Классика Computer Science

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Разработка и реализация

Э. ТАНЕНБАУМ, А. ВУДХАЛЛ

CD-ROM ПРИЛАГАЕТСЯ

PH PTR

ПИТЕР

A. Tanenbaum, A. Woodhull

Operating Systems: Design and Implementation

КЛАССИКА COMPUTER SCIENCE

Э. ТАНЕНБАУМ, А. ВУДХАЛЛ

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Разработка и реализация

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Новосибирск · Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2006

ББК 32.973-018.2
УДК 004.451
Т16

Таненбаум Э., Вудхалл А.
Т16 Операционные системы: разработка и реализация (+CD). Классика CS. — СПб.: Питер, 2006. — 576 с.: ил.

ISBN 5-469-00148-2

Второе издание классического труда Эндрю Таненбаума «Operating Systems: Design and Implementation» — это единственный в своем роде учебник, в котором успешно сочетаются теория и практика построения операционных систем. В ней подробно описываются процессы и межпроцессное взаимодействие, семафоры, мониторы, передача сообщений, алгоритмы работы планировщика, ввод/вывод, разрешение тупиковых ситуаций, драйверы устройств, алгоритмы управления памятью, разработка файловых систем, а также затрагиваются вопросы безопасности и защиты данных. Обсуждается и конкретная UNIX-совместимая операционная система MINIX и приводится ее исходный код (его вы найдете на компакт-диске). Это позволяет не только изучать основополагающие принципы, но и наблюдать, как они применяются в реальных операционных системах.

ББК 32.973-018.2
УДК 004 451

ISBN 0136386776 (англ)
ISBN 5-469-00148-2

Краткое содержание

Содержание

Глава 4. Управление памятью 341

Об авторах

Эндрю Таненбаум (Andrew S. Tanenbaum) получил степень бакалавра в Массачусетсском технологическом институте и степень доктора наук в Калифорнийском университете в Беркли. Он является профессором кибернетики в университете Врийе (Vrije) в Амстердаме, где возглавляет Группу компьютерных систем. Кроме того, автор является деканом межуниверситетской школы аспирантов по кибернетике и обработке изображений (Advanced School for Computing and Imaging), занимающейся исследованиями в области современных параллельных систем, распределенных систем и систем обработки изображений. Тем не менее он изо всех сил старается не превратиться в бюрократа.

В прошлом автор занимался исследованиями в области компиляторов, операционных систем, компьютерных сетей и локальных распределенных систем. В настоящее время его усилия в основном направлены на разработку глобальных распределенных систем, пользователями которых являются миллионы людей. Результатом этих исследовательских проектов стали более 70 статей в журналах и отчетах конференций. А. Таненбаум является автором пяти книг.

Профессор Таненбаум написал значительное количество программ. Под его руководством разрабатывалась архитектура проекта Amsterdam Compiler Kit — инструмента, широко применяемого для написания кросс-платформенных компиляторов. Кроме того, он руководил созданием учебной операционной системы MINIX — упрощенной версии системы UNIX, предназначенной для обучения студентов. Вместе со своими аспирантами и программистами он помогал в разработке высокопроизводительной распределенной операционной системы Amoeba. В настоящее время системы MINIX и Amoeba свободно распространяются через Интернет, могут использоваться для обучения и открыты для исследований.

Его аспиранты, многие из которых дошли до степени доктора наук, достигли больших успехов. Профессор Таненбаум очень гордится своими учениками. В этом смысле он напоминает курицу-наседку.

Профессор Таненбаум является членом Ассоциации по вычислительной технике (ACM, Association for Computing Machinery), почетным членом Института инженеров по электротехнике и электронике (IEEE, Institute of Electrical and Electronics Engineers), членом Голландской королевской академии искусств и наук. В 1994 году ему была присуждена премия ACM Карла В. Карлстрома (Karl V. Karlstrom) как выдающемуся преподавателю. А. Таненбаум включен в список *Who's Who in the World*. Его домашняя страница в Интернете расположена по адресу http://www.cs.vu.nl/~ast/.

Альберт Вудхалл (Albert S. Woodhull) получил степень бакалавра в Массачусетсском технологическом университете и степень доктора в университете Вашингтона. Поступив в Массачусетсский институт, чтобы стать электротехником, он окончил его как биолог. С 1973 года он был связан со Школой естественных наук Хэмпширского колледжа, Массачусетс. Как биолог, пользующийся электрон-

ным оборудованием, он начал работать с микрокомпьютерами, когда они стали доступны. Его технические курсы для студентов развились в лекции, посвященные взаимодействию и программированию задач реального времени.

Доктор Вудхалл всегда испытывал большой интерес к преподаванию и к вопросам влияния науки и технологии на производство. Перед поступлением в аспирантуру он в течение двух лет преподавал естественные науки в Нигерии. Позже он потратил несколько своих отпусков на обучение студентов вычислительной технике в Национальном университете Никарагуа.

В сферу его интересов входят компьютеры как электронные системы и взаимодействие компьютеров с другими электронными системами. Он особенно наслаждается преподаванием в областях архитектуры вычислительной техники, операционных систем и компьютерных коммуникаций, программирования на языке ассемблер. Также он работает консультантом по разработке электронного оборудования и связанного с ним программного обеспечения.

Помимо этого у него немало других, не академических интересов, включая спортивные игры на открытом воздухе, радиолюбительство и чтение. Он любит путешествовать и изучать другие языки помимо родного английского. Его страничка в Сети управляется MINIX и может быть найдена по адресу http://minix1.hampshire.edu/asw/.

Предисловие

Большинство книг, посвященных операционным системам, в теории сильнее, чем на практике. Та же, которую вы держите в руках, в этом смысле более сбалансирована. Она скрупулезно вникает во все теоретические основы, такие как процессы, взаимодействие между процессами, семафоры, мониторы, передача сообщений, алгоритмы работы планировщика, ввод/вывод, тупиковые ситуации, драйверы устройств, алгоритмы управления памятью и страницами, разработка файловых систем, а также затрагивает вопросы безопасности и защиты данных. Но в то же время обсуждается и конкретная, UNIX-совместимая операционная система MINIX и приводится копия ее исходных кодов (на компакт-диске). Это позволяет не только изучать основополагающие принципы, но и видеть, как эти принципы применяются в реальных операционных системах.

Впервые появившись в 1987 году, эта книга произвела небольшую революцию в понимании того, как нужно изучать операционные системы. До того большинство книг посвящалось только теоретической части. С появлением MINIX во многих школах стали проводить лабораторные занятия, на которых ученики могли «изнутри» увидеть, как работают операционные системы. Мы сочли эту тенденцию весьма желательной и надеемся, что вторая редакция только укрепит ее.

За первые десять лет ОС MINIX претерпела множество изменений. Первоначальный код был рассчитан на IBM PC с процессором 8086 и 2456 Кбайт памяти, с двумя дисководами и без жестких дисков. Эта система основывалась на UNIX Version 7. С течением времени система развивалась в различных направлениях. Например, текущая версия может работать на чем угодно, начиная со старых PC (в 16-битном реальном режиме) и заканчивая современными Pentium с огромными жесткими дисками (в 32-битном защищенном режиме). Кроме того, система теперь базируется не на Version 7, а на международном стандарте POSIX (IEEE 1003.1 и ISO 9945-1). Добавлено множество новых возможностей, на наш взгляд, может быть, даже слишком много. Впрочем, некоторым и этого мало, что и привело к появлению Linux. В дополнение, MINIX была перенесена на множество других платформ, включая Macintosh, Amiga, Atari и SPARC. Эта книга охватывает только версию MINIX 2.0, которая может работать на системах с процессором 80x86, системах, способных эмулировать такой процессор и на SPARC.

Практически каждое место предыдущего издания в чем-то изменилось. Вся теоретическая информация была пересмотрена и добавлены новые важные сведения. Тем не менее основное изменение — это переход к рассмотрению новой, POSIX-совместимой версии MINIX. На прилагающемся к книге компакт-диске имеются все исходные коды, а также инструкции по установке системы (смотрите файл README.TXT в корневом каталоге).

Установка MINIX на PC, как для индивидуального использования, так и для проведения лабораторной практики, — простой последовательный процесс. Для

этого нужно выделить на жестком диске раздел объемом не менее 80 Мбайт и следовать инструкциям из README.TXT. Чтобы напечатать этот файл на PC, загрузите MS-DOS, если она еще на загружена (в Windows — щелкните на значке MS-DOS), и выполните команду copy readme.txt prn.

Просмотреть файл можно с помощью edit, wordpad, notepad или любой другой программы, пригодной для просмотра простых ASCII-файлов.

Для школ (или для пользователей), не имеющих PC, есть два варианта. На компакт-диске есть два эмулятора. Первый (его автор Пол Эштон (Paul Ashton)) предназначен для SPARC. С его помощью MINIX можно запустить как обычное приложение, поверх Solaris. MINIX компилируется в двоичный код SPARC и работает с максимально возможной скоростью. В этом режиме MINIX функционирует не как операционная система, а как пользовательская программа, поэтому необходимы небольшие изменения на низком уровне.

Второй эмулятор создал Кевин П. Лоутон (Kevin P. Lawton) из компании Bosch Software. Эта программа интерпретирует инструкции процессора 80386 и имитирует работу некоторых устройств ввода/вывода, достаточных для функционирования MINIX. Конечно, интерпретация по определению приводит к некоторой потере производительности, но зато значительно упрощает отладку и обучение. Преимущество этого эмулятора в том, что он будет работать на любой системе, поддерживающей X Window. Дополнительную информацию о данных программах вы можете найти на компакт-диске.

Разработка MINIX все время продолжается. На диске система предлагается такой, какой она была на момент публикации. О текущем состоянии дел можно узнать на домашней странице MINIX в Сети, http://www.cs.vu.nl/~ast/minix.html. Кроме того, MINIX посвящена группа новостей Usenet comp.os.minix. Оформив регистрацию, можно всегда быть в курсе, что происходит в мире MINIX. Для тех, у кого имеется электронная почта, но нет доступа к Usenet, есть список рассылки. Напишите письмо с текстом *subscribe minix-1 <ваше полное имя>* на адрес listserv@listserv.nodac.edu. В ответ вам придет письмо с дальнейшими указаниями.

Существует и руководство по устранению проблем, рассчитанное только на преподавателей (издание Prentice Hall). Все рисунки из книги переведены в формат PostScript, их можно использовать для изготовления плакатов. Эти рисунки находятся на веб-странице по адресу http://www.cs.vu.nl/~ast/, щелкните на ссылке *Software and supplementary materia for my booksl.*

В работе над проектом нам помогали множество людей. Прежде всего мы хотели бы поблагодарить Кис Бот (Kees Bot) за то, что он выполнил львиную долю работы про приведению MINIX в соответствие со стандартом и по подготовке установочного пакета. Без его помощи мы бы никогда не сделали это сами. Он лично написал большие куски кода (например, терминальный ввод/вывод POSIX), привел в порядок другие разделы и исправил множество накопившихся за годы ошибок. Спасибо тебе за хорошую работу.

За годы развития ОС MINIX большой вклад в ее развитие внесли Брюс Эванс (Bruce Evans), Филип Хомбург (Philip Homburg), Вилл Роуз (Will Rose) и Майкл Темари (Michael Temari). Множество других людей вложили свою лепту

с помощью конференций. Таких людей очень много, и степень их участия в разработке варьируется в широких пределах, поэтому мы не в силах перечислить даже часть из них. Мы просто выражаем общую благодарность им всем.

Также мы хотим сказать теплые слова тем, кто читал рукопись и давал советы. Особенную благодарность мы выражаем Джону Кейси (John Casey), Дэйлу Гриту (Dale Grit) и Франсу Каашоеку (Frans Kaashoek).

Ряд студентов университета Врийе (Амстердам) участвовали в тестировании первых версий компакт-диска. Это были Ахмед Бату (Ahmed Batou), Горан Докик (Goran Dokic), Питер Гийзел (Peter Gijzel), Томер Гил (Tomer Gil), Денис Гримберген (Dennis Grimbergen), Родерик Гресбик (Roderick Groesbeek), Уотер Гэринг (Wouter Haring), Гвидо Коллери (Guido Kollerie), Марк Ласше (Mark Lassche), Раймонд Рис (Raymond Ris), Франс тер Борг (Frans ter Borg), Алекс ван Боллегуи (Alex van Ballegooy), Рис ван дер Вельден (Ries van der Velden), Александр Уэлс (Alexander Wels) и Томас Зееман (Thomas Zeeman). Мы хотим сказать спасибо им всем за тщательную работу и подробные отчеты.

