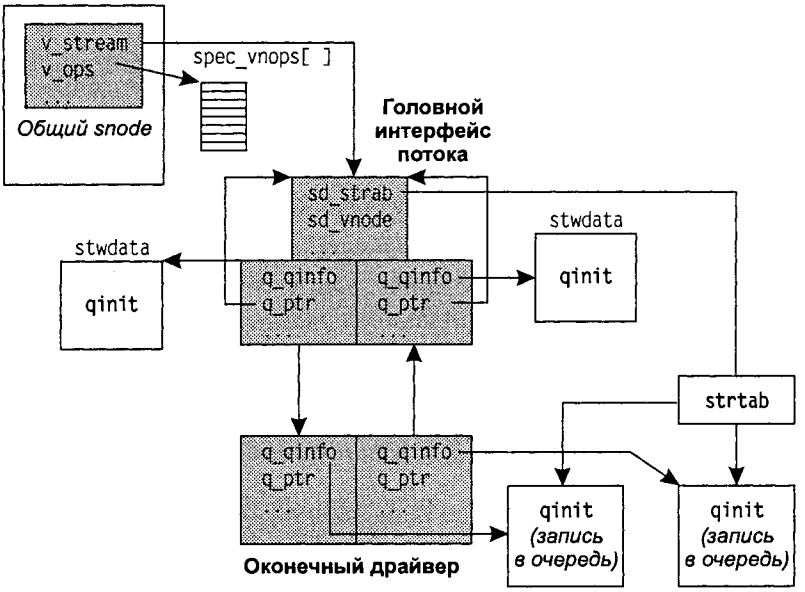


Рис. 17.11. Структуры данных после завершения работы stropen()

17.5.3. Помещение модулей в поток

Пользователь может поместить модуль в открытый посредством вызова ioctl с командой I_PUSH. Ядро выделяет пару очередей и вызывает qattach() для добавления их в поток. Процедура q_attach() инициализирует модуль, обнаруживая его вхождения strtab в таблице fmodsw[]. Она соединяет модуль с потоком, располагая его сразу после головного интерфейса, и вызывает для него процедуру open.

Пользователь может изъять модуль из очереди командой I_POP вызова ioctl. При этом всегда удаляется модуль, расположенный непосредственно за головным интерфейсом потока. Таким образом, модули удаляются из потока в порядке FIFO («первым вошел, первым вышел»).

Подсистема STREAMS поддерживает механизм автоматического помещения модулей в стек потока. Для этой цели используются команды вызова ioctl специального *административного драйвера STREAMS* (administrative driver, sad [8]). Администратор может указать набор модулей, помещаемых в определенный поток подсистемой при первом его открытии. Процедура

stropen() проверяет, активна ли опция автоматического размещения модулей в стеке потока. Если это так, процедура находит и подключает необходимые модули в требуемом порядке.

Существуют два альтернативных метода помещения модуля в поток. Один из них предполагает применение библиотечных процедур, которые открывают поток и присоединяют к нему корректные модули. Еще одним вариантом является загрузка процесса-демона при инициализации системы, производящего аналогичные действия. В любом случае при открытии приложением файла устройства произойдет подключение к потоку, содержащему все необходимые модули.

17.5.4. Клонирование устройств

Понятие клонирования устройств впервые появилось на страницах книги в разделе 16.4.5. В основе клонирования лежит тот факт, что устройства определенных типов могут иметь несколько эквивалентных копий. Каждой копии устройства необходим уникальный младший номер. Если пользователь открывает такое устройство, ему не нужно заботиться о том, с какой именно копией он начнет работать. Пользователь не ищет свободный младший номер для копии устройства самостоятельно, а (как правило) за это ответственен драйвер устройства.

Клонирование используется для большинства устройств STREAMS, таких как сетевые протоколы или *псевдотерминалы*. Следовательно, подсистема STREAMS поддерживает *драйвер клонирования*, который позволяет автоматизировать задачу клонирования устройств. *Устройство клонирования* имеет собственный старший номер и реализуется как драйвер STREAMS. Этот драйвер предоставляет каждому устройству STREAMS, поддерживающему клонирование, по одному файлу устройства. Его старший номер совпадает с номером *устройства клонирования*, а младший номер равен старшему номеру реального устройства.

Представим, к примеру, что драйвер клонирования имеет старший номер 63. Файл устройства /dev/tcp может представлять все потоки протокола TCP. Если драйвер TCP имеет старший номер устройства, равный 31, то файл /dev/tcp будет обладать старшим номером 63 и младшим номером 31. Когда пользователь открывает /dev/tcp, ядро выделяет объект snode и общий объект snode для этого файла и затем вызывает процедуру spec_open(). Процедура загружает операцию d_open драйвера клонирования, передавая указатель на номер устройства (то есть указатель на поле s_dev общего объекта snode).

Операция d_open драйвера клонирования реализована как процедура clnopen(). Она производит выделение младшего номера из поля s_dev (в рассматриваемом примере — 31) и использует его как индекс к таблице cdevsw[] с целью обнаружения драйвера TCP. Затем она загружает операцию d_open для этого драйвера, передавая ей в качестве параметров флаг CLONEOPEN и но-

мер устройства. В нашем примере это приведет к вызову функции tcpopen(). Функция tcpopen() распознает флаг CLONEOPEN, генерирует неиспользованный ранее младший номер устройства и записывает его в объект snode.

После возврата clnopen() процедура spec_open() различает случай открытия клона. Общий объект snode был ассоциирован с устройством клонирования (/dev/tcp), поэтому процедура spec_open() должна выделить для этого соединения новые объекты vnode и snode. Она инициализирует поле v_stream созданного объекта vnode так, чтобы оно ссылалось на головной интерфейс потока, и копирует в оба объекта новые младший и старший номера устройства (из поля s_dev). Затем вызывается stropen() для открытия нового потока.

Последним этапом работы spec_open() является обнуление поля v_stream оригинального общего объекта snode (ассоциированного с /dev/tcp). Система воспринимает это так, что устройство не открывалось ранее. Позже, если другой процесс пытается открыть файл /dev/tcp, ядро производит ту же серию операций и строит новый поток, а также выделяет номер устройства для него. В результате пользователь получает уникальное соединение TCP и ему не нужно знать о том, какой при этом используется младший номер.

17.6. Вызовы ioctl подсистемы STREAMS

Подсистеме STREAMS необходимы специализированные механизмы работы с ioctl. Через некоторые команды ioctl производятся все операции с головным интерфейсом потока, остальная часть команд относится к драйверу или промежуточным модулям. Все команды преобразуются в сообщения и посылаются по потоку в нисходящем направлении. Это приводит к возникновению двух различных проблем для механизмов синхронизации процессов и передачи данных между прикладным пространством и пространством ядра.

За синхронизацию процессов отвечает головной интерфейс потока. Если он может обработать команду, то осуществляет это действие синхронно в контексте процесса. В этом случае никаких проблем не возникает. Если головному интерфейсу потока необходимо передать команду в нисходящем направлении, то он блокирует процесс и отправляет сообщение M_IOCTL, содержащее команду с ее входными параметрами. Модуль обрабатывает эту команду и затем возвращает результат выполнения в сообщении M_IOCACK. Если сообщение не может быть обработано ни одним драйвером или модулем, драйвер генерирует сообщение M_IOCNACK. Если головной интерфейс потока получает любое из этих сообщений, то в ответ он пробуждает процесс и передает ему результаты выполнения.

Проблема передачи данных связана с необходимостью обмена параметрами и результатами выполнения команд между программой и модулем (или драйвером), обрабатывающими вызовы ioctl. Если команда ioctl вызывается по отношению к обычному символьному устройству, драйвер обрабатывает

ее в контексте вызывающего процесса. Каждая команда ioctl имеет связанный с ней блок параметров, размеры и содержание которых неодинаковы. Драйвер копирует этот блок из прикладного пространства в ядро, выполняет команду и дублирует результаты в прикладном пространстве.

Описанный метод представляет собой камень преткновения для драйверов и модулей STREAMS. Модуль получает команды в виде сообщений M_IOCTL, поступающие асинхронно. При этом модуль находится в системном контексте. Он не имеет права доступа к прикладному адресному пространству и поэтому не может скопировать блок параметров команды или передать блок результатов в пространство процесса.

Подсистема STREAMS предлагает два способа решения проблемы. Одно из решений заключается во введении нового типа команд ioctl под названием I_STR. Второй метод применяется для обработки обычных вызовов ioctl и необходим для поддержки совместимости со старыми версиям программ. Он получил название *прозрачных вызовов ioctl* (transparent ioctl) и не требует внесения изменений в существующие приложения.