Альберт Вудхилл также хотел бы поблагодарить некоторых своих бывших студентов, особенно Питера В. Янга (Peter W. Yang) из колледжа Хэмпшира, а также Марию Изабель Санчес (Maria Isabel Sanchez) и Вильяма Падди Варгаса (William Puddy Vargas) из Национального университета Никарагуа (Universidad National Autonomia Nicaragua) за то, что их интерес к MINIX поддерживал его усилия.

Наконец, мы были бы ничто без наших семей. Сьюзан (Suzanne) прошла через все это уже десять раз, Барбара (Barbara) — девять, а Марвин (Marvin) — восемь раз. Даже маленький Брэм (Bram) терпел все это уже четырежды. Такую обыденную помощь не всегда ценишь, но я всегда признателен за вашу любовь и поддержку.

Что же до Барбары (Barbara) Альберта, то это ее первое испытание. Без ее помощи, терпения и хорошего чувства юмора ничего этого не было бы возможно. Гордону очень повезло с тем, что все это время он по большей части проводил в колледже. Тем не менее это счастье — иметь сына, которого интересуют и заботят те же вещи, что восторгают меня.

Эндрю С. Таненбаум (Andrew S. Tanenbaum)
Альберт С. Вудхалл (Albert S. Woodhull)

От издательства

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

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу http://www.piter.com/download.

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

Глава 1
Введение

Без программного обеспечения любой компьютер — просто бесполезная груда железа. Именно благодаря программам компьютер может хранить, обрабатывать и искать информацию; отображать мультимедийные документы; вести поиск в Интернете и выполнять множество других важных задач, для которых он и нужен. Программное обеспечение можно грубо разбить на две большие группы: системные программы, управляющие работой самого компьютера, и прикладные программы, выполняющие задачи пользователя. Самая главная системная программа — это **операционная система**, она управляет всеми системными ресурсами и обеспечивает основу, на которой можно писать прикладные программы.

Современная компьютерная система состоит из одного или нескольких процессоров, оперативной памяти (часто называемой RAM — Random Access Memory), дисков, клавиатуры, монитора, принтеров, сетевого интерфейса и других устройств, то есть является сложной комплексной системой. Написание программ, которые следят за всеми компонентами, корректно используют их и при этом работают оптимально, представляет собой крайне трудную задачу. Если бы каждому программисту приходилось задумываться о том, как работают жесткие диски, и помнить десятки различных ситуаций, которые могут случиться при чтении блока данных, то многие программы, скорее всего, вообще не были бы написаны.

Много лет спустя стало совершенно ясно, что нужно как-то оградить программистов от сложности работы с аппаратным обеспечением. Постепенно был выработан следующий путь: поверх аппаратуры помещается дополнительная программная прослойка, которая управляет всем оборудованием и предоставляет пользователю интерфейс или *виртуальную машину*. Операционная система отвечает за управление всеми перечисленными устройствами и обеспечивает пользователя имеющими простой, доступный интерфейс программами для работы с аппаратурой. Эти системы составляют предмет данной книги.

Расположение операционной системы в общей структуре компьютера показано на рис. 1.1. Внизу находится аппаратное обеспечение, которое во многих случаях само состоит из двух или более уровней (или слоев). Самый нижний уровень содержит физические устройства, состоящие из интегральных микросхем, проводников, источников питания, электронно-лучевых трубок и т. п. То, как они устроены и как работают, относится к сфере деятельности инженеров, специалистов по электронике.

Выше (у некоторых машин) расположен *микроархитектурный уровень* — примитивная программная прослойка, напрямую работающая с оборудованием и упрощающая интерфейс для программ более высокого уровня. Эта программа, обычно называемая *микропрограммой*, располагается в ПЗУ. В действительности это просто интерпретатор, который получает машинные команды, такие как

MOVE, JUMP или ADD, и выполняет их в несколько маленьких шагов. Например, чтобы выполнить команду ADD, микропрограмма должна определить, где расположены слагаемые, извлечь их, сложить и куда-то поместить результат. Набор интерпретируемых микропрограммой инструкций определяет *машинный язык*. Этот язык вовсе не является частью аппаратного обеспечения, но производители компьютеров всегда называют его именно так, поэтому многие считают что он действительно «машинный».

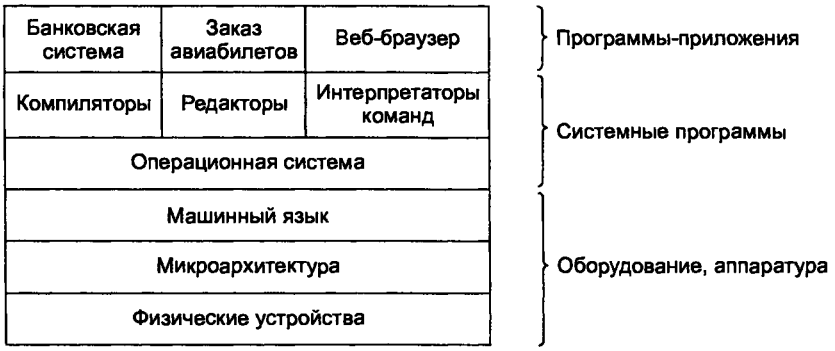


Рис. 1.1. Компьютерная система состоит из аппаратного обеспечения, системных программ и приложений

У некоторых машин микропрограммного уровня нет. Такие системы называются *RISC* (Reduced Instruction Set Computers — компьютеры с упрощенным набором инструкций). В этих машинах инструкции языка выполняются аппаратурой непосредственно. В качестве примеров можно привести Motorola 680x0, у которой есть микропрограммный уровень, и IBM PowerPC, у которого микропрограммы нет.

Обычно машинный язык содержит от 50 до 300 команд, служащих преимущественно для перемещения данных по компьютеру, выполнения арифметических операций и сравнения величин. Управление устройствами на этом уровне осуществляется с помощью загрузки определенных величин в специальные *регистры устройств*. Например, диску можно дать команду чтения, записав в его регистры адрес места на диске, адрес в основной памяти, число байтов для чтения и направление действия (чтение или запись). На практике нужно передавать большее количество параметров, а статус операции, возвращаемый диском, достаточно сложен. Кроме того, при программировании многих устройств ввода/вывода (I/O — Input/Output) очень важную роль играют временные соотношения.

Операционная система предназначена для того, чтобы скрыть от пользователя все эти сложности. Она состоит из уровня программного обеспечения, который частично избавляет от необходимости общения с аппаратурой напрямую, вместо этого предоставляя программисту более удобную систему команд. Действие чтения блока из файла в этом случае представляется намного более простым, чем когда нужно заботиться о перемещении головок диска, ждать, пока они установятся на нужное место и т. д.

Над операционной системой на нашем рисунке расположены остальные системные программы. Здесь находятся интерпретатор команд (оболочка), системы окон, компиляторы, редакторы и т. д. Важно понимать, что подобные программы не являются частью операционной системы, хотя обычно поставщики компьютеров устанавливают их на машины. Это очень важное замечание. Под операционной системой обычно понимается то программное обеспечение, которое запускается в *режиме ядра* или, как его еще называют, *режиме супервизора*. Она защищена от вмешательства пользователя с помощью аппаратных средств (мы не рассматриваем в данный момент некоторые старые микропроцессоры, которые вообще не имеют аппаратной защиты). Компиляторы и редакторы запускаются в *пользовательском режиме*. Если пользователю не нравится какой-либо компилятор, он при желании может написать свой собственный, но он не может написать собственный обработчик прерываний системных часов, являющийся частью операционной системы и обычно защищенный аппаратно от попыток его модифицировать.

Наконец, над системными программами расположены прикладные программы. Обычно они покупаются или пишутся пользователем для решения собственных проблем — обработки текста, электронных таблиц, технических расчетов или игр.

1.1. Что такое операционная система?

Большинство пользователей компьютеров имеют некоторый опыт общения с операционной системой, но обычно они испытывают затруднения при попытке дать определение операционной системы. В известной степени проблема связана с тем, что операционные системы выполняют две основные, но практически не связанные между собой функции: расширение возможностей машины и управление ее ресурсами. И в зависимости от того, какому пользователю вы зададите вопрос, вы услышите в ответ больше или об одной функции, или о другой. Давайте рассмотрим обе функции.

1.1.1. Операционная система как расширенная машина

Как было упомянуто ранее, *архитектура* (система команд, организация памяти, ввод/вывод данных и структура шин) большинства компьютеров на уровне машинного языка примитивна и неудобна для работы с программами, особенно в процессе ввода/вывода данных. Чтобы это утверждение не показалось голословным, кратко рассмотрим пример того, как происходит ввод/вывод данных с гибкого диска через совместимые микросхемы контроллера NEC PD765, используемые на большинстве персональных компьютеров с процессором Intel. (В этой книге мы будем использовать и термин «гибкий диск», и термин «дискета».) Контроллер PD765 имеет 16 команд, каждая задается передачей от 1 до 9 байтов в регистр устройства. Это команды для чтения и записи данных, перемещения

головки диска и форматирования дорожек, а также для инициализации, распознавания, установки в исходное положение и калибровки контроллера и приводов.

Основными командами являются команды `read` и `write` (чтение и запись). Каждая из них требует 13 параметров, упакованных в 9 байт. Эти параметры определяют такие элементы, как адрес блока на диске, который нужно прочитать, количество секторов на дорожке, физический режим записи, расстановку промежутков между секторами. Они же сообщают, что делать с меткой адреса данных, которые были удалены. Если вы не можете сразу это осмыслить, не волнуйтесь — полностью это понятно лишь посвященным. Когда выполнение операции завершается, чип контроллера возвращает упакованные в 7 байт 23 параметра, отражающие наличие и типы ошибок. Но этого не достаточно, и программист при работе с гибким диском должен также постоянно знать, включен двигатель или нет. Если двигатель выключен, его следует включить (с длительным ожиданием запуска) прежде, чем данные будут прочитаны или записаны. Двигатель не может оставаться включенным слишком долго, так как гибкий диск изнашивается. Программист вынужден выбирать между длинными задержками во время загрузки и изнашивающимися гибкими дисками (с вероятностью потери данных на них).

Даже если не вдаваться глубже в подробности этого процесса, становится ясно, что обыкновенный программист вряд ли захочет столкнуться с такими деталями при работе с гибким диском (или жестким диском, работа с ним не менее сложна). Вместо этого программисту нужны простые высокоуровневые абстракции. В случае работы с дисками типичной абстракцией является коллекция именованных файлов, содержащихся на диске. Каждый файл может быть открыт для чтения или записи, прочитан или записан, а потом закрыт. А такие детали, как текущее состояние двигателя или использование при записи модифицированной частотной модуляции, не должны содержаться в абстракции, предстающей перед пользователем.

Программа, скрывающая истину об аппаратном обеспечении и представляющая простой список поименованных файлов, которые можно читать и записывать, является операционной системой. Операционная система не только устраняет необходимость работы непосредственно с дисками и предоставляет простой, ориентированный на работу с файлами интерфейс, но и скрывает множество неприятной работы с прерываниями, счетчиками времени, организацией памяти и другими элементами низкого уровня. В каждом случае абстракция, предлагаемая операционной системой, намного проще и удобнее в обращении, чем то, что может предложить нам непосредственно основное оборудование.

С точки зрения пользователя операционная система выполняет функцию расширенной машины или виртуальной машины, в которой проще программировать и легче работать, чем непосредственно с аппаратным обеспечением, составляющим реальный компьютер. То, каким образом операционная система достигает своей цели — долгая история, но мы подробно рассмотрим этот процесс в нашей книге. Подведем итог вышесказанному: операционная система предоставляет нам ряд возможностей, которые могут использовать программы с по-

мощью специальных команд, называемых системными вызовами. Мы приведем примеры наиболее общих системных вызовов далее в этой главе.

1.1.2. Операционная система как менеджер ресурсов

Концепция, рассматривающая операционную систему, прежде всего, как удобный интерфейс пользователя — это взгляд сверху вниз. Альтернативный взгляд, снизу вверх, дает представление об операционной системе как о механизме, присутствующем в устройстве компьютера для управления всеми частями этой сложнейшей машины. Современные компьютеры состоят из процессоров, памяти, датчиков времени, дисков, мыши, сетевого интерфейса, принтеров и огромного количества других устройств. В соответствии со вторым подходом работа операционной системы заключается в обеспечении организованного и контролируемого распределения процессоров, памяти и устройств ввода/вывода между различными программами, состязающимися за право их использовать.

Представьте, что случилось бы, если бы на одном компьютере оказались работающими три программы, и все они одновременно попытались бы напечатать свои выходные данные на одном и том же принтере. Возможно, первые несколько строк на листе появились бы от первой программы, следующие несколько — из второй программы, затем бы следовало несколько строк от третьей программы и т. д. В результате получилась бы полная неразбериха. Операционная система наводит порядок в подобных ситуациях, буферизируя на диске все данные, предназначенные для печати. В процессе работы программы операционная система сохраняет ее выходные данные на диске во временном файле. Затем, по окончании работы этой программы, система отправляет данные на принтер, в то время как другая программа может продолжать формировать свои выходные данные, не обращая внимания на то, что они пока еще фактически не посылаются на печатающее устройство.

Когда компьютером (или сетью) пользуются несколько пользователей, необходимость в управлении памятью, устройствами ввода/вывода, другими ресурсами и их защите сильно возрастает, поскольку пользователи могут обращаться к ним в абсолютно непредсказуемом порядке. К тому же часто приходится распределять между пользователями не только оборудование, но и информацию (файлы, базы данных и т. д.). С этой точки зрения основная задача операционной системы заключается в отслеживании того, кто и какой ресурс использует, в обработке запросов на ресурсы, в подсчете коэффициента загрузки и разрешении проблем конфликтующих запросов от различных программ и пользователей.

1.2. История операционных систем

История развития операционных систем насчитывает уже много лет. В следующих разделах книги мы кратко рассмотрим некоторые основные моменты этого

развития. Так как операционные системы появились и развивались в процессе конструирования компьютеров, то эти события исторически тесно связаны. Поэтому чтобы представить, как выглядели операционные системы, мы обсудим следующие друг за другом поколения компьютеров. Такая схема взаимосвязи поколений операционных систем и компьютеров довольно груба, но она обеспечивает некоторую структуру, без которой ничего не было бы понятно.

Первый настоящий цифровой компьютер был изобретен английским математиком Чарльзом Бэббиджем (Charles Babbage, 1792–1871). Хотя большую часть жизни Бэббидж посвятил попыткам создания своей «аналитической машины», он так и не смог заставить ее работать должным образом. Это была чисто механическая машина, а технологии того времени не были достаточно развиты для изготовления многих деталей и механизмов высокой точности. Не стоит и говорить, что его аналитическая машина не имела операционной системы.

Интересный исторический факт: Бэббидж понимал, что для аналитической машины ему необходимо программное обеспечение, поэтому он нанял молодую женщину по имени Ада Лавлейс (Ada Lovelace), дочь знаменитого британского поэта Лорда Байрона. Она и стала первым в мире программистом, а язык программирования Ada® назван в ее честь.

1.2.1. Первое поколение (1945–1955): электронные лампы и коммутационные панели

После неудачных попыток Бэббиджа вплоть до Второй мировой войны в конструировании цифровых компьютеров не было практически никакого прогресса. Примерно в середине 1940-х Говард Айкен (Howard Aiken) в Гарварде, Джон фон Нейман (John von Neumann) в Институте углубленного изучения в Принстоне, Дж. Преспер Эккерт (J. Presper Eckert), Вильям Мочли (William Mauchley) в Пенсильванском университете, Конрад Цузе (Konrad Zuse) в Германии и многие другие продолжили работу в направлении создания вычислительных машин. На первых машинах использовались механические реле, но они были очень медлительны, длительность такта составляла несколько секунд. Позже реле заменили электронными лампами. Машины получались громоздкими, заполняющими целые комнаты, с десятками тысяч электронных ламп, но все равно они были в миллионы раз медленнее, чем даже самый дешевый современный персональный компьютер.

В те времена каждую машину и разрабатывала, и строила, и программировала, и эксплуатировала, и поддерживала в рабочем состоянии одна команда. Все программирование выполнялось на абсолютном машинном языке, управление основными функциями машины осуществлялось просто при помощи соединения коммутационных панелей проводами. Тогда еще не были известны языки программирования (даже ассемблера не было). Об операционных системах никто и не слышал. Обычный режим работы программиста был таков: записаться на определенное время на специальном стенде, затем спуститься в машинную комнату, вставить свою коммутационную панель в компьютер и провести несколько следующих часов в надежде, что во время работы ни одна из двадцати

тысяч электронных ламп не выйдет из строя. Фактически тогда на компьютерах занимались только прямыми числовыми вычислениями, например расчетами таблиц синусов, косинусов и логарифмов.

К началу 50-х, с выпуском перфокарт, установившееся положение несколько улучшилось. Стало возможно вместо использования коммутационных панелей записывать и считывать программы с карт, но во всем остальном процедура вычислений оставалась прежней.

1.2.2. Второе поколение (1955–1965): транзисторы и системы пакетной обработки

В середине 50-х изобретение и применение транзисторов радикально изменило всю картину. Компьютеры стали достаточно надежными, появилась высокая вероятность того, что машина будет работать довольно долго, выполняя при этом полезные функции. Впервые сложилось четкое разделение между проектировщиками, сборщиками, операторами, программистами и обслуживающим персоналом.

Машины, теперь называемые мэйнфреймами, располагались в специальных комнатах с кондиционированным воздухом, где ими управлял целый штат профессиональных операторов. Только большие корпорации, правительственные учреждения или университеты могли позволить себе технику, цена которой исчислялась миллионами долларов. Чтобы выполнить задание (то есть программу или комплект программ), программист сначала должен был записать его на бумаге (на FORTRAN или ассемблере), а затем перенести на перфокарты. После этого — принести колоду перфокарт в комнату ввода данных, передать одному из операторов и идти пить кофе в ожидании, когда будет готов результат.

Когда компьютер заканчивал выполнение какого-либо из текущих заданий, оператор подходил к принтеру, отрывал лист с полученными данными и относил его в комнату для распечаток, где программист позже мог его забрать. Затем оператор брал одну из колод перфокарт, принесенных из комнаты ввода данных, и считывал их. Если в процессе расчетов был необходим компилятор языка FORTRAN, то оператору приходилось брать его из картотечного шкафа и загружать в машину отдельно. Из-за одного только хождения операторов по машинному залу впустую терялась масса драгоценного компьютерного времени.

Если учитывать высокую стоимость оборудования, не удивительно, что люди довольно скоро занялись поиском способа повышения эффективности использования машинного времени. Общепринятым решением стала *система пакетной обработки*. Первоначально замысел состоял в том, чтобы собрать полный поднос заданий (колод перфокарт) в комнате входных данных и затем переписать их на магнитную ленту, используя небольшой и (относительно) недорогой компьютер, например IBM 1401, который был очень хорош для считывания карт, копирования лент и печати выходных данных, но не подходил для числовых вычислений.

Другие, более дорогостоящие машины, такие как IBM 7094, использовались для настоящих вычислений. Это изображено на рис. 1.2.

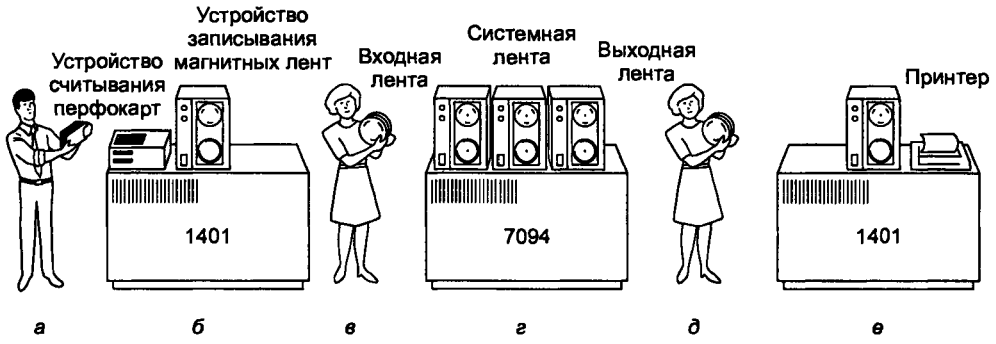


Рис. 1.2. Ранняя система пакетной обработки: *а* — программист приносит карты для IBM 1401; *б* — IBM 1401 записывает пакет заданий на магнитную ленту; *в* — оператор приносит входные данные на ленте к IBM 7094; *г* — IBM 7094 выполняет вычисления; *д* — оператор переносит ленту с выходными данными на IBM 1401; *е* — IBM 1401 печатает выходные данные

Примерно после часа сбора пакета заданий лента перематывалась и ее относили в машинную комнату, где устанавливали на лентопротяжном устройстве. Затем оператор загружал специальную программу (прообраз сегодняшней операционной системы), которая считывала первое задание с ленты и запускала его. Выходные данные записывались на вторую ленту вместо того, чтобы идти на печать. Завершив очередное задание, операционная система автоматически считывала с ленты следующее и начинала обрабатывать его. После обработки всего пакета оператор снимал ленты с входной и выходной информацией, ставил новую ленту со следующим заданием, а готовые данные помещал на IBM 1401 для печати в *автономном режиме* (то есть без связи с главным компьютером).

Структура типичного входного задания показана на рис. 1.3. Оно начиналось с карты $ЗАДАНИЕ, на которой указывалось максимальное время выполнения задания в минутах, загружаемый учетный номер и имя программиста. Затем поступала карта $FORTRAN, дающая операционной системе указание загрузить компилятор языка FORTRAN с системной магнитной ленты. Эта карта следовала за программой, которую нужно было компилировать, а после нее шла карта $ЗАГРУЗИТЬ, указывающая операционной системе загрузить только что скомпилированную объектную программу. (Скомпилированные программы часто записывались на временных лентах, данные с которых могли стираться сразу после использования, и их загрузка должна была выполняться явно.) Следом шла карта $ЗАПУСТИТЬ с данными, дающая операционной системе команду выполнять программу. Наконец, карта завершения $КОНЕЦ отмечала конец задания. Эти примитивные управляющие перфокарты были предшественниками современных языков управления и интерпретаторов команд.

Большие компьютеры второго поколения использовались главным образом для научных и технических вычислений, таких как решение дифференциальных уравнений в частных производных, часто встречающихся в физике и инженерных задачах. В основном на них программировали на языке FORTRAN и ассемблере, а типичными операционными системами были FMS (Fortran Monitor

System) и IBSYS (операционная система, созданная корпорацией IBM для компьютера IBM 7094).

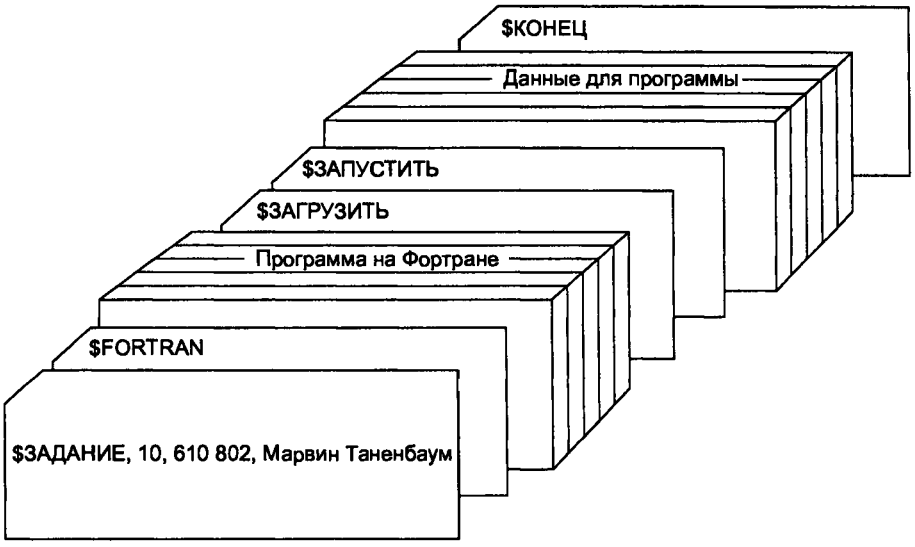


Рис. 1.3. Структура типичного задания FMS

1.2.3. Третье поколение (1965–1980): интегральные схемы и многозадачность

К началу 60-х годов большинство изготовителей компьютеров имело две отдельные, полностью несовместимые производственные линии. С одной стороны, существовали научные крупномасштабные компьютеры с пословной обработкой текста типа IBM 7094, использовавшиеся для числовых вычислений в науке и технике. С другой стороны — коммерческие компьютеры с посимвольной обработкой, такие как IBM 1401, широко используемые банками и страховыми компаниями для сортировки и печати данных.

Развитие и поддержка двух совершенно разных производственных линий для изготовителей были достаточно дорогим удовольствием. Кроме того, многим покупателям изначально требовалась небольшая машина, однако позже ее возможностей становилось недостаточно, и требовался более мощный компьютер, который работал бы с теми же самыми программами, но быстрее.