17.6.1. Команда I_STR вызова ioctl

Стандартный синтаксис системного вызова описывается следующим образом:

```
ioctl (fd, cmd, arg);
```

где fd является файловым дескриптором, cmd — целое число, указывающее вызываемую команду, и arg — дополнительный аргумент, специфичный для указанной команды (чаще всего адрес блока параметров). Драйвер интерпретирует содержимое arg исходя из выбранной команды cmd и производит копирование соответствующих параметров из прикладного адресного пространства.

Процесс может создать специальное сообщение STREAMS ioctl, указав в качестве значения cmd константу I_STR. При этом параметр arg должен указывать на структуру strioctl, имеющую следующий формат:

```
struct strioctl {
   int ic_cmd; /* выполняемая команда */
   int ic_timeout; /* период тайм-аута */
   int ic_len; /* длина блока параметров */
   char *ic_dp; /* адрес блока параметров */
};
```

Если головной интерфейс потока не может обработать вызов ioctl, то он создает сообщение типа M_IOCTL и копирует в него значение ic_cmd. Он также выделяет блок параметров (адресуемый полями ic_len и ic_dp) из прикладного пространства и копирует его в создаваемое сообщение. Затем это сообщение передается в нисходящем направлении. Через некоторое время оно попадает в модуль, обрабатывающий указанную в сообщении команду. При этом сообщение содержит всю информацию, необходимую для обработки коман-

ды. Если команда требует возврата данных, модуль записывает их в то же сообщение, изменяет его тип на M_IOCACK и отправляет обратно в восходящем направлении. Головной интерфейс потока затем производит копирование полученных результатов в блок параметров, расположенный в прикладном адресном пространстве.

Головной интерфейс потока передает сообщение в нисходящем направлении до тех пор, пока оно не достигнет того модуля, который сможет его распознать и обработать. Такой модуль посылает обратно головному интерфейсу сообщение M_IOCACK, показывающее, что команда распознана. Если сообщение не распознается ни одним модулем, то в какой-то момент времени оно достигает драйвера. Если драйвер также не может идентифицировать его, то отправляет в обратную сторону сообщение M_IOCNACK, получив которое, головной интерфейс потока генерирует соответствующий код ошибки.

Описанное решение является достаточно эффективным, но требует для обрабатываемых команд соблюдения некоторых ограничений. Оно не подходит для предыдущих версий приложений, не использующих I_STR. Головной интерфейс потока не умеет интерпретировать параметры команд, поэтому их приходится полностью размещать внутри блока. Поясним на примере. Если один из параметров является указателем на строку, находящуюся в другом месте прикладного пространства, головной интерфейс произведет копирование указателя, а не строки. Становится очевидным, что требуется более общий подход, который бы годился для всех возможных случаев, даже если он окажется чуть медленнее или менее эффективным.

17.6.2. Прозрачные команды ioctl

Прозрачные команды вызова ioctl предлагают механизм решения проблемы копирования данных для тех вызовов, которые не используют I_STR. Если процесс вызывает прозрачную команду ioctl, головной интерфейс потока создает сообщение M_IOCTL и копирует в него параметры cmd и arg. Как правило, в arg содержится указатель на блок параметров, размер и наполнение которого известны только модулю, обрабатывающему команду. Головной интерфейс потока посылает сообщение в нисходящем направлении и затем блокирует вызывающий процесс.

Если модуль получает сообщение, то отправляет в ответ другое сообщение типа M_COPYIN, передавая ему размер и местонахождение блока параметров. Головной интерфейс потока будит процесс, вызвавший команду ioctl для обработки сообщения M_COPYIN. Процесс создает новое сообщение типа M_IOCARGS, копирует в него данные из прикладного пространства и направляет его вниз потока, после чего блокируется снова.

После того как модуль получает сообщение M_IOCARGS, он интерпретирует содержащиеся в нем параметры и производит обработку сообщения. В некоторых случаях модуль создает одно или несколько ответных сообщений

M_COPYOUT, передавая через них обратно результаты и указывая местонахождение, по которому их необходимо записать. Например, если один из параметров является указателем на строку, модуль отправляет дополнительное сообщение, содержащее саму строку.

В какой-то момент времени модуль получает все необходимые параметры и тогда производит обслуживание сообщения. Если необходимо возвратить результаты процессу, модуль создает одно или несколько сообщений M_COPYOUT, передает им результаты и указывает местонахождения, по которым они должны быть записаны. При этом головной интерфейс потока каждый раз пробуждает процесс, который записывает сообщение в свое адресное пространство. После окончания копирования всех результатов модуль отправляет сообщение M_IOCACK. Поток, в свою очередь, пробуждает процесс в последний раз и завершает выполнение вызова ioctl.

17.7. Выделение памяти

Управление памятью в подсистеме STREAMS имеет некоторые специальные требования и вследствие этого не может возлагаться на обычный распределитель памяти ядра. Модули и драйверы постоянно обмениваются сообщениями и требуют эффективного механизма их выделения и освобождения. Процедуры put и service не должны блокировать процессы. Если распределитель памяти не сможет предоставить необходимую память сразу же, то он должен разрешить возникшую проблему без блокировки. Одним из возможных решений является повторная попытка выделения чуть позже. Кроме этого, многие драйверы STREAMS поддерживают прямой доступ к памяти (Direct Memory Access, DMA) к буферам устройств. Подсистема STREAMS позволяет преобразовывать такую память в сообщения напрямую, минуя стадию промежуточного копирования в основную память.

Для управления основной памятью служат процедуры allocb(), freeb() и freemsg(). Синтаксис вызова allocb() приведен ниже:

```
mp = allocb (size, pri);
```

Вызов производит выделение структур msgb, datab и буфера данных, имеющего размер не менее least байтов. Процедура возвращает указатель на msgb. Она инициализирует структуру msgb и устанавливает в ней указатель на структуру datab, которая, в свою очередь, содержит информацию о начале и конце буфера. Процедура также устанавливает поля b_rptr и b_wptr структуры msgb на указание начала буфера. Параметр pri не используется и сохранен лишь из соображений обратной совместимости. Процедура freeb() освобождает только одну структуру msgb, в то время как процедура msgfree() просматривает цепочку b_cont и освобождает все структуры msgb сообщения. В обоих случаях ядро производит уменьшение счетчика ссылок соответствующих

структур datab. Если значение счетчика становится равным нулю, ядро также освобождает datab и буфер, на который эта структура указывает.

Выделение трех объектов в индивидуальном порядке является неэффективным и медленным. Подсистема STREAMS предлагает альтернативный выход на основе структур данных под названием mdbblock. Каждая структура имеет размер 128 байтов и содержит msgb, datab и указатель на *обработчик откребления* (release handler), о котором будет рассказано в следующем разделе. Оставшиеся байты структуры могут быть использованы для буфера данных.

Посмотрим, что происходит при вызове модулем процедуры allocb() с целью выделения сообщения. Процедура allocb() вызывает kmem_alloc() для размещения структуры mdbblock, передавая ей флаг NO_SLP. Этот флаг показывает на необходимость возврата kmem_alloc() с ошибкой вместо блокировки, если процедура не в силах выделить память незамедлительно. Если размещение прошло успешно, процедура allocb() проверяет, поместится в структуре mdbblock информация указываемого размера. При положительном ответе процедура производит инициализацию структуры и возвращает указатель на msgb (см. рис. 17.12). В рассматриваемом случае единственный вызов kmem_alloc() производит выделение msgb, datab и буфера.

Если запрашиваемый размер окажется больше, allocb() вызывает kmem_alloc() повторно для выделения буфера. В этом варианте оставшиеся байты mdbblock не используются. Если вызов kmem_alloc() возвращает ошибку, процедура allocb() освобождает все ранее полученные ресурсы и возвращает NULL, что свидетельствует о сбое.

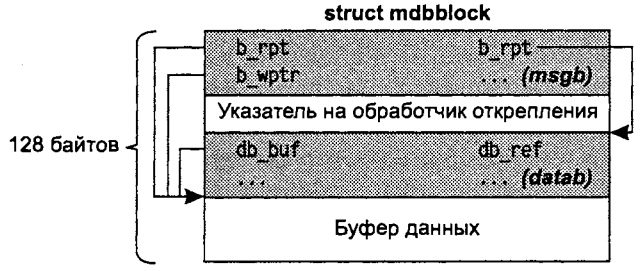


Рис. 17.12. Размещение сообщения небольшого размера

Сбой выполнения allocb() должен обрабатываться модулем или драйвером. Одним из возможных вариантов является сброс данных. Такой подход используется многими сетевыми драйверами при невозможности обработки всего входящего трафика. Однако модулю часто приходится ожидать появления свободной памяти. Процедуры модуля put и service не должны быть блокирующими, поэтому необходимо другое решение проблемы ожидания свободной памяти.