Фирма IBM попыталась решить эти проблемы разом, выпустив серию машин IBM/360. 360-е были серией программно совместимых машин, варьирующихся от компьютеров размером с IBM 1401 до машин, значительно более мощных, чем IBM 7094. Эти компьютеры различались только ценой и производительностью (максимальным объемом памяти, быстродействием процессора, количеством разрешенных устройств ввода/вывода и т. д.). Так как все машины имели одинаковую структуру и набор команд, программы, написанные для одного ком-

пьютера, могли работать на всех других (по крайней мере, в теории). Кроме того, 360-е были разработаны для поддержки как научных (то есть численных), так и коммерческих вычислений. Одно семейство машин могло удовлетворить нужды всех покупателей. В последующие годы, используя более современные технологии, корпорация IBM выпустила компьютеры, совместимые с 360, эти серии известны под номерами 370, 4300, 3080 и 3090.

360-е стали первой основной линией компьютеров, на которой использовались мелкомасштабные интегральные схемы, дававшие преимущество в цене и качестве по сравнению с машинами второго поколения, созданными из отдельных транзисторов. Корпорация IBM добилась мгновенного успеха, а идею семейства совместимых компьютеров скоро приняли и все остальные основные производители. В компьютерных центрах до сих пор можно встретить потомков этих машин. В настоящее время они часто используются для управления огромными базами данных (например, для систем бронирования и продажи билетов на авиалиниях) или как серверы узлов Интернета, которые должны обрабатывать тысячи запросов в секунду.

Основное преимущество «одного семейства» оказалось одновременно и величайшей его слабостью. По замыслу его создателей все программное обеспечение, включая операционную систему OS/360, должно было одинаково хорошо работать на всех моделях компьютеров: и в небольших системах, которые часто заменяли 1401-е и применялись для копирования перфокарт на магнитные ленты, и на огромных системах, заменяющих 7094-е и использовавшихся для расчета прогноза погоды и других сложных вычислений. Кроме того, предполагалось, что одну операционную систему можно будет использовать в системах как с несколькими внешними устройствами, так и с большим их количеством; а также как в коммерческих, так и в научных областях. Но самым важным было, чтобы это семейство машин давало результаты независимо от того, кто и как его использует.

Ни IBM, ни кто-либо еще не мог написать программного обеспечения, удовлетворяющего всем этим противоречивым требованиям. В результате появилась огромная и необычайно сложная операционная система, примерно на два или три порядка превышающая по величине FMS. Она состояла из миллионов строк, написанных на ассемблере тысячами программистов, содержала тысячи и тысячи ошибок, что повлекло за собой непрерывный поток новых версий, в которых пытались исправить эти ошибки. В каждой новой версии устранялась только часть ошибок, вместо них появлялись новые, так что общее их число, вероятно, оставалось постоянным.

Один из разработчиков OS/360, Фред Брукс (Fred Brooks), впоследствии написал остроумную и язвительную книгу с описанием своего опыта работы с OS/360. Мы не можем здесь дать полную оценку этой книги, но достаточно будет сказать, что на ее обложке изображено стадо доисторических животных, увязших в яме с дегтем. Обложка книги [71] демонстрирует похожую точку зрения на операционные системы, бывшие динозаврами в мире компьютеров.

Несмотря на свои огромные размеры и недостатки, OS/360 и подобные ей операционные системы третьего поколения, созданные другими производителя-

ми компьютеров, на самом деле достаточно неплохо удовлетворяли требованиям большинства клиентов. Они даже сделали популярными несколько ключевых технических приемов, отсутствовавших в операционных системах второго поколения. Самым важным достижением явилась многозадачность. На компьютере IBM 7094, когда текущая работа приостанавливалась в ожидании операций ввода/вывода с магнитной ленты или других устройств, центральный процессор просто бездействовал до окончания операции ввода/вывода. При сложных научных вычислениях и ограниченных возможностях процессора устройства ввода/вывода задействовались довольно редко, так что это потраченное впустую время не играло существенной роли. Но при коммерческой обработке данных время ожидания устройства ввода/вывода могло занимать 80 или 90 % всего рабочего времени, поэтому необходимо было что-нибудь сделать во избежание длительного простаивания весьма дорогостоящего процессора.

Решение этой проблемы заключалось в разбиении памяти на несколько частей, называемых разделами, каждому из которых давалось отдельное задание, как показано на рис. 1.4. Пока одно задание ожидало завершения работы устройства ввода/вывода, другое могло использовать центральный процессор. Если в оперативной памяти содержалось достаточное количество заданий, центральный процессор мог быть загружен почти на все 100 % по времени. Множество одновременно хранящихся в памяти заданий требовало наличия специального оборудования для защиты каждого задания от возможного любопытства и ущерба со стороны остальных заданий. 360-я и другие системы третьего поколения были снабжены подобными аппаратными средствами.

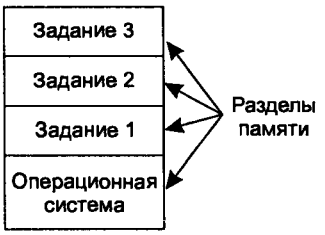


Рис. 1.4. Многозадачная система с тремя заданиями в памяти

Другим важным плюсом операционных систем третьего поколения стала способность считывать задание с перфокарт на диск по мере того, как их приносили в машинный зал. Всякий раз, когда текущее задание заканчивалось, операционная система могла загружать новое задание с диска в освободившийся раздел памяти и запускать его. Этот технический прием называется *подкачкой* данных или спулингом (spooling, это английское слово произошло от аббревиатуры Simultaneous Peripheral Operation On Line — совместная периферийная операция в интерактивном режиме), и его также используют для выдачи полученных данных. С появлением подкачки стали больше не нужны 1401-е и исчезли многократные переносы магнитных лент.

Хотя операционные системы третьего поколения вполне подходили для больших научных вычислений и справлялись с крупными коммерческими обработками данных, они все еще, по существу, представляли собой разновидности системы пакетной обработки. Многие программисты тосковали по первому поколению машин, когда они могли распоряжаться всей машиной в течение нескольких часов и имели возможность быстро отлаживать свои программы. При системах третьего поколения временной промежуток между передачей задания и возвращением результатов часто составлял несколько часов, так что единственная неуместная запятая могла стать причиной сбоя при компиляции, и получалось, что программист тратил впустую половину дня.

Желание сократить время ожидания ответа привело к разработке *режима разделения времени*, варианту многозадачности, при котором у каждого пользователя есть свой диалоговый терминал. Если двадцать пользователей зарегистрированы в системе, работающей в режиме разделения времени, и семнадцать из них думают, беседуют или пьют кофе, то центральный процессор по очереди предоставляется трем пользователям, желающим работать на машине. Так как люди, отлаживая программы, обычно выдают короткие команды (например, компилировать процедуру на пяти страницах[1]) чаще, чем длинные (например, упорядочить файл с миллионами записей), то компьютер может обеспечивать быстрое интерактивное обслуживание нескольких пользователей. При этом он может работать над большими пакетами в фоновом режиме, когда центральный процессор не занят другими заданиями. Первая серьезная система с режимом разделения времени **CTSS** (Compatible Time Sharing System — совместимая система разделения времени) была разработана в Массачусетсском технологическом институте (M.I.T.) на специально переделанном компьютере IBM 7094 [14]. Однако режим разделения времени не стал действительно популярным до тех пор, пока не получили широкого распространения необходимые технические средства защиты.

После успеха системы CTSS Массачусетсский технологический институт, система исследовательских лабораторий Bell Labs и корпорация General Electric (тогда главный изготовитель компьютеров) решили начать разработку «компьютерного предприятия общественного пользования» — машины, которая должна была поддерживать сотни одновременных пользователей в режиме разделения времени. Образцом для новой машины послужила система распределения электроэнергии. Когда вам нужна электроэнергия, вы просто вставляете штепсель в розетку и получаете энергии столько, сколько вам нужно. Проектировщики этой системы, известной как *MULTICS* (MULTiplexed Information and Computing Service — мультиплексная информационная и вычислительная служба), представляли себе одну огромную вычислительную машину, воспользоваться услугой которой мог каждый человек в районе Бостона. Мысль о том, что машины, гораздо более мощные, чем их мэйнфрейм GE-645, будут продаваться миллионами по цене тысяча долларов за штуку всего лишь через тридцать лет, казалась

[1] В этой книге мы будем использовать термины «процедура», «подпрограмма» и «функция» в одном значении.

чистейшей научной фантастикой, как если бы сегодня кто-либо вздумал проектировать сверхзвуковые трансатлантические подводные поезда.

В итоге, MULTICS подала много конструктивных идей компьютерным теоретикам, но превратить ее в серьезный продукт и добиться коммерческого успеха оказалось намного тяжелее, чем ожидалось. Система исследовательских лабораторий Bell Labs выбыла из проекта, а компания General Electric совсем оставила компьютерный бизнес. Однако Массачусетсский технологический институт проявил упорство и со временем получил работающую систему. В конце концов, она была продана как коммерческое изделие компанией Honeywell, купившей компьютерный бизнес General Electric, и установлена примерно в восьмидесяти больших компаниях и университетах по всему миру. Но к настоящему времени в связи с падением цен на компьютерное оборудование идея компьютерного предприятия общественного пользования выдохлась. Несмотря на неудачу с точки зрения коммерции, система MULTICS значительно повлияла на последующие операционные системы. Это описано в книгах [15, 16, 18, 63, 66].

Еще одним важным моментом развития во времена третьего поколения был феноменальный рост мини-компьютеров, начиная с выпуска машины PDP-1 корпорацией DEC в 1961 году. Компьютеры PDP-1 обладали оперативной памятью, состоящей всего лишь из 4 К 18-битовых слов, но стоили они по 120 тысяч долларов за штуку (это меньше 5 % от цены IBM 7094) и поэтому расхватывались как горячие пирожки. На некоторых видах нечисловой работы они работали почти с такой же скоростью, как IBM 7094, что дало толчок к появлению новой индустрии. За этой машиной последовала целая серия других PDP (в отличие от семейства IBM, полностью несовместимых), и как кульминация — PDP-11.

Кен Томпсон (Ken Thompson), один из специалистов по компьютерам в Bell Labs, работавший над проектом MULTICS, впоследствии нашел мини-компьютер PDP-7, которым никто не пользовался, и решил написать усеченную однопользовательскую версию системы MULTICS. Эта работа позже развилась в операционную систему *UNIX*, ставшую популярной в академическом мире, в правительственных управлениях и во многих компаниях.

История развития UNIX уже многократно рассказывалась в самых различных книгах (например, [68]). Часть ее будет представлена в главе 10. Пока достаточно сказать, что по причине широкой доступности исходного кода различные организации создали свои собственные (несовместимые) версии, что привело к хаосу. Были разработаны две главные версии: *System V* корпорации AT&T и *BSD* (Berkeley Software Distribution) Калифорнийского университета Беркли. Эти системы, в свою очередь, распадаются на отдельные разновидности. Чтобы стало возможным писать программы, работающие в любой UNIX-системе, Институт инженеров по электротехнике и электронике IEEE разработал стандарт системы UNIX, называемый *POSIX*, который теперь поддерживают большинство версий UNIX. Стандарт POSIX определяет минимальный интерфейс системного вызова, который должны поддерживать совместимые системы UNIX. Некоторые другие операционные системы теперь тоже поддерживают интерфейс POSIX.

1.2.4. Четвертое поколение (с 1980 года по наши дни): персональные компьютеры

Следующий период в эволюции операционных систем связан с появлением больших интегральных схем (LSI, Large Scale Integration) — кремниевых микросхем, содержащих тысячи транзисторов на одном квадратном сантиметре. С точки зрения архитектуры персональные компьютеры (первоначально называемые *микрокомпьютерами*) были во многом похожи на мини-компьютеры класса PDP-11, но, конечно, отличались по цене. Если появление мини-компьютеров позволило отделам компаний и факультетам университетов иметь собственный компьютер, то с появлением микропроцессоров каждый человек получил возможность купить свой собственный персональный компьютер. Мощные персональные компьютеры, используемые в бизнесе, университетах и правительственных учреждениях, называют рабочими станциями, но в действительности это просто большие персональные компьютеры. Обычно такие компьютеры соединяются сетью.

Широкая доступность вычислительных мощностей, особенно интерактивных систем, обладающих неплохой графикой, привело к росту производства программного обеспечения. Большинство производимых программ имели дружественный интерфейс, то есть рассчитывались не просто на пользователей, ничего не знающих о программе, но на пользователей, не испытывающих желания ее изучать. Это сильно отличалось от OS/360, где язык управления задачами (JCL, Job Control Language) был настолько запутан, что о нем писались целые книги.

Изначально на сцене персональных компьютеров и рабочих станций доминировали две операционные системы: MS-DOS от Microsoft и UNIX. MS-DOS широко использовалась на персональных компьютерах Intel и других системах, основанных на процессоре Intel 8088 и его наследниках: 80286, 80386 и 80486 (которые далее будут обозначаться просто как 286, 386 и 486 соответственно), а также Pentium и Pentium Pro. Первые версии MS-DOS были относительно примитивны, но со временем в систему вошли многие другие возможности, в том числе и многие из UNIX. Как следующая ступень после MS-DOS, Microsoft предлагается ОС WINDOWS. Сначала она работала поверх MS-DOS (то есть была скорее оболочкой, чем операционной системой), но в 1995 году была выпущена самостоятельная версия, Windows 95®, которой уже не требовалась MS-DOS. Другая операционная система от Microsoft, Windows NT, поддерживает некоторую совместимость с Windows 95, но на внутреннем уровне она полностью переписана.

Другой основной соперник — UNIX. Эта операционная система доминирует на рабочих станциях и других мощных компьютерах, таких как сетевые серверы. Особенно популярна эта ОС на высокопроизводительных системах на RISC процессорах. Вычислительная мощность таких машин обычно соответствует мощности мини-компьютера, хотя они и предназначены для одного пользователя. Поэтому вполне логично применять на таких машинах ОС для мини-компьютеров, а именно UNIX.

С середины 80-х годов начали расти и развиваться сети персональных компьютеров, управляемых *сетевыми* и *распределенными операционными системами* [80]. В сетевой операционной системе пользователи знают о существовании многочисленных компьютеров, могут регистрироваться на удаленных машинах и копировать файлы с одной машины на другую. Каждый компьютер работает под управлением локальной операционной системы и имеет своего собственного локального пользователя (или пользователей).

Сетевые операционные системы несущественно отличаются от однопроцессорных операционных систем. Ясно, что они нуждаются в сетевом интерфейсном контроллере и специальном низкоуровневом программном обеспечении, поддерживающем работу контроллера, а также в программах, разрешающих пользователям удаленную регистрацию в системе и доступ к удаленным файлам. Но эти дополнения по сути не изменяют структуры операционной системы.

Распределенная операционная система, напротив, представляется пользователям традиционной однопроцессорной системой, хотя она и составлена из множества процессоров. При этом пользователи не должны беспокоиться о том, где работают их программы или где расположены файлы; все это должно автоматически и эффективно обрабатываться самой операционной системой.

Чтобы создать настоящую распределенную операционную систему, недостаточно просто добавить несколько страниц кода к однопроцессорной операционной системе, так как распределенные и централизованные системы имеют существенные различия. Распределенные системы, например, часто позволяют прикладным задачам одновременно обрабатываться на нескольких процессорах, поэтому требуется более сложный алгоритм загрузки процессоров для оптимизации распараллеливания.

Наличие задержек при передаче данных в сетях означает, что эти алгоритмы должны работать с неполной, устаревшей или даже неправильной информацией. Эта ситуация радикально отличается от однопроцессорной системы, в которой операционная система обладает полной информацией относительно состояния системы.

1.2.5. История MINIX

Во времена молодости UNIX (Version 6), ее исходные коды были широко доступны по лицензии AT&T и часто изучались. Джон Лайонс (John Lions) из университета Нового Южного Уэльса в Австралии даже написал небольшую брошюру, описывающую шаг за шагом работу UNIX. С разрешения AT&T эта брошюра использовалась во многих университетских курсах по операционным системам.

С выходом UNIX Version 7 стало ясно, что UNIX превратилась в дорогостоящий коммерческий продукт, поэтому лицензия, под которой распространялась Version 7, запрещала изучение исходного кода на учебных курсах, чтобы не подвергать риску его статус коммерческого секрета. Поэтому многие университеты просто прекратили изучение UNIX, довольствуясь одной теорией.

К сожалению, изучение одной только теории формирует у студентов однобокий взгляд на то, какой в действительности может быть операционная система. В книгах и курсах, посвященных операционным системам, в огромных подробностях рассматриваются такие теоретические главы, как, например, алгоритмы планирования, которые на практике не столь важны. Действительно важные вещи, такие как ввод/вывод и файловые системы, зачастую опускаются, так как им не посвящено достаточно теории.

Чтобы исправить ситуацию, один из авторов этой книги (Э. Таненбаум) решил написать собственную операционную систему, которая с точки зрения пользователя была бы совместима с UNIX, но внутри была бы совершенно самостоятельной. Так как в этой системе не используется ни строчки кода AT&T, она не попадает под действие лицензионных ограничений и может свободно использоваться при обучении. Таким образом, студенты могут «вскрывать» реальную операционную систему, чтобы увидеть, как она устроена изнутри, точно так же, как студенты-медики вскрывают лягушек. Название MINIX происходит от mini-UNIX, так как эта система достаточно мала, чтобы даже не-гуру мог понять, как она работает.

У MINIX есть и еще одно преимущество перед UNIX. Она на десять лет моложе UNIX, поэтому ее код обладает более модульной структурой. Например, файловая система MINIX вообще не является частью ядра, а работает как отдельная пользовательская программа. Другое отличие в том, что UNIX создавалась, чтобы быть эффективной. MINIX же создавалась, чтобы быть читаемой (насколько может быть читаемым текст любой программы на сотни страниц). Поэтому, например, в коде MINIX имеются тысячи комментариев.

ОС MINIX разрабатывалась в расчете на совместимость с UNIX Version 7. Эта версия была выбрана за основу благодаря ее простоте и элегантности. Иногда говорят, что Version 7 была улучшением не только по сравнению с предыдущими версиями, но и по сравнению с последующими. С пришествием POSIX, развитие MINIX начало стремиться к новому стандарту, поддерживая в то же время обратную совместимость с существующими программами. Это обычный для компьютерной индустрии путь развития, так как никакой производитель не захочет представлять систему, которой никто не сможет пользоваться. Рассматриваемая в этой книге версия MINIX базируется на стандарте POSIX (в отличие от версии, рассматриваемой в первом издании, которая базировалась на V7).

Как и UNIX, MINIX написана на языке программирования C, чтобы упростить ее перенос на различные компьютеры. Первая реализация предназначалась для IBM PC, так как эти компьютеры были широко распространены. Затем система была перенесена на Atari, Amiga, Macintosh и SPARC. Придерживаясь философии «маленькое есть прекрасное», MINIX изначально не требовала для работы жесткий диск, тем самым вписываясь в студенческий бюджет (сейчас это может показаться удивительным, но в середине 80-х, когда MINIX впервые увидела свет, жесткие диски все еще были дорогой новинкой). Со временем и функциональность, и объем системы росли, и в итоге потребовался жесткий диск. Но философия MINIX не была забыта, и для работы вполне достаточно раздела объемом 30 мегабайт. В противоположность этому сейчас для коммерческих UNIX 200 мегабайт считается абсолютным минимумом.

Для среднего пользователя, сидящего за IBM PC, MINIX мало отличается от UNIX. Имеются стандартные программы, такие как cat, grep, ls, make, выполняющие те же действия, что и их аналоги в UNIX. Как и сама операционная система, эти программы были полностью переписаны автором, студентами и некоторыми другими посвященными людьми.

По всей этой книге в качестве примера будет использоваться MINIX, но большинство комментариев (конечно, кроме тех, которые ссылаются на сами исходные коды) применимы и к UNIX. Многие применимы и к другим операционным системам. Это замечание нужно учитывать, читая книгу.

Отступая от темы, можно сказать несколько слов о LINUX и связи LINUX с MINIX. Вскоре после создания MINIX, для обсуждения этой операционной системы была сформирована группа новостей USENET. За несколько недель на нее подписалось более сорока тысяч подписчиков, и большинство из них хотели добавить в систему множество новых возможностей, чтобы сделать ее лучше и больше (ну, или хотя бы просто больше). Каждый день несколько сотен человек давали советы, предлагали идеи и куски кода. Создатель системы несколько лет успешно сопротивлялся этому напору, чтобы система оставалась достаточно маленькой и понятной для студентов. Постепенно стало понятно, что автор будет придерживаться этой стратегии. Именно в этот момент финский студент, Линус Торвальдс, решил создать собственный клон MINIX, который должен был стать рабочей системой со множеством возможностей. А не учебным пособием для студентов. Это и было рождение LINUX.

1.3. Понятия операционной системы

Интерфейс между операционной системой и пользовательскими программами определяется набором «расширенных инструкций», предоставляемых системой. По традиции эти расширенные инструкции называют *системными вызовами*, хотя сейчас для их реализации используется несколько разных способов. Чтобы действительно понять, что может делать операционная система, нужно тщательно изучить этот интерфейс. Имеющиеся вызовы могут значительно отличаться у разных операционных систем (хотя скрывающиеся за ними концепции оказываются общими).

Тем самым нам приходится делать выбор между размытыми обобщениями («у операционных систем есть системные вызовы для чтения файлов») и спецификой конкретной системы («у MINIX имеется системный вызов READ с тремя параметрами: один указывает, какой файл будет считываться, другой указывает, куда поместить считанные данные, и третий задает количество байт, которые будут считаны»).

Мы выбрали второй подход. Он сложнее, но дает больше понимания того, как работают операционные системы. В разделе 1.4 дан более подробный обзор системных вызовов, имеющихся в MINIX и UNIX. Для простоты мы будем рассматривать только MINIX, но в большинстве случаев соответствующие вызовы UNIX основаны на стандарте POSIX. Тем не менее перед рассмотрением реальных сис-

темных вызовов имеет смысл окинуть взглядом MINIX с высоты птичьего полета, чтобы почувствовать, что это за система. Этот обзор в равной степени применим и к UNIX.

Системные вызовы MINIX можно грубо разделить на две категории: вызовы для работы с процессами и вызовы для работы с файловой системой. Рассмотрим каждую из этих групп.

1.3.1. Процессы

Ключевое понятие MINIX и любой другой операционной системы — *процесс*. Процессом, по существу, называют программу в момент выполнения. С каждым процессом связывается его *адресное пространство* — список адресов в памяти от некоторого минимума (обычно нуля) до некоторого максимума, которые процесс может прочесть и в которые он может писать. Адресное пространство содержит саму программу, данные к ней и ее стек. Со всяким процессом связывается некий набор регистров, включая счетчик команд, указатель стека и другие аппаратные регистры, плюс вся остальная информация, необходимая для запуска программы.

Мы более детально рассмотрим понятие процесса в главе 2, но сейчас для того, чтобы интуитивно осознать, что это такое, вспомним о системах, работающих в режиме разделения времени. Предположим, периодически операционная система решает остановить работу одного процесса и запустить другой, потому что первый израсходовал отведенную для него часть рабочего времени центрального процессора в прошедшую секунду.

Если процесс был приостановлен подобным образом, позже он должен быть запущен заново из того же состояния, в каком его остановили. Следовательно, всю информацию о процессе нужно где-либо явно сохранить на время его приостановки. Например, процесс может иметь открытыми для чтения несколько файлов одновременно. Связанный с каждым файлом указатель дает текущую позицию (то есть номер байта или записи, которые будут прочитаны следующими). При временном прекращении процесса все указатели нужно сохранить так, чтобы команда чтения, выполненная после возобновления процесса, прочла правильные данные. Во многих операционных системах вся информация о каждом процессе, дополнительная к содержимому его собственного адресного пространства, хранится в таблице операционной системы. Эта таблица называется *таблицей процессов* и представляет собой массив (или связанный список) структур, по одной на каждый существующий в данный момент процесс.

Таким образом, приостановленный процесс состоит из собственного адресного пространства, обычно называемого *образом памяти* (core image, core в переводе означает «сердечник», в честь использовавшейся давным-давно памяти на магнитных сердечниках), и компонентов таблицы процесса, содержащей, помимо других величин, его регистры.

Главными системными вызовами, управляющими процессами, являются вызовы, связанные с созданием и окончанием процессов. Рассмотрим типичный пример. Процесс, называемый *интерпретатором команд* или *оболочкой* (shell),

читает команды с терминала. Пользователь только что напечатал команду, содержащую запрос на компиляцию программы. Теперь оболочка должна создать новый процесс, который запустит компилятор. Когда процесс закончит компиляцию, он выполнит системный вызов, завершающий его собственную работу.