Для этой цели STREAMS предлагает процедуру под названием bufcall(). Если модуль не может выделить сообщение, то он вызывает bufcall(), передает

ей указатель на функцию возврата и размер создаваемого сообщения. Подсистема STREAMS добавляет функцию возврата во внутреннюю очередь. После того, как освободится достаточное количество памяти, подсистема обработает очередь и вызовет каждую содержащуюся в ней функцию возврата.

Часто функция возврата имеет собственную процедуру service. Однако она не обладает информацией о наличии свободной памяти. При выполнении функции возврата другие процессы могут снизить количество доступной памяти. В этом случае модуль обычно производит повторную загрузку bufcall().

17.7.1. Расширенные буферы STREAMS

Некоторые драйверы STREAMS поддерживают карты ввода-вывода, содержащие *память с двунаправленным доступом* (dual-access RAM), реже называемую *памятью с двумя портами* (dual-ported RAM). Такие карты имеют буферы памяти, к которым имеет право доступа как аппаратура устройства, так и центральный процессор. Расширенный буфер может быть отражен в адресном пространстве ядра или процесса, что позволяет приложению обращаться к нему и изменять его содержимое без промежуточного копирования данных в оперативную память.

Драйверы STREAMS помещают данные в сообщения и передают их по потоку вверх. Для защиты от копирования содержимого буфера карт ввода-вывода подсистема предлагает методику использования их в качестве буфера данных сообщения. Вместо allocb() драйвер вызывает другую процедуру под названием esballoc() и передает ей адрес буфера. Подсистема STREAMS размещает структуры msgb и datab (из mdbblock), не выделяя при этом буфер данных. Его замещает буфер карты ввода-вывода, ссылки на который записываются в соответствующие поля msgb и datab.

Использование расширенных буферов порождает проблему с их освобождением. Обычно модуль применяет для этой цели процедуры freeb() или freemsg(). Ядро системы демонтирует структуры msgb, datab и буфер данных. Процедура kmem_free() освобождает такие объекты и восстанавливает память. Однако буферы драйверов не могут быть возвращены в общий пул памяти, так как относятся к карте ввода-вывода.

Процедура esballoc() в сравнении со стандартными процедурами выделения памяти поддерживает дополнительный параметр, который задает адрес функции *обработчика освобождения* (release handler). При удалении сообщения ядро освобождает структуры msgb и datab, после чего вызывает обработчик для открепления буфера данных. Обработчик производит необходимые действия по маркировке буфера как свободного. После этого карта ввода-вывода может использовать такой буфер повторно. Синтаксис процедуры esballoc() следующий:

```
mp = esballoc (base, size, pri, free_rtnp);
```

где параметры base и size описывают буфер, а free_rptn является адресом кода функции обработчика. Параметр pri поддерживается только из соображений совместимости и не используется в SVR4. Процедура esballoc() возвращает указатель на структуру msgb.

17.8. Мультиплексирование

Подсистема STREAMS обладает средством, позволяющим выполнять мультиплексирование (multiplexing). Оно позволяет присоединять несколько потоков к одному потоку, называемому мультиплексором (multiplexor). Это средство поддерживается только для драйверов, но не для модулей. Существуют три типа мультиплексоров: разветвления по входу, выходу и двунаправленные. Верхний (upper) мультиплексор (или разветвление *по входу* (fan-in)) может присоединяться к нескольким потокам, расположенным выше него. Нижний (lower) мультиплексирующий драйвер (или разветвление *по выходу* (fan-out)) подключается к нескольким нижним по отношению к нему потокам. Двунаправленные мультиплексоры обеспечивают набор соединений в обоих направлениях. Мультиплексоры могут комбинироваться самыми различными способами, формируя тем самым сложные конфигурации (одна из которых продемонстрирована на рис. 17.13).

Подсистема STREAMS поддерживает базовую среду и набор процедур мультиплексирования. При этом задача управления несколькими потоками и корректной маршрутизации данных возлагается на сами драйверы.

17.8.1. Мультиплексирование по входу

Простейший вариант мультиплексирования по входу возникает в результате открытия нескольких младших номеров устройства в одном и том же драйвере. Подсистема STREAMS не предлагает какой-либо дополнительной поддержки такой схемы, кроме обработки системных вызовов open и close. Любой драйвер, открытый с несколькими младшими номерами устройств, является мультиплексором по входу.

При первом открытии устройства STREAMS ядро системы создает для него поток и устанавливает на этот поток ссылки через объект snode и общий объект snode. Если через некоторое время другой (или текущий) процесс откроет тот же файл устройства, ядро обнаружит, что файл уже имеет поток (то есть поток уже был создан), поэтому новый процесс будет использовать новый поток. Новый поток будет создан и в случае, если другой процесс откроет иной файл устройства, имеющий такие же младшие и старшие номера устройства. Ядро распознает тот факт, что данное устройство было открыто ранее (так как обратится к единственному общему объекту snode), и предоставит возможность двум процессам разделять между собой один и тот же поток. В данном случае мультиплексирование не используется.

Совсем другая ситуация возникает, если второй процесс открывает другой младший номер устройства. В этом случае ядро создает для него отдельный поток, а также объект snode и общий snode. Так оба потока имеют один и тот же старший номер устройства, они обслуживаются одним драйвером. Процедура open драйвера будет вызвана дважды, по одному разу на каждый поток. Драйвер будет управлять двумя наборами очередей, возможно даже с неодинаковыми комбинациями помещенных в них модулей. После получения данных из устройства драйвер производит их проверку и решает, какому из потоков передать их далее. Как правило, это решение обусловлено информацией, содержащейся в данных, либо портом контроллера, на который они поступили.

На рис. 17.13 показан пример драйвера карты Ethernet, выступающего в роли мультиплексора по входу. Оба потока могут содержать разные наборы модулей. Например, один из потоков использует модуль IP, в то время как другой — модуль ICMP. При получении данных из сети драйвер проверяет их содержимое и в зависимости от характера информации принимает решение о том, в какой из потоков их следует передать.

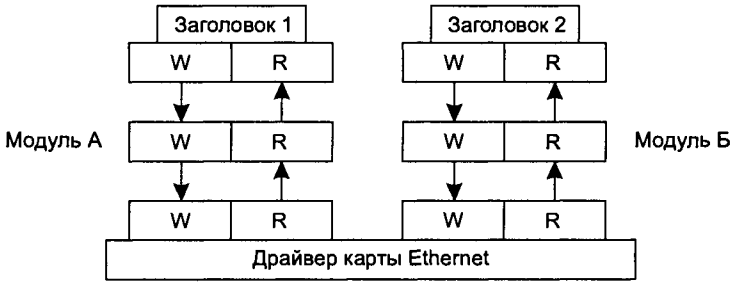


Рис. 17.13. Пример мультиплексора разветвления по входу

Подсистема STREAMS не предлагает какой-либо поддержки мультиплексоров по входу. Драйвер управляет структурами данных и отслеживает по содержащейся в них информации, какие потоки присоединены к этому драйверу. В структурах хранятся указатели на различные очереди чтения, таким образом драйвер имеет возможность посылать данные в любой поток. Забота об управлении потоками данных лежит на драйвере, так как *подсистема STREAMS не предоставляет никаких средств управления данными для мультиплексоров.*

17.8.2. Мультиплексирование по выходу

Мультиплексоры разветвления по выходу являются драйверами псевдоустройств. Их интерфейсы не управляют какими-либо физическими устройствами, а взаимодействуют с одним или несколькими потоками. Для формирования такой конфигурации процесс создает потоки разветвления по входу и выходу, и затем соединяет поток разветвления по входу с несколькими по-

токами разветвления по выходу. Подсистема STREAMS поддерживает для этой цели специализированные команды I_LINK и I_UNLINK вызова ioctl, используемые для установки и рассоединения мультиплексоров по входу.

Рассмотрим создание мультиплексора на примере. Представьте, что к системе одновременно подключены карты Ethernet и FDDI. Для каждой из них имеется драйвер STREAMS. В этом случае можно реализовать в системе уровень IP в виде мультиплексирующего драйвера и присоединить его к обоим интерфейсам. Листинг 17.2 демонстрирует создание такой конфигурации.

Листинг 17.2

```
fd_enet = open ("/dev/enet, O_RWDR");
fd_fddi = open ("/dev/fddi, O_RWDR");
fd_ip = open ("/dev/ip, O_RWDR");
ioctl (fd_ip, I_LINK, fd_inet);
ioctl (fd_ip, I_LINK, fd_fddi);
```

В этом примере пропущена часть строк, отвечающая за проверку возвращаемых величин и обработку ошибок. Первые три строки открывают драйверы enet (Ethernet), fddi и ip соответственно. В следующих строках производится соединение потока ip с потоком enet, а также с потоком fddi. Результат выполнения кода показан на рис. 17.14.