Если процесс может создавать несколько других процессов (называющихся *дочерними процессами*), а эти процессы, в свою очередь, тоже могут создать дочерние процессы, перед нами предстает дерево процессов, изображенное на рис. 1.5. Связанные процессы — это те, которые объединены для выполнения некоторой задачи, и им нужно часто передавать данные от одного к другому и синхронизировать свою деятельность. Такая связь называется *межпроцессным взаимодействием* и будет обсуждена в деталях в главе 2.

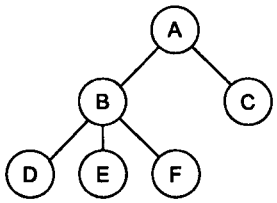


Рис. 1.5. Дерево процесса. Процесс A создал два дочерних процесса B и C. Процесс B создал три дочерних процесса D, E и F

Другие системные вызовы предназначаются для запросов о предоставлении дополнительной памяти (или освобождении не использующейся памяти), ожидании завершения дочерних процессов и наложении одной программы на другую.

Время от времени необходимо передавать информацию работающему процессу так, чтобы он не простаивал в ожидании получения этой информации. Например, процесс, связанный с другим процессом на удаленном компьютере, делает это, посылая сообщения по сети. Чтобы предотвратить возможность потери сообщения или ответа на него, отправитель может потребовать от собственной операционной системы уведомления, если по истечении определенного интервала ожидания не будет получено подтверждение о получении сообщения. В этом случае он сможет повторить отправку сообщения. После установки таймера программа продолжит выполнение другой работы.

Если по истечении определенного количества секунд ответа нет, операционная система посылает процессу *сигнал*. Сигнал вызывает временную остановку работы процесса независимо от того, что процесс делает в данный момент; сохраняет его регистры в стеке и запускает специальную процедуру обработки сигнала (например, передающую повторно предположительно потерянное сообщение). После завершения обработки сигнала работающий процесс запускается заново в том состоянии, в котором он находился до сигнала. Сигналы являются программными аналогами аппаратных прерываний и могут быть сгенерированы по различным причинам, а не только из-за истечения какого-либо интервала времени. Многие аппаратные прерывания (например, вызванные выполнением недо-

пустимой команды или использованием неправильного адреса) также преобразуются в сигналы процессу, в котором произошла ошибка.

Каждому пользователю, которому разрешено пользоваться системой, системный администратор присваивает *UID* (User IDentification — идентификатор пользователя). У каждого работающего процесса есть идентификатор пользователя, запустившего его. Дочерний процесс получает тот же самый UID, что и его родитель. Пользователь с особым идентификатором UID, называемый в UNIX *суперпользователем* (superuser), имеет особые полномочия и может игнорировать множество правил защиты. В огромных системах только системный администратор знает пароль, необходимый для того, чтобы стать суперпользователем. Однако множество обыкновенных пользователей (особенно студенты) тратят значительное количество времени и труда на то, чтобы найти брешь в системе, которая позволит им стать суперпользователями без пароля.

1.3.2. Файлы

Другая обширная группа системных вызовов относится к файловой системе. Как было замечено ранее, основной функцией операционной системы является скрытие особенностей дисков и других устройств ввода/вывода и предоставление пользователю понятной и удобной абстрактной модели независимых от устройств файлов. Системные вызовы очевидно необходимы для создания, удаления, чтения или записи файлов. Перед тем как прочитать файл, его нужно разместить на диске и открыть, а после прочтения его нужно закрыть. Все эти функции осуществляют системные вызовы.

Предоставляя место для хранения файлов, операционные системы используют понятие *каталога* (directory) как способ объединения файлов в группы. Например, студент может иметь по одному каталогу для каждого изучаемого им курса (для программ, необходимых в рамках этого курса), каталог для электронной почты и еще один — для своей домашней веб-страницы. Для создания и удаления каталогов также необходимы системные вызовы. Они же обеспечивают перемещение существующего файла в каталог и удаление файла из каталога. Содержимое каталогов могут составлять файлы или другие каталоги. Эта модель создает структуру — файловую систему, — как показано на рис. 1.6.

Иерархии процессов и файлов организованы в виде деревьев, но на этом сходство заканчивается. Иерархия процессов обычно не очень глубока (в ней редко бывает больше трех уровней), тогда как файловая структура достаточно часто имеет четыре, пять или даже больше уровней в глубину. Иерархия процессов обычно живет очень недолго, как правило, несколько минут, иерархия каталогов может существовать годами. Принадлежность и защита также различны для процессов и файлов. Обычно только родительский процесс может управлять или даже просто иметь доступ к дочернему процессу, однако практически всегда существует механизм, позволяющий читать файлы и каталоги не только владельцу файла, а более широкой группе пользователей.

Каждый файл в иерархии каталогов можно определить, задав его *имя пути*, называемое также полным именем файла. Путь начинается из вершины структу-

ры каталогов, называемой *корневым каталогом*. Такое абсолютное имя пути состоит из списка каталогов, которые нужно пройти от корневого каталога к файлу, с разделением отдельных компонентов косой чертой. На рис. 1.6 путь к файлу CS101 выглядит как /Faculty/Prof.Brown/Courses/CS101. Первая косая черта говорит о том, что этот путь — абсолютный, то есть начинается от корневого каталога. В MS-DOS и Windows для разделения компонентов вместо символа косой черты используется обратная косая черта (\). Тогда этот путь будет выглядеть так: \Faculty\Prof.Brown\Courses\CS101. В нашей книге для записи пути мы в основном будем использовать соглашения UNIX.

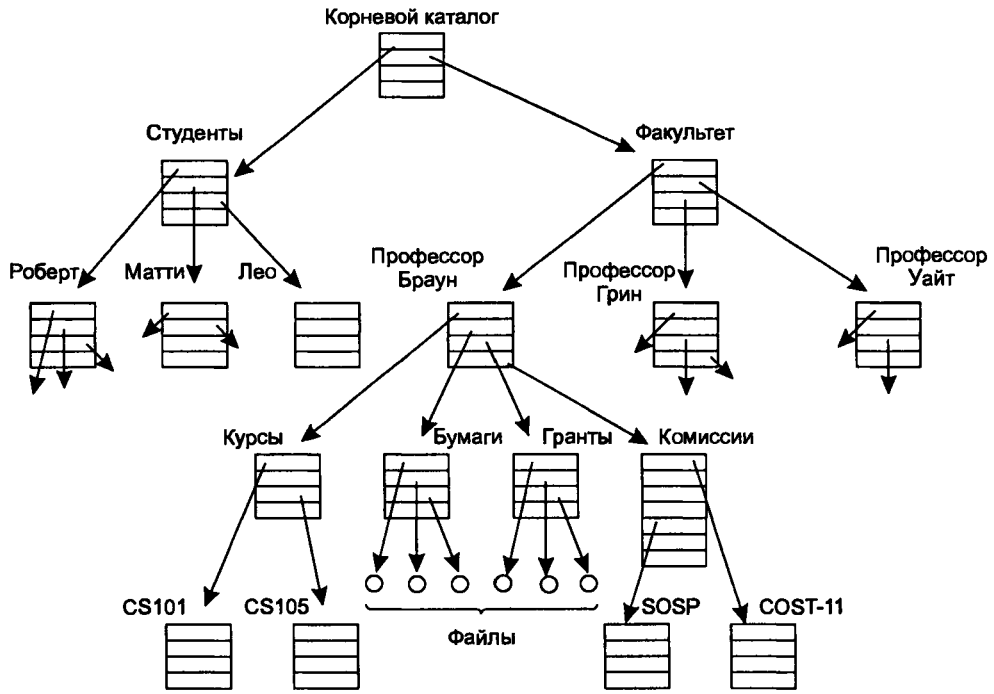
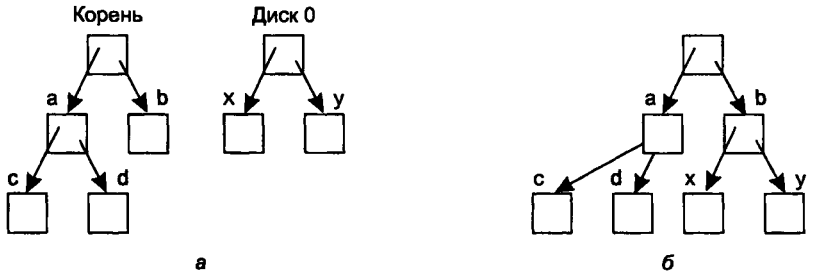


Рис. 1.6. Файловая система факультета университета

В каждый момент времени у каждого процесса есть текущий *рабочий каталог*, в котором ищутся пути файлов, не начинающиеся с косой черты. Например, если на рис. 1.6 /Faculty/Prof.Brown является рабочим каталогом, то использование пути Courses/CS101 даст тот же самый файл, что и абсолютный путь, написанный выше. Процессы могут изменять свой рабочий каталог, используя системные вызовы.

Перед тем как прочесть или записать файл, его нужно открыть, в это же время проверяется разрешение доступа. Если доступ разрешен, система возвращает небольшое целое число, называемое *дескриптором файла* и используемое в последующих операциях. Если доступ запрещен, то возвращается код ошибки.

Другое важное понятие в UNIX — это установленная (смонтированная) файловая система. Почти все персональные компьютеры имеют один или два дисковода для гибких дисков, куда можно вставить и откуда можно вынуть диск. Чтобы предоставить возможность общения со сменными носителями (включая компакт-диски), UNIX позволяет присоединять файловую систему сменного диска к главному дереву. Рассмотрим ситуацию на рис. 1.7, *а*. Перед вызовом системной процедуры `mount` *корневая файловая система* на жестком диске и вторая файловая система на гибком диске существуют раздельно и никак не связаны между собой.



а *б*

Рис. 1.7. *а* — перед установкой файлы на диске 0 недоступны; *б* — после монтирования они становятся частью общей файловой структуры

Однако файлы на гибком диске нельзя использовать, потому что для них невозможно определить путь. UNIX не позволяет присоединять к началу пути название диска или его номер, так как это привело бы к жесткой зависимости от устройств, которой операционная система должна избегать. Вместо этого системный вызов `mount` позволяет присоединять файловую систему на гибком диске к корневой файловой системе в том месте, где этого захочет программа. На рис. 1.7, *б* файловая система гибкого диска была установлена в каталог `b`, таким образом, обеспечен доступ к файлам по путям `/b/x/` и `/b/y`. Если каталог `b` содержал какие-либо файлы, они будут недоступны, пока смонтирован гибкий диск, так как теперь `/b` ссылается на корневой каталог гибкого диска. (Невозможность доступа к этим файлам не так страшна, как кажется с первого взгляда: файловые системы почти всегда устанавливаются в пустые каталоги.) Если система содержит несколько жестких дисков, они все могут быть встроены в одно дерево таким же образом.

Еще одно важное понятие в UNIX — это *специальный файл*. Специальные файлы служат для того, чтобы устройства ввода/вывода выглядели как файлы. При этом можно прочесть информацию из специальных файлов или записать ее туда с помощью тех же самых системных вызовов, что используются для чтения и записи файлов. Существует два вида специальных файлов: *блочные специальные файлы* и *символьные специальные файлы*. Блочные специальные файлы используются для моделирования устройств, состоящих из набора произвольно адресуемых блоков, таких как диски. Открывая блочный специальный файл

и читая, скажем, блок 4, программа может напрямую получить доступ к четвертому блоку на устройстве, без обращения к содержащейся на нем файловой системе. Таким же образом символьные специальные файлы используются для моделирования принтеров, модемов и других устройств, которые принимают или выдают поток символов. По соглашению специальные файлы хранятся в каталоге /dev. Например, /dev/lp может быть строковым принтером.

И последнее понятие, которое мы обсудим во введении, — это каналы (pipe), имеющие отношение и к процессам и к файлам. *Канал* (также иногда называемый трубой) представляет собой псевдофайл, который можно использовать для связи двух процессов, как показано на рис. 1.8. Если процессы *A* и *B* захотят пообщаться с помощью канала, они должны установить его заранее. Когда процесс *A* хочет отправить данные процессу *B*, он пишет их в канал, как если бы это был выходной файл. Процесс *B* может прочесть данные, читая их из канала, как если бы он был файлом с входными данными. Таким образом, соединение между процессами в UNIX выглядит очень похожим на обычное чтение и запись файлов. Более того, только сделав специальный системный вызов, процесс может обнаружить, что выходной файл, в который он пишет данные, не реальный файл, а канал.