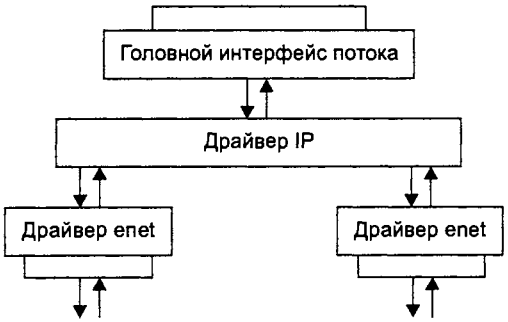


Рис. 17.14. Драйвер IP, выступающий в роли мультиплексора по выходу

В следующем разделе мы рассмотрим процесс настройки мультиплексора по выходу более подробно.

17.8.3. Связывание потоков

Драйвер мультиплексора по выходу должен быть связан с двумя парами очередей, в отличие от обычных драйверов STREAMS, связанных только с одной парой. В мультиплексоре имеется одна *верхняя* (upper) и *нижняя* (lower) пары. В структурах streamtab мультиплексоров поля st_rdinit и st_wrinit указывают на структуры qinit верхней пары очередей, в то время как поля st_muxrinit и st_muxwinit ссылаются на нижнюю пару очередей. Каждой очереди требу-

ется лишь ограниченный набор процедур. Верхняя очередь чтения должна поддерживать вызовы open и close. Нижняя очередь чтения, а также верхняя очередь записи должны иметь реализацию процедуры put. Все остальные процедуры являются необязательными.

На рис. 17.15 показаны потоки ip и enet до выполнения команды I_LINK. Структуры strdata и stwdata разделяются между всеми головными интерфейсами потоков и содержат структуры qinit чтения и записи соответственно. Только драйвер ip имеет нижнюю пару очередей, которая пока не используется.

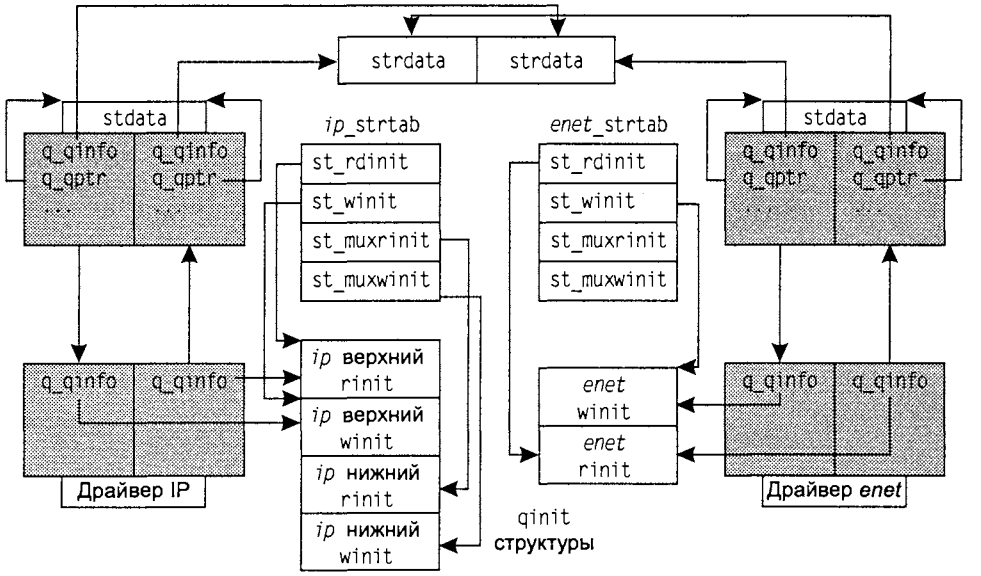


Рис. 17.15. Потоки ip и enet перед связыванием

Посмотрим, что произойдет с потоками при выполнении команды I_LINK. Процедура strioctl() производит действия по инициализации всех вызовов ioctl. В случае команды I_LINK предпринимаются действия, указанные ниже.

1. Проверка верхних и нижних потоков на корректность, а также верхнего потока, который должен быть мультиплексором.
2. Проверка потоков на циклы. Зацикливание потоков может произойти, если нижний поток ранее уже был присоединен к верхнему потоку напрямую или через несколько промежуточных потоков. Подсистема STREAMS завершает выполнение вызова I_LINK с ошибкой, если результат приводит к возникновению зацикливания.
3. Изменение очередей головного интерфейса потока enet, которые должны указывать на нижнюю пару очередей драйвера ip.
4. Обнуление полей q_ptr головного интерфейса потока enet, после чего они перестают ссылаться на структуру stdata.

5. Создание структуры linkblk, содержащей указатели на связываемые очереди: q_top указывает на очередь записи драйвера ip, q_bot указывает на очередь записи головного интерфейса потока enet. Структура linkblk также содержит *идентификатор соединения* (link ID), который в дальнейшем может быть использован при маршрутизации. Подсистема STREAMS поддерживает уникальный идентификатор для каждого соединения и передает его процессу в качестве возвращаемого параметра вызова I_LINK ioctl.
6. Отправка linlblk по потоку в нисходящем направлении драйверу ip внутри сообщения M_IOCTL и ожидание его возвращения.

На рис. 17.16 показана конфигурация соединений после завершения выполнения I_LINK. **Жирными** стрелками обозначены *новые* соединения, установленные STREAMS.

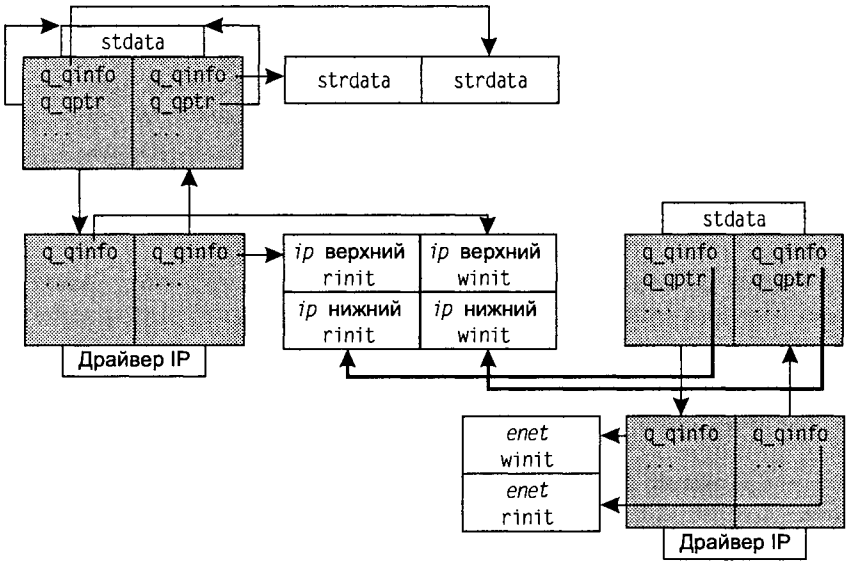


Рис. 17.16. Потоки ip и enet после связывания

Драйвер ip распоряжается остальными элементами конфигурации мультиплексора. Он поддерживает структуры данных, описывающие все нижние по отношению к нему потоки. При получении сообщения M_IOCTL драйвер добавляет в эти структуры новые вхождения, описывающие поток enet. Это вхождение должно, как минимум, содержать указатель на нижнюю очередь и идентификатор соединения (извлекаемые из структуры linkblk, находящейся внутри сообщения). Указанный минимум переменных позволяет передавать сообщения в нисходящем направлении. В следующем разделе мы рассмотрим, как происходит транспортировка данных через мультиплексор.

17.8.4. Потоки данных

Поток fddi подключается к драйверу ip точно так же, как и поток enet. Драйвер ip получает второе сообщение M_IOCTL и добавляет новое вхождение для потока fddi. После настройки результирующая конфигурация должна корректно маршрутизировать входящие и исходящие сообщения.

Если процесс посылает данные в нисходящем направлении, драйверу ip необходимо решить, куда отправить полученную информацию — на карту Ethernet или FDDI. Принятое решение может быть основано на IP-адресе получателя пакета, если службы обслуживаются разными подсетями. Затем драйвер находит адрес очереди записи соответствующего нижнего потока в его закрытых структурах данных. Далее он вызывает canput() для проверки готовности принятия данных потоком и, если получает положительный ответ, вызывает процедуру put нижней очереди записи.

При настройке последовательности потоков, производимой подсистемой STREAMS, обеспечивается гарантия корректности прохождения данных в восходящем направлении. Если данные приходят из карты Ethernet или FDDI, драйвер посылает их дальше наверх. Через некоторое время данные достигают головного интерфейса потока. Здесь они обрабатываются процедурой put очереди чтения головного интерфейса. Однако эта очередь содержит указатель на структуру qinit нижней очереди чтения мультиплексора. Следовательно, полученное сообщение будет обработано драйвером ip, который перешлет его обратно в направлении головного интерфейса потока.

Подсистема STREAMS не обладает прямой поддержкой управления потоком данных для мультиплексоров. Следовательно, эта задача должна быть выполнена самим драйвером ip.

17.8.5. Обычные и постоянные соединения

Обычное соединение сохраняется до тех пор, пока не будет закрыта последняя копия потока. Альтернативным вариантом является принудительное отсоединение потока процессом через команду I_UNLINK вызова ioctl, в качестве параметра которой передается идентификатор соединения, возвращенный ранее I_LINK.

Для конфигураций с применением мультиплексоров поток разветвления по выходу драйвера может оказаться управляющим. При построении многоуровневых конфигураций участвуют несколько мультиплексоров. Управляющий поток каждого мультиплексора на каждом уровне должен соединяться с мультиплексором следующего уровня. Если потоки установлены корректно, вызов процедуры close для управляющего потока самого верхнего уровня приводит к демонтированию всей цепочки.

Представим одну из сложных конфигураций потоков, например набор потоков, представленный на рис. 17.3. Наиболее приемлемым вариантом явля-

ется однократная настройка такой конфигурации, которая будет в дальнейшем постоянно доступна для приложений. Одним из способов достижения этой цели является использование процесса-демона, который производит открытие и связывание всех потоков, а также помещает в них любые необходимые модули. Затем этот процесс переходит в состояние длительной блокировки, оставляя открытыми дескрипторы управляемых им потоков. При этом конфигурация не будет демонтирована, даже если любые другие процессы перестанут ее использовать.

Процессы могут начать использование конфигурации при помощи вызовов open драйверов наивысшего уровня (в примере на рис. 17.3 это драйверы TCP и UDP). Такие устройства, как правило, поддерживают клонирование, и поэтому их открытие приводит к созданию новых младших номеров устройств (и, соответственно, новых потоков того же драйвера).

Описанное решение требует отдельного процесса для поддержания открытого потока. При этом не существует какой-либо защиты от уничтожения процесса по какой-либо причине. Подсистема STREAMS поддерживает альтернативный выход из проблемной ситуации. Вместо команд I_LINK и I_UNLINK применяются команды I_PLINK и I_PUNLINK. Команда I_PLINK создает *постоянные соединения*, которые будут оставаться активными даже в том случае, если поток не открыт ни одним процессом. Такие соединения могут быть принудительно демонтированы при вызове команды I_PUNLINK с аргументом, содержащим идентификатор соединения, возвращенный ранее командой I_PLINK.

17.9. FIFO и каналы

Одним из выигрышных качеств подсистемы STREAMS является простота реализации файлов FIFO и каналов. О файлах FIFO и каналах с точки зрения подсистемы взаимодействия процессов можно прочесть в разделе 6.2. В этой главе вы прочитаете описание реализации этих объектов в SVR4, а также увидите преимущества нового подхода.

17.9.1. Файлы FIFO STREAMS

Файлы FIFO также называют именованными каналами (named pipes). Процесс создает FIFO при помощи системного вызова mknod, передавая ему полное имя, разрешения и флаг S_IFIFO в качестве параметров. Файл может находиться в каталоге любой обычной файловой системы, например s5fs или ufs. После создания файла FIFO каждый процесс, знающий его имя, может производить в такой файл чтение или запись (если он обладает соответствующими полномочиями). Файл продолжает существовать в системе до тех пор, пока не будет удален принудительно системным вызовом unlink. Ввод-вывод таких файлов осуществляется в порядке FIFO — «первым вошел, первым

вышел». Таким образом, после открытия файл FIFO функционирует почти как обычный канал (конвейер).

В SVR4 все операции с файлами FIFO осуществляются в файловой системе под названием fifofs. При реализации FIFO в этой системе используются потоки и драйвер *обратной связи* (loopback). Если процесс вызывает mknod для создания FIFO, ядро просматривает полное имя с целью получения vnode родительского каталога. Затем оно вызывает операцию VOP_CREATE родительского объекта vnode для создания файла в данном каталоге. Ядро устанавливает в индексном дескрипторе файла флаг IFIFO, указывая тем самым, что файл имеет тип FIFO.

На рис. 17.17 показана настройка файла FIFO в системе SVR4. Для использования файла процессу необходимо первоначально его открыть. Если системный вызов open обнаруживает в индексном дескрипторе флаг IFIFO, то вызывает операцию specvp(), которая, в свою очередь, запускает процедуру fifofs под названием fifovp(). Эта процедура создает объект *fifonode*, во многом сходный с snode. Объект vnode, находящийся внутри fifonode, содержит указатели на вектор операций fifofs. Файловая система fifofs возвращает этот объект vnode вызову open. Теперь все последующие ссылки на файл приведут к использованию операций fifofs.

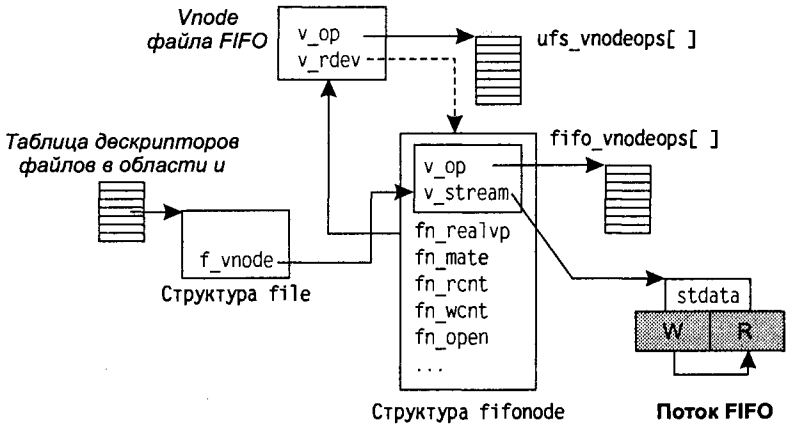


Рис. 17.17. Файлы FIFO в подсистеме STREAMS

После этого системный вызов open запускает операцию VOP_OPEN нового объекта vnode. Это приводит к вызову процедуры fifo_open(). Так как файл открывается впервые, с ним еще не ассоциировано ни одного потока. Процедура fifo_open() создает новый головной интерфейс потока и присоединяет его очередь записи к его же очереди чтения. Процедура сохраняет указатель на головной интерфейс потока (структуру stdata) и поле v_stream объекта vnode. При последующих открытиях файла процедура fifo_open() обнаружит, что поток уже существует, и все пользователи разделяют доступ к нему.

Когда процесс записывает данные в файл FIFO, головной интерфейс потока отправляет их в нисходящем направлении в очередь записи, которая незамедлительно пересылает их обратно в очередь чтения, где они ожидают прочтения. Читающие процессы запрашивают данные из очереди чтения и блокируются головным интерфейсом потока, если данные отсутствуют. Если ни один из процессов больше не открывает файл FIFO, поток демонтируется. Поток будет заново создан при следующем открытии файла. Сам файл FIFO существует в системе, пока не будет удален принудительно.

17.9.2. Каналы STREAMS

Для создания неименованного канала применяется системный вызов pipe. До появления SVR4 поток данных в конвейерах не был направленным. Системный вызов pipe возвращал два дескриптора, один для записи данных, а другой — для чтения. В SVR4 реализация каналов была пересмотрена с учетом возможностей STREAMS. В этой версии системы каналы являются двунаправленными.

Как и раньше, системный вызов pipe возвращает два дескриптора. Однако в SVR4 они оба могут быть использованы для чтения и записи. Данные, записанные в один из дескрипторов, читаются из другого и наоборот. Такая возможность достигается применением пары потоков. Системный вызов pipe создает два объекта fifonode, а также головной интерфейс потока для каждого из них. Затем он соединяет очереди так, чтобы очередь записи каждого головного интерфейса связывалась с очередью чтения другого головного интерфейса. Получаемая в результате этих действий конфигурация показана на рис. 17.18.