Рис. 1.8. Два процесса, соединенные каналом

1.3.3. Оболочка

Операционная система MINIX представляет собой программу, выполняющую системные вызовы. Редакторы, компиляторы, ассемблеры, компоновщики и командные интерпретаторы не являются частью операционной системы, несмотря на их большую важность и полезность. Поскольку есть риск запутаться в этих вещах, в данном разделе мы кратко рассмотрим только командный интерпретатор UNIX, называемый *оболочкой* (shell). Хотя она не входит в операционную систему, но во всю пользуется многими функциями операционной системы и поэтому является хорошим примером того, как могут применяться системные вызовы. Кроме этого, оболочка предоставляет основной интерфейс между пользователем, сидящим за своим терминалом, и операционной системой, если, конечно, пользователь не использует графический интерфейс.

Когда какой-либо пользователь входит в систему, запускается оболочка. Стандартным входным и выходным устройством для оболочки является терминал (монитор с клавиатурой). Оболочка начинает работу с печати *приглашения* (prompt) — знака доллара, говорящего пользователю, что оболочка ожидает ввода команды. Если теперь пользователь напечатает, например,

```
date
```

оболочка создаст дочерний процесс и запустит программу date. Пока дочерний процесс работает, оболочка ожидает его завершения. После завершения дочернего процесса оболочка опять печатает приглашение и пытается прочесть следующую входную строку. Пользователь может перенаправить стандартный вывод данных в файл:

```
date >file
```

Таким же образом можно переопределить устройство, с которого читаются входные данные, как показано ниже:

```
sort <file1 >file2
```

Эта команда предписывает программе сортировки считать данные из файла 1 и вывести результат в файл 2.

Выходные данные одной программы можно использовать в качестве входных данных для другой, соединив их каналом. Так, команда

```
cat file1 file2 file3 | sort >/dev/lp
```

предписывает программе cat объединить (concatenate) три файла и послать выходные данные программе sort, которая расставит все строки в алфавитном порядке. Результат работы sort перенаправляется в файл /dev/lp, обычно обозначающий принтер.

Если пользователь наберет знак & после команды, оболочка не будет ждать окончания ее выполнения. В этом случае она немедленно напишет новое приглашение. То есть в результате команды

```
cat file1 file2 file3 | sort >/dev/lp &
```

сортировка запустится как фоновое задание, разрешая пользователю продолжать нормальную работу во время выполнения сортировки. Оболочка имеет множество других интересных особенностей, для обсуждения которых у нас здесь, к сожалению, недостаточно места. Но большинство книг по UNIX описывают оболочки довольно подробно.

1.4. Системные вызовы

Вооружившись общим пониманием того, как MINIX работает с процессами и файлами, можно приступить к изучению интерфейса между операционной системой и пользовательскими программами, то есть системных вызовов. Несмотря на то что это обсуждение затрагивает конкретно стандарт POSIX (международный стандарт 9945-1) и MINIX, у большинства других современных операционных систем есть системные вызовы, выполняющие те же самые функции, хотя детали могут быть различны. Так как фактический механизм обращения к системным функциям является в высокой степени машинно-зависимым и часто должен реализовываться на ассемблере, существуют библиотеки процедур, делающие возможным обращение к системным процедурам из программ на C и на других языках с тем же успехом.

Для того чтобы прояснить механизм системных вызовов, кратко рассмотрим системный вызов read. Как упоминалось выше, у него есть три параметра: первый служит для задания файла, второй указывает на буфер, третий задает количество байтов, которое нужно прочитать. Как практически все системные вызовы, он запускается из программы на С с помощью вызова библиотечной процедуры с тем же именем, что и системный вызов: read. Вызов из программы на С может выглядеть так:

```
count = read(fd, buffer, nbytes);
```

Системный вызов (и библиотечная процедура) возвращает количество действительно прочитанных байтов в переменной count. Обычно эта величина совпадает с параметром nbytes, но может быть меньше, если, например, в процессе чтения процедуре встретился конец файла.

Если системный вызов не может быть выполнен или из-за неправильных параметров или из-за дисковой ошибки, значение счетчика count устанавливается равным −1, а номер ошибки помещается в глобальную переменную errno. Программы всегда должны проверять результат системного вызова, чтобы отслеживать появление ошибки.

У MINIX в целом имеется 53 системных вызова. Те из них, которые перечислены в табл. 1.1, для удобства разбиты на шесть групп. В последующих секциях мы кратко рассмотрим каждый вызов, чтобы увидеть, что он делает. В целом, выполняемые этими системными вызовами функции определяют большую часть возможностей операционной системы, так как управление ресурсами на персональных компьютерах сведено к минимуму (по крайней мере, по сравнению с большими машинами, обслуживающими множество пользователей).

Таблица 1.1. Некоторые из основных системных вызовов MINIX

Вызов	Описание
Управление процессами	
pid = fork( )[1]	Создает дочерний процесс, идентичный родительскому
pid = waitpid(pid, &statloc, options)	Ожидает завершения дочернего процесса
s = wait(&status)	Старая версия waitpid
s = execve(name, argv, environp)	Перемещает образ памяти процесса
Exit(status)	Завершает выполнение процесса и возвращает статус
size = brk(addr)	Устанавливает размер сегмента данных
pid = getpid()	Возвращает идентификатор процесса, сделавшего вызов
pid = getgrp()	Возвращает идентификатор группы процессов для сделавшего вызов процесса
pid = setsid()	Создает новую сессию и возвращает ее идентификатор группы процессов
l = ptrace(req, pid, addr, data)	Используется для отладки
Сигналы	
s = sigaction(sig, &act, &oldact)	Устанавливает реакцию на сигнал

продолжение

Таблица 1.1 (*продолжение*)

Вызов	Описвние
s = sigreturn(&context)	Возвращается из обработчика сигнала
s = sigprocmask(howm &set, &old)	Определяет или устанавливает маску сигналов для процесса
s = sigpending(set)	Определяет набор блокированных сигналов
s = sigsuspend(sigmask)	Устанавливает маску сигналов для процесса и приостанавливает его
s = kill(pid, sig)	Посылает сигнал процессу
residual = alarm(seconds)	Устанавливает сигнальный таймер
s = pause()	Приостанавливает процесс до прихода следующего сигнала
Управление файлами	
fd = creat(name, mode)	Устаревший способ создать файл
fd =mknod(name, mode, addr)	Создает обычный, специальный или относящийся к директории i-узел
fd=open(file, how, ...)	Открывает файл для чтения, записи или того и другого
s = close(fd)	Закрывает открытый файл
n = read(fd, buffer, nbytes)	Читает данные из файла в буфер
n = write(fd, buffer, nbytes)	Пишет данные из буфера в файл
pos = lseek(fd, offset, whence)	Передвигает указатель файла
s = stat(name, &buf)	Получает информацию о состоянии файла
s = fstat(fd, &buf)	Получает информацию о состоянии файла
fd = dup(fd)	Закрепляет за открытым файлом новый дескриптор
s = pipe(&fd[0])	Создает канал
s = ioctl(fd, request, argp)	Специальные действия с файлом
s = access(name, amode)	Проверить доступность файла
s = rename(old, new)	Переименовать файл
s = fcntl(fd, cmd, ...)	Захват файла и другие действия
Управление каталогами и файловой системой	
s = mkdir(name, mode)	Создает новый каталог
s = rmdir(name)	Удаляет пустой каталог
s = link(name1, name2)	Создает новый элемент с именем name2, указывающий на name1
s = unlink(name)	Удаляет элемент каталога
s = mount(special, name, flag)	Монтирует файловую систему
s = umount(special)	Демонтирует файловую систему
s = sync()	Сбросить все кэшированные блоки на диск
s = chdir(dirname)	Изменяет рабочий каталог
s = chroot(dirname)	Изменяет корневую директорию
Защита	
s = chmod(name, mode)	Изменяет биты защиты файла

Вызов	Описание
uid = getuid()	Определяет идентификатор сделавшего вызов
gid = getgid()	Определяет идентификатор группы сделавшего вызов
s = setuid(uid)	Устанавливает идентификатор пользователя
s = setgid(gid)	Устанавливает идентификатор группы
s = chown(name, owner, group)	Меняет идентификатор владельца файла
oldmask = umask(complmode)	Устанавливает маскирование разрешений
Работа со временем	
seconds = time(&seconds)	Получает время, прошедшее с 1 января 1970 года
s = stime(tp)	Устанавливает время, прошедшее с 1 января 1970 года
s = utime(file, timep)	Устанавливает время последнего доступа к файлу
s = times(buffer)	Определяет время работы пользовательского процесса и системы

[1] Возвращаемая величина s равна −1, если произошла ошибка. Возвращаемые коды выглядят так: *pid* выдает идентификатор процесса, *fd* — описатель файла, *n* — количество байтов, *position* — смещение в файле и *seconds* — прошедшее время. Параметры описываются дальше в тексте.

Особое внимание следует обратить на то, что преобразование вызовов процедур POSIX в системные вызовы не является взаимно однозначным. Стандарт POSIX определяет ряд процедур, которые должны поддерживать совместимые системы, но он не указывает, являются ли они системными вызовами, библиотечными вызовами или чем-нибудь еще. В некоторых случаях, особенно когда требуемые процедуры являются всего лишь разновидностями друг друга, один системный вызов обрабатывает сразу несколько библиотечных вызовов.

1.4.1. Системные вызовы для управления процессами

Первая группа в табл. 1.1 управляет процессами. Начнем рассмотрение с вызова fork. Системный вызов fork (разветвление) является единственным способом создания нового процесса в UNIX. Он создает точную копию исходного процесса, включая дескрипторы файла, регистры и т. п. После вызова fork исходный процесс и его копия (родительский и дочерний) развиваются по отдельности друг от друга. Все переменные имеют одинаковые величины во время вызова fork, но как только родительские данные скопированы для создания дочернего процесса, последующие изменения в одном из них уже не влияют на другой. (Текст программы, который не изменяется, распределяется между родительским и дочерним процессами.) Вызов fork возвращает величину, равную нулю в дочернем процессе и равную идентификатору дочернего процесса или *PID* в родительском. Используя возвращенный PID, два процесса могут различить, какой из них родительский, а какой — дочерний.

В большинстве случаев после вызова fork дочернему процессу необходимо выполнить программный код, отличный от предназначенного для родительского процесса. Рассмотрим пример оболочки. Она читает команды с терминала, за-

пускает дочерний процесс, ждет, пока дочерний процесс выполнит команду, и читает следующую команду после завершения работы дочернего процесса. Ожидая, пока дочерний процесс закончит работу, родительский процесс выполняет системный вызов **waitpid**, который ожидает завершения дочернего процесса (или всех дочерних процессов, если их на данный момент несколько). **Waitpid** может ждать окончания какого-либо определенного дочернего процесса или любого дочернего процесса, для этого нужно задать первый параметр вызова равным –1. Когда **waitpid** выполнен, указатель, задаваемый вторым параметром **statloc**, будет установлен на статус завершения дочернего процесса (нормальное или аварийное завершение и выходное значение). Третий параметр определяет различные необязательные настройки.

Теперь рассмотрим, как вызов **fork** используется оболочкой. Когда печатается команда, оболочка создает дочерний процесс, который должен выполнить команду пользователя. Он делает это с помощью системного вызова **exec**, заменяющего весь его образ памяти файлом, названным в первом параметре. (Фактически самим системным вызовом является **exec**, но несколько различных библиотечных процедур вызывают его с разными параметрами и незначительно отличающимися именами. Мы здесь воспользуемся ими как системными вызовами.) Весьма упрощенная оболочка, иллюстрирующая использование команд **fork**, **waitpid** и **exec**, показана в листинге 1.1.

Листинг 1.1. Усеченная оболочка[1]

```
#define TRUE 1
while (TRUE) {                                  /* вечный цикл */
     type_prompt( );                            /* печать приглашения на  экране */
     read_command(command, parameters);         /* читать входные данные с терминала */
     if (fork( ) != 0) {                        /* запускает дочерний процесс */
                                                /* текст родительского процесса */
        waitpid(-1, &status, 0);                /* ждать окончания дочернего процесса */
     } else {
        /* текст дочернего процесса */
        execve(command, parameters, 0);         /* выполнение command */
     }
}
```

В самом общем случае у команды **exec** есть три параметра: имя файла, который будет выполняться, указатель на массив аргументов и указатель на массив переменных окружения. Эти параметры мы кратко обсудим в дальнейшем. Различные библиотечные программы, включая **execl**, **execv**, **execle** и **execve**, разрешают пропускать параметры или определять их другими способами. В книге мы воспользуемся названием **exec** для того, чтобы представить системный вызов, вызываемый всеми этими процедурами.

Рассмотрим следующую команду:

```
cp file1 file2
```

которая используется для копирования файла **file1** в файл **file2**. После создания оболочкой дочернего процесса последний находит и исполняет файл **cp** и передает ему имена исходного и целевого файлов.

[1] На протяжении всей книги значение константы *TRUE* предполагается равным 1.

Основной модуль программы cp (как и большинство других головных программ на C) содержит определение:

```
main(argc, argv, envp)
```

в котором в параметр *argc* входит количество записей в командной строке, включая имя программы. Например, для строки вверху argc равен 3.

Второй параметр argv является указателем на массив указателей. Элемент *i* массива указывает на *i*-ю запись в командной строке. В нашем примере argv[0] должен указывать на слово cp, а argv[1] и argv[2] — на слова file1 и file2 соответственно.

Третий параметр функции main, envp, является указателем на массив строковых переменных окружения вида *имя* = *величина*, которые используются для передачи программе такой информации, как тип терминала или имя домашнего каталога. В листинге 1.1 третий параметр равен нулю, поскольку ничего не передается дочернему процессу.

Если команда exec кажется сложной, не огорчайтесь, потому что это один из наиболее сложных системных вызовов в POSIX. Все остальные намного проще. В качестве еще одного примера рассмотрим exit, процессы должны использовать его при завершении работы. У него есть всего один параметр, статус выхода, изменяющийся от 0 до 255. Он возвращается родительскому процессу через переменную *status* в системных вызовах wait и waitpid. Младший байт этой переменной содержит значение статуса выхода, который равен 0 при нормальном завершении работы и ненулевому значению при завершении по ошибке. Старший байт содержит статус завершения дочернего процесса. Например, если родительский процесс выполнит оператор:

```
n = waitpid(-1, &status, options);
```

то его работа будет приостановлена до завершения дочернего процесса. Если дочерний процесс завершится, скажем, через exit с параметром 4, то, когда родительский процесс продолжит работу, n будет содержать PID дочернего процесса, а status — значение 0x0400 (в языке C принято писать 0x перед шестнадцатеричной записью чисел, это соглашение всегда будет использоваться в книге).

В MINIX под процессы отводится часть памяти, которая, в свою очередь, делится на три сегмента: *текстовый* (то есть код программы), *сегмент данных* (переменные) и *сегмент стека*. Сегмент данных растет снизу вверх, а стек увеличивается сверху вниз, как показано на рис. 1.9. Между ними существует часть неиспользованного адресного пространства. Стек автоматически занимает такую часть этого участка памяти, какую необходимо, но расширение сегмента данных выполняется явным образом. Для этого используется специальный системный вызов brk, задающий новый адрес для границы сегмента данных. Этот адрес может быть как больше текущего значения (сегмент растет), так и меньше (сегмент уменьшается). Но, конечно же, этот адрес должен быть меньше, чем указатель стека, так как в противном случае данные и стек могут перекрываться, что запрещено.

Для удобства программиста предлагается библиотечная процедура *sbrk*, также меняющая размер сегмента данных. Ее единственный параметр указывает, на

сколько должен быть увеличен размер сегмента (чтобы уменьшить сегмент, нужно передавать отрицательные значения). Процедура работает так: с помощью brk определяется текущий размер сегмента, затем вычисляется новый размер, после чего делается еще один системный вызов, запрашивающий требуемое количество байт. Как brk, так и sbrk не являются частью стандарта POSIX и зависят от реализации.

Следующий системный вызов для работы с процессами является заодно и простейшим из них, это GETPID. Этот вызов просто возвращает идентификатор вызвавшего его процесса. Обратите внимание на то, что при вызове FORK значение идентификатора дочернего процесса получает только родительский процесс. Если дочернему процессу потребуется узнать собственный PID, ему придется использовать GETPID. Вызов GETGRP возвращает идентификатор группы процессов, в которую входит процесс, сделавший вызов. Вызов SETSID создает новую сессию и устанавливает PID группы процессов равным PID процесса, сделавшего вызов. Сессии относятся к необязательной возможности POSIX, называемой *управлением задачами*, которая не реализована в MINIX и не будет беспокоить нас в дальнейшем.

Последний системный вызов из этой группы, PTRACE, используется отладчиками для управления работой отлаживаемой программы. Он позволяет отладчику обращаться к памяти отлаживаемого процесса, а также управлять им другими способами.

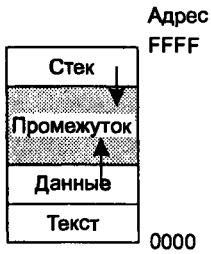


Рис. 1.9. Под процессы отводится три сегмента: текст, данные и стек

1.4.2. Системные вызовы для управления сигналами

Хотя обычно все взаимодействие между процессами запланировано, существуют ситуации, когда требуется незапланированное взаимодействие. Например, если пользователь случайно потребовал от текстового редактора показать содержимое очень большого файла и заметил свою ошибку. Должен существовать способ прервать работу редактора. В MINIX пользователь может нажать на клавиатуре клавишу Del, которая пошлет текстовому редактору соответствующий сигнал. Поймав его, редактор прекращает распечатку. Сигналы могут использоваться и для того, чтобы отслеживать некоторые аппаратные исключения, например

недопустимые инструкции или переполнение при операциях с плавающей запятой. Задержки также реализуются через сигналы.

Если процесс никак не анонсировал свое желание отвечать на сигналы, то, получив сигнал, он просто принудительно завершается. Чтобы избежать подобной судьбы и рассказать о своем желании отвечать на сигналы определенного типа, процесс может использовать системный вызов SIGACTION, который позволяет указать новый адрес процедуры для обработки сигнала и узнать адрес прежней процедуры. После того как такой вызов сделан, при получении сигнала соответствующего типа (например, при нажатии клавиши Del) состояние процесса будет сохранено в стеке и вызван обработчик сигнала. Обработчик может работать сколь угодно долго и делать при этом любые системные вызовы. Но на практике обработчики сигналов обычно весьма коротки. Завершив свою работу, обработчик вызывает SIGRETURN, чтобы продолжить работу процесса с того места, где он был прерван. Вызов SIGACTION заменяет прежний системный вызов SIGNAL, который теперь, для обратной совместимости, реализован в виде библиотечной процедуры.

В MINIX сигналы можно блокировать. Блокированный сигнал задерживается до тех пор, пока он не будет разблокирован. То есть, сигнал не доставляется, но и не теряется. Вызов SIGPROCMASK позволяет процессу задать набор блокируемых сигналов, передавая ядру битовую карту. Кроме того, процесс может узнать, какие сигналы произошли, но были заблокированы. Это позволяет сделать системный вызов SIGPENDING, возвращающий набор сигналов в виде битовой маски. Наконец, системный вызов SIGSUSPEND позволяет процессу атомарно задать битовую карту блокированных сигналов и приостановить свое выполнение.

Вместо того чтобы передавать в качестве обработчика сигнала указатель на функцию, программа может использовать константу SIG_IGN, чтобы все последующие сигналы определенного типа игнорировались. Передав константу SIG_DFL, можно восстановить обработку сигнала по умолчанию. По умолчанию процесс либо принудительно завершается по сигналу, либо сигнал просто игнорируется. Это зависит от конкретного сигнала. В качестве примера того, как используется SIG_IGN, рассмотрим работу оболочки при запуске фонового процесса с помощью такой команды:

```
command &
```

В этом случае сигнал Del от клавиатуры не должен влиять на работу фонового процесса, поэтому после вызова FORC, но перед EXEC оболочка делает следующие вызовы:

```
sigaction(SIGINT, SIG_IGN, NULL);
```

и

```
sigaction(SIGQUIT, SIG_IGN, NULL);
```

Эти вызовы отключают обработку сигнала DEL и сигнала принудительного завершения процесса (этот сигнал вызывается нажатием Ctrl+\ и делает то же самое, что и Del, но, если его не обрабатывать или не блокировать, делает распечат-

ку памяти завершенного процесса). Для обычных процессов, выполняющихся не в фоновом режиме, игнорировать сигналы не нужно.

Нажать Del — не единственный способ послать процессу сигнал. Системный вызов KILL позволяет одному процессу послать сигнал другому (у обоих процессов должен быть одинаковый uid, так как не связанные процессы не могут послать друг другу сигналы). Возвращаясь к рассмотренному выше примеру, предположим, что запущенный фоновый процесс потребовалось завершить. Сигналы SIGQUIT и SIGINT этот процесс игнорирует, так что требуется что-то другое. Решение дает программа kill, которая с помощью системного вызова KILL может послать сигнал любому процессу. Процесс можно принудительно завершить, послав ему сигал 9 (SIGKILL). Этот сигнал нельзя обработать или игнорировать.

Во многих приложениях реального времени процесс нужно прерывать по истечении некоторого интервала времени, за который он должен успеть выполнить свою задачу. Например, повторно передать пакет данных по ненадежной линии связи. Для этой ситуации предназначен системный вызов ALARM. Параметр этого вызова описывает интервал времени, по истечении которого процессу передается сигнал SIGALARM. В любой момент времени процесс может запланировать только один сигнал. Если сделать вызов ALARM, указав задержку 10 секунд, а затем, по истечении 3 секунд, еще раз вызвать ALARM с параметром 20 секунд, придет сигнал только от того вызова, который сделан последним. Первый вызов отменяется. Чтобы отменить сделанный ранее вызов ALARM, нужно еще раз вызвать ALARM, передав в качестве аргумента 0. Если пришедший сигнал SIGALARM не обрабатывать, то выполняется действие по умолчанию, и процесс завершается.

Иногда возникают ситуации, когда процессу нечем заняться до прибытия сигнала. Например, рассмотрим компьютерную программу для проверки скорости чтения и внимательности. Эта программа выводит некоторый текст, а затем делает вызов ALARM с параметром 30 секунд. Пока ученик читает текст, программа ничего не должна делать. Программа может выполнять пустой цикл, но это будет бессмысленным расточением процессорного времени, которое может потребоваться другому процессу. Гораздо лучше использовать системный вызов PAUSE, который указывает MINIX приостановить выполнение процесса до прихода сигнала.

1.4.3. Системные вызовы для управления файлами

Многие системные вызовы имеют отношение к файловой системе. В этом разделе мы рассмотрим вызовы, работающие с отдельными файлами, а в следующем разделе обратимся к тем, которые оперируют каталогами или файловой системой в целом. Для создания нового файла служит вызов CREAT (ответ на вопрос, почему именно CREAT, а не CREATE затерялся в тумане времен). Его параметры — имя файла и права доступа. Так, команда

```
fd = creat("abc", 0751);
```

создаст файл с именем abc и установит для него права доступа 0751 (в C числа с ведущим нулем считаются восьмеричными). Младшие 9 бит этого числа пока-

зывают, что владелец файла имеет права доступа rwx (7 означает права на чтение, запись и исполнение), члены группы владельца имеют права на чтение и исполнение (5), а прочие пользователи — только на исполнение (1).

CREAT не только создает новый файл, но и открывает его на чтение, независимо от указанного режима. Дальнейшая работа с файлом производится через его дескриптор, значение которого возвращается вызовом. Если вызвать CREAT для существующего файла, то файл усекается до нулевой длины (если, конечно, права доступа позволяют это). Сейчас вызов CREAT устарел и поддерживается для обратной совместимости, вместо него нужно использовать OPEN.

Чтобы создать специальный файл, нужно вызывать не CREAT, а MKNOD. Вот пример его типичного использования:

```
fd = mknod("/dev/ttyc2", 020744, 0x0402).
```

Эта команда создает файл с именем /dev/ttyc2 (обычно это имя соответствует второй консоли) и задает для него права доступа 020744 (специальный символьный файл с правами rwxr--r--). Третий параметр составлен из двух байт, из которых старший задает основное устройство (4), а младший — вторичное устройство (2). Основное устройство могло бы быть любым, а вторичное для /dev/ttyc2 должно быть равно 2. Делать вызов MKNOD может только суперпользователь, в противном случае возникает ошибка.

Чтобы прочитать или записать файл, его сначала нужно открыть при помощи вызова OPEN. Для этого вызова указывается имя открываемого файла (задается или абсолютный путь файла, или ссылка на рабочий каталог) и код O_RDONLY, O_WRONLY или O_RDWR, означающий, что файл открывается для чтения, записи или и того и другого. Для создания нового файла используется код O_CREAT. Возвращаемый дескриптор файла затем можно употребить при чтении или записи. Потом файл закрывается с помощью вызова CLOSE, который делает дескриптор файла доступным при следующем открытии (OPEN).

Наиболее