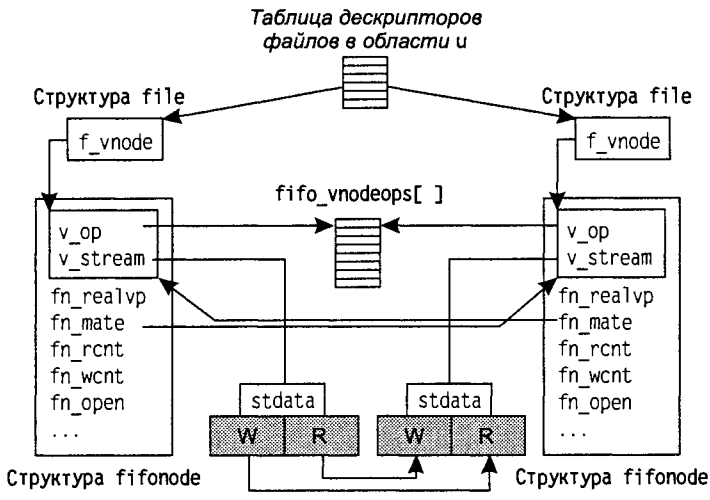


Рис. 17.18. Каналы в STREAMS

Такой подход обладает несколькими важными преимуществами. Канал стал двунаправленным, что расширило область его применения. Двунаправленные коммуникации между процессами требуются для многих приложений. До появления SVR4 приходилось открывать сразу два канала. Более того, реализация каналов через потоки позволила использовать большее количество управляющих операций. Например, такой канал может быть доступен для не связанного с ним процесса.

Такую возможность предоставляет процедура библиотеки C fattach. Ее синтаксис представлен ниже:

```
error = fattach (fd, path);
```

где fd — файловый дескриптор, ассоциируемый с потоком, а параметр path является полным именем файла, владельцем которого должен быть вызывающий процесс (либо вызывающий процесс обязан обладать привилегиями root). Вызывающий процесс должен иметь право на запись файла. Процедура fattach использует специализированную файловую систему под названием *namefs* и монтирует копию этой файловой системы в файл, представленный path. В отличие от других файловых систем, которые способны монтироваться только в каталоги, namefs может монтироваться в обычных файлах. При монтировании она связывает дескриптор файла потока fd с точкой монтирования.

После присоединения любая ссылка на данное полное имя приведет к обращению к соответствующему ему потоку. Такая ассоциация просуществует до тех пор, пока не будет сброшена процедурой fdetach. После завершения работы fdetach полное имя снова будет ссылаться она оригинальный файл. Процедура fdetach часто применяется для связывания одного конца канала с полным именем. Это позволяет приложениям создавать каналы и затем динамически ассоциировать их с полными именами, предоставляя таким образом доступ к этим каналам любым процессам.

17.10. Сетевые интерфейсы

Подсистема STREAMS предоставляет инфраструктуру ядра для сетевой поддержки в System V UNIX. Разработчики программ нуждаются в высокоуровневом интерфейсе для написания сетевых приложений. Базовая структура сокетов, представленная в 4.1cBSD в 1982 году, предлагает недостаточную поддержку интранет-программирования. В System V UNIX эти трудности решаются через набор интерфейсов, расположенных на верхнем уровне STREAMS. Одним из них является *интерфейс поставщиков транспорта* (Transport Provider Interface, TPI), определяющий взаимодействия между поставщиками транспорта и их пользователями. Еще один интерфейс получил название *интерфейса транспортного уровня* (Transport Layer Interface, TLI). Он обеспечивает средства высокоуровневого программирования. Так как сокеты появились задолго до подсистемы STREAMS, для них было соз-

дано множество приложений. Для упрощения задачи переноса таких приложений в систему SVR4 была добавлена поддержка сокетов в виде набора библиотек и модулей STREAMS.

17.10.1. Интерфейс поставщиков транспорта (TPI)

Поставщик транспорта — это сетевой модуль (такой как TCP), реализующий четвертый (транспортный) уровень в сетевой модели OSI [4]. Пользователем транспорта является приложение, например, протокол передачи файлов ftp. Интерфейс TPI построен на основе STREAMS и определяет формат и содержимое сообщений, доставляемых при взаимодействии поставщика транспорта с пользователем этого транспорта.

Сообщения TPI могут исходить как от приложений, так и от поставщиков. Каждое такое сообщение находится внутри блока сообщений STREAMS типов M_PROTO и M_PCPROTO. В первом поле сообщения указывается тип сообщения TPI. Этот тип определяет формат и содержимое остальной части сообщения. Например, приложение может создать сообщение T_BIND_REQ для связывания потока с портом. Такое сообщение содержит номер порта, к которому необходимо присоединить поток, а также набор других параметров, специфичных для данного запроса. Поставщик транспорта отвечает на запрос посредством отправки сообщения T_BIND_ACK, содержащего результаты выполнения операции.

Интерфейс TPI также определяет формат передачи данных по потоку вверх или вниз. Приложение создает сообщение с заголовком T_DATA_REQ или блоком T_UNITDATA_REQ, после которого следует один или несколько блоков M_DATA, в которых находится тело сообщения. Его заголовок содержит адрес получателя и номера порта в формате, специфичном для используемого протокола. После прохождения данных по сети поставщик транспорта добавляет к сообщению заголовок T_DATA_IND или блок сообщения T_UNITDATA_IND[1].

Интерфейс TPI играет роль стандарта взаимодействий между поставщиком транспорта и его пользователями. Например, приложение может создать одно и то же сообщение для присоединения к порту, используя как TCP-, так и UDP-соединение. Такой подход повышает независимость транспортных протоколов. Однако TPI не обладает простым программным интерфейсом. Это заложено в подсистему сокетов и интерфейс транспортного уровня.

17.10.2. Интерфейс транспортного уровня (TLI)

Интерфейс TLI [1] изначально появился в System V UNIX и позже был реализован в SVR3 в 1986 году. Он является процедурным интерфейсом для открытия и использования сетевых соединений. Функции TLI частично приме-

[1] Типы полей T_UNITDATA_REQ и T_UNITDATA_IND используются для дейтаграмм, в то время как типы T_DATA_REQ и T_DATA_IND применяются для потоков байтов.

нимы для реализации взаимодействия типа «клиент-сервер». Они могут быть использованы как для ориентированных на соединение протоколов, так и для служб без установления соединения. Функции TLI реализованы на основе операций STREAMS.

В протоколах, использующих соединение (см. рис. 17.19), сервер открывает конечную транспортную точку посредством вызова t_open() и затем подключает ее к порту при помощи вызова t_bind(). Затем запускается процедура t_listen(), где процесс блокируется до тех пор, пока клиент не запросит соединение. В тот же промежуток времени клиентская программа, как правило, находящаяся на другой машине в сети, вызывает t_open() и t_bind(), после чего использует процедуру t_connect() для создания соединения с сервером. При выполнении последней процедуры клиент блокируется до того момента, пока соединение не будет создано.

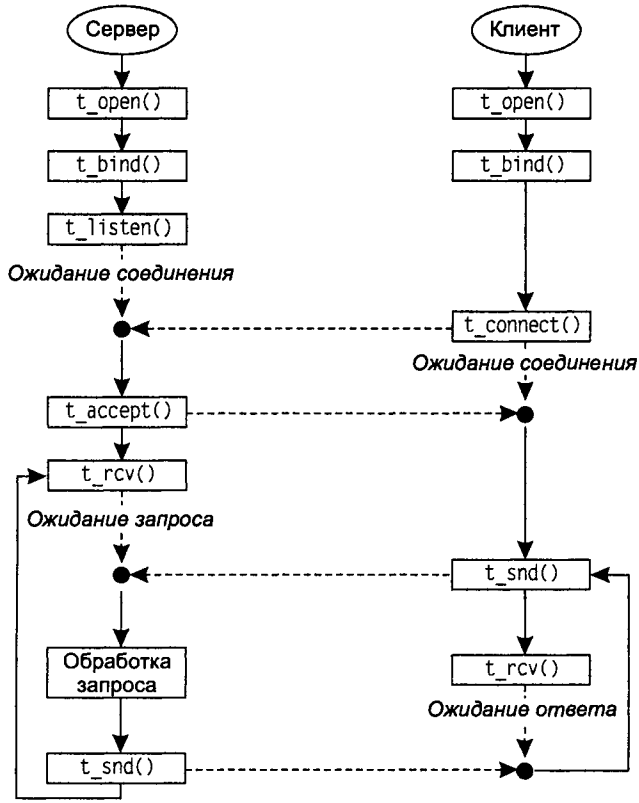


Рис. 17.19. Функции TLI для протоколов, ориентированных на соединение

После того как запрос на соединение прибывает на сервер, функция t_listen() завершает выполнение. Затем сервер вызывает t_accept() для подтверждения установления соединения. Процедура отправляет ответное сообщение кли-

енту, который, в свою очередь, выходит из процедуры t_connect(). На этот момент работы соединение установлено. Далее сервер переходит в цикл ожидания, вызывая t_rcv() для получения клиентских запросов, обрабатывает их и вызывает t_snd() для отправки ответного сообщения. Клиент использует t_snd() для отправки запросов и t_rcv () для получения ответных сообщений.

Протоколы без установления соединений работают несколько по-другому (рис. 17.20). Сервер и клиент также используют вызовы t_open() и t_bind (). Однако нам не нужно создавать соединение, поэтому нет необходимости в применении вызовов t_listen(), t_connect() или t_accept(). Вместо этого сервер выполняет цикл ожидания, вызывая периодически функцию t_rcvudata(), которая блокирует его до отправки сообщения клиентом. После приема сообщения управление возвращается серверу с указанием на адрес и порт отправителя и телом сообщения. Сервер обрабатывает сообщение и отвечает клиенту при помощи вызова t_sndudata(). Клиент также использует вызов t_sndudata() для посылки сообщений и t_rcvdata() для получения ответов.

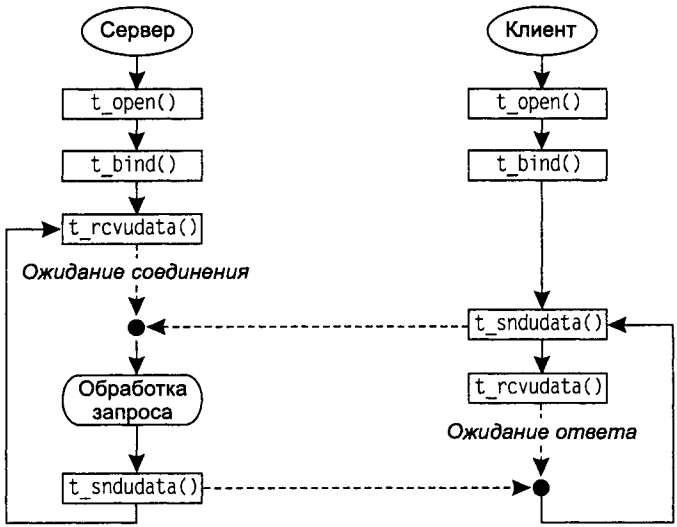


Рис. 17.20. Функции TLI для протоколов без установления соединения

17.10.3. Сокеты

Сокеты [5] были впервые представлены в 4.1BSD в 1982 году. Они предоставляют программный интерфейс, который предназначен как для взаимодействия процессов, так и для сетевых коммуникаций. *Сокет* — это конечная точка соединения, представленная в виде абстрактного объекта, который может быть использован процессом для отправки и получения сообщений. Несмотря на то, что сокеты не являются привычными для System V UNIX, в системе SVR4 поддерживаются все средства сокетов BSD, что позволяет задейство-

вать большой набор приложений, созданных с привлечением интерфейса сокетов.

Интерфейс сокетов по всем параметрам напоминает TLI. Между функциями интерфейса TLI и сокетами существует практически полное соответствие. Таблица 17.1 показывает общецелевые функции TLI и эквивалентные вызовы сокетов.

Таблица 17.1. Функции TLI и соответствующие им вызовы сокетов

Функции TLI	Функции сокетов	Назначение
T_open()	socket()	Открытие конечной точки соединения
T_bind()	bind()	Связывание конечной точки с портом
T_listen()	listen()	Запрос ожидания соединения
T_connect()	connect()	Отправка запроса на соединение
T_accept()	accept()	Подтверждение соединения
T_rcv()	recvmsg()	Получение сообщения из соединения
T_snd()	sendmsg()	Отправка сообщения в соединение
T_rcvudata()	recvfrom()	Получение сообщения от любого узла
T_sndudata()	sendto()	Отправка сообщения на определенный узел

Невзирая на перечисленные сходства, параметры и семантика вызовов TLI и сокетов резко отличаются друг от друга. Несмотря на то, что интерфейсы TLI и сокеты разрабатывались с учетом поддержки друг друга, при реализации сокетов в подсистеме STREAMS возникают определенные проблемы [12]. Рассмотрим подробнее некоторые факты, ставшие причиной несовместимости между двумя интерфейсами.

Сокеты используют процедурный интерфейс взаимодействия, не прибегающий к сообщениям. Если приложение вызывает функцию сокетов, ядро посылает данные в сеть посредством прямого вызова низкоуровневых транспортных функций. Эти функции выбираются при помощи просмотра таблиц, после чего им перенаправляются данные. Такой подход позволяет более высоким уровням интерфейса сокетов разделять информацию о состоянии с транспортными уровнями через глобальные структуры данных. В подсистеме STREAMS каждый модуль наследуется от другого и не имеет никаких глобальных состояний. Несмотря на то, что такая модульная структура обладает множеством преимуществ, с ее помощью сложно реализовать некоторые функции сокетов, зависящих от глобальных состояний.

Вызовы сокетов осуществляются в контексте вызывающего их процесса. Следовательно, все ошибки передаются процессу синхронно. Подсистема STREAMS обрабатывает данные несинхронно. Такие вызовы, как write или putmsg, выполняются после того, как данные окажутся в головном интерфейсе потока. Если модуль нижнего уровня генерирует ошибку, то это повлияет

лишь на последующие попытки записи. Вызов, ставший причиной ошибки, будет завершен успешно.

Некоторые проблемы возникают и при попытке реализовать сокеты на основе TPI. Однако некоторые опции интерфейса сокетов применяются в других точках выполнения, чем это производится с помощью TPI. Например, приложения сокетов указывают максимальное количество не принятых соединений (backlog) в системном вызове listen() после открытия и связывания сокета. Интерфейс TPI в такой же ситуации требует указания этого параметра в сообщении T_BIND_REQ, посылаемом во время операции связывания.

17.10.4. Реализация сокетов в SVR4

На рис. 17.21 показана реализация сокетов в системе SVR4. Средства сокетов предоставляются библиотекой прикладного уровня socklib и модулем STREAMS под названием sockmod. Эти два компонента дополняют друг друга. Библиотека socklib переводит вызовы сокетов в системные вызовы и сообщения STREAMS. Модуль sockmod отвечает за взаимодействия с транспортным протоколом и поддерживает специфическую для интерфейса сокетов семантику.

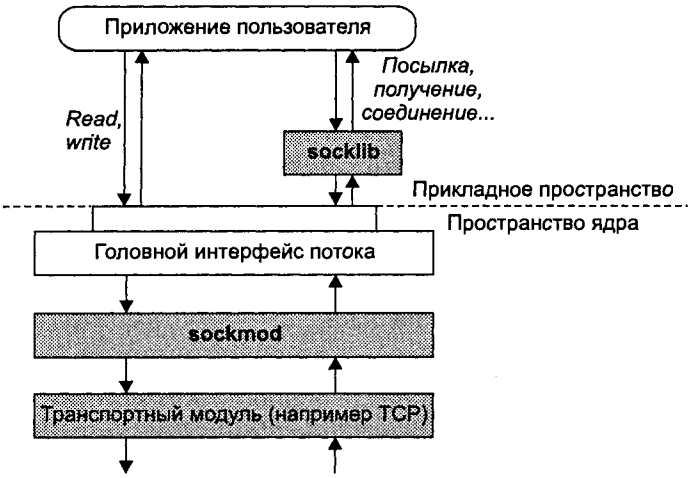


Рис. 17.21. Применение сокетов в SVR4

Если процесс вызывает socket для создания сокета, библиотека socklib производит отображение параметров вызова в имя файла устройства, используя для этой цели *средство выбора сети SVR4* (Network Selection Facility) [1]. Библиотека открывает файл, создавая таким образом поток, и вызывает команду I_PUSH ioctl для помещения модуля sockmod сразу под головным интерфейсом потока. После создания такой конфигурации библиотека socklib и модуль sockmod взаимодействуют для обработки прикладных запросов.

Представим на примере, что может произойти при вызове процессом функции connect для присоединения потока TCP к удаленному серверу. Библиотека создает сообщение T_CONN_REC интерфейса TPI и отправляет его в нисходящем направлении по потоку посредством вызова putmsg. Затем она вызывает getmsg для ожидания ответа. После того, как модуль sockmod перехватит сообщение, он производит сохранение адреса приемника для дальнейшего использования и посылает сообщение TCP. Модуль TCP обрабатывает запрос и отправляет по потоку подтверждение. После возврата функции getmsg библиотека socklib производит извлечение результатов из сообщения и передает управление приложению.

Далее процесс может отправлять данные в установленное соединение при помощи вызова sendmsg. Это сообщение снова будет передаваться по потоку в нисходящем направлении библиотекой socklib через вызов putmsg. После того как модуль sockmod получит сообщение, он добавит к нему заголовок, содержащий адрес получателя, который был запомнен при создании соединения.

Библиотека и модуль должны хранить некоторую информацию о состоянии сокета. Например, при создании соединения библиотека socklib записывает статус соединения и адрес получателя. Это позволяет не обрабатывать вызовы sendmsg для неприсоединенных сокетов. Если информация о соединении будет поддерживаться только sockmod, вызывающий процесс не сможет получить корректный код ошибки.

Также недостаточно хранить информацию о состоянии только в библиотеке socklib. Процесс вправе создать сокет и затем вызвать exec, изменив тем самым состояние, указанное в socklib. Реализация сокетов способна обнаруживать такие изменения, так как вызов exec инициализирует socklib в известном состоянии. Если процесс пытается воспользоваться сокетом после выполнения exec, библиотека socklib посылает сообщение ioctl модулю sockmod для восстановления статуса-кво. Так как модуль sockmod размещается в пространстве ядра, его состояние не нарушается при выполнении exec.

Кроме освещенных в этом разделе вопросов, существуют и другие интересные детали и проблемы реализации сокетов в операционной системе SVR4. Более подробно они излагаются в работе [12].

17.11. Заключение

Подсистема STREAMS обеспечивает базовую структуру для написания драйверов устройств и сетевых протоколов. Создаваемые программы обладают высокой настраиваемостью и модульностью. Подсистема STREAMS предоставляет для драйверов то же самое, что и каналы UNIX для команд интерпретаторов. Она позволяет создавать независимые модули, каждый из которых

играет роль фильтра и производит некоторые специфические действия над потоком данных. Подсистема позволяет комбинировать отдельные модули различным образом для построения потока. Такой поток работает как двунаправленный канал, передавая данные между приложением и устройством (или сетевым интерфейсом) и производя необходимые с ними действия в промежуточных модулях.

Модульная архитектура потоков STREAMS позволила использовать сетевые протоколы соответственно уровням. На каждом таком уровне находится отдельный модуль. Подсистема STREAMS также применяется для взаимодействия процессов. В SVR4 каналы и файлы FIFO были реализованы на основе потоков. Более того, многие символьные драйверы, в том числе и подсистема драйверов терминалов, были переписаны с использованием STREAMS. Одним из последних дополнений в подсистему STREAMS стала многопроцессорная поддержка [3], [9].

17.12. Упражнения

1. Почему в STREAMS применяются отдельные структуры msgb и datab вместо использования единого заголовка буфера?
2. В чем проявляются различия между драйвером STREAMS и модулем STREAMS?
3. Какие существуют взаимосвязи между двумя очередями модуля? Должны ли они выполнять одинаковые функции?
4. Как влияет наличие или отсутствие процедуры service на поведение очереди?
5. Для получения данных из потока могут быть выполнены два системных вызова — read и getmsg. Объясните разницу между ними. В каких ситуациях более целесообразно применять какой-либо определенный вызов?
6. Почему большинство процедур STREAMS не имеет права блокировки? Что может сделать процедура put, если не способна обработать сообщение незамедлительно?
7. Чем полезно применение уровней приоритетов?
8. Объясните, как и в каких случаях очередь становится вновь доступной (back-enabled)?
9. Какие действия производит головной интерфейс потока? Почему все головные интерфейсы разделяют между собой один и тот же набор процедур?
10. Почему драйверы STREAMS доступны через таблицу cdevsw?

11. По какой причине устройства STREAMS требуют специализированной поддержки при помощи команд ioctl? Почему в потоке может быть активен только один вызов ioctl?
12. Объясните, почему сброс входящих сетевых сообщений часто является лучшим решением при отсутствии свободной памяти?
13. Чем отличается мультиплексор от модуля, используемого независимо в двух различных потоках?
14. Почему подсистема STREAMS не поддерживает управление потоками данных для мультиплексоров?
15. Почему показанный на рис. 17.14 уровень IP реализован как драйвер STREAMS, а не как модуль?
16. Какими преимуществами обладают постоянные соединения?
17. Каналы STREAMS позволяют двунаправленную передачу данных, в то время как это невозможно в обыкновенных неименованных каналах. Опишите приложение, которое бы могло воспользоваться преимуществами нового средства системы. Каким образом можно реализовать такое же средство без привлечения каналов STREAMS?
18. Укажите различия между интерфейсами TPI и TLI. За какие взаимодействия отвечает каждый из них?
19. Сравните сокеты и интерфейс TLI в качестве базовых структур для создания сетевых приложений. Какими преимуществами и недостатками будет обладать каждый подход? Какие средства одного интерфейса могут быть легко перенесены на другой?
20. В разделе 17.10.4 описывается применение сокетов в SVR4 на основе подсистемы STREAMS. Можно ли реализовать в BSD-подобных системах интерфейс, похожий на STREAMS, через сокеты? Какие важные вопросы при этом необходимо решить?
21. Создайте модуль STREAMS, который преобразует все символы новой строки в комбинацию «возврат каретки + перевод строки» (CR+LF) при следовании данных в восходящем направлении, и выполняющий обратное преобразование при следовании данных в обратную сторону. Представьте, что сообщения могут содержать только печатаемые ASCII-символы.
22. Процесс вправе настроить поток динамически путем помещения необходимого числа модулей в стек. Каждый модуль не знает о том, какой модуль находится выше или ниже его. Каким образом модуль будет различать, как интерпретировать сообщение, полученное из соседнего модуля? Какие ограничения существуют при выборе собираемых вместе модулей и в порядке их следования? Как интерфейс TPI решает данную проблему?

17.13. Дополнительная литература

1. American Telephone and Telegraph, «UNIX System V Release 4 Network Programmer's Guide», 1991.
2. American Telephone and Telegraph, «UNIX System V Release 4 Internals Students Guide», 1991.
3. Garg, A., «Parallel STREAMS: A Multi-Processor Implementation», Proceedings of the Winter 1990 USENIX Technical Conference, Jan. 1990.
4. International Standards Organization, «Open Systems Interconnection — Basic Reference Model», ISO 7498, 1984.
5. Leffler, S., Joy, W., Fabry, R., and Karels, M., «Networking Implementation Notes — 4.3BSD Edition», University of California, Berkeley, CA, Apr. 1986.
6. Presotto, D. L., and Ritchie, D. M., «Interprocess Communications in the Ninth Edition UNIX System», UNIX Research System Papers, Tenth Edition, Vol. II, Saunders College Publishing, 1990, pp. 523–530.
7. Rago, S., «Out-of-band Communication in STREAMS», Proceedings of the Summer 1989 USENIX Technical Conference, Jun. 1989, pp. 29–37.
8. Ritchie, D. M., «A Stream Input-Output System», AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Oct. 1984, pp. 1897–1910.
9. Saxena, S., Peacock, J. K., Verma, V., and Krishnan, M., «Pitfalls in Multithreading SVR4 STREAMS and Other Weightless Processes», Proceedings of the Winter 1993 USENIX Technical Conference. Jan. 1993, pp. 85–95.
10. UNIX System Laboratories, «STREAMS Modules and Drivers, UNIX SVR4.2», UNIX Press, Prentice-Hall, Englewood Cliffs, NJ, 1992.
11. UNIX System Laboratories, «Operating System API Reference, UNIXSVR4.2», UNIX Press, Prentice-Hall, Englewood Cliffs, NJ, 1992.
12. Vessey, I., and Skinner, G., «Implementing Berkeley Sockets in System V Release 4», Proceedings of the Winter 1990 USENIX Technical Conference, Jan. 1990, pp. 177–193.

Алфавитный указатель

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

А

Б

В

Г

Д

Ж

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ш

Э

Юреш Вахалия

UNIX изнутри

Перевели с английского Е. Васильев, Л. Серебрякова

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Руководитель проекта	*В. Шачин*
Научные редакторы	*М. Астапов, М. Иванов*
Литературный редактор	*Е. Васильев*
Художник	*Н. Биржаков*
Иллюстрации	*М. Жданова, В. Шендерова, М. Шендерова*
Корректоры	*Н. Лукина, А. Моносов*
Верстка	*Р. Гришанов*

Лицензия ИД № 05784 от 07.09.01.
Подписано в печать 30.06.03. Формат 70×100/16. Усл. п. л. 68,37.
Тираж 3000 экз. Заказ № 317.
ООО «Питер Принт». 196105, Санкт-Петербург, ул. Благодатная, д. 67в
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953005 – литература учебная.
Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М Горького
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций
197110, Санкт-Петербург, Чкаловский пр., 15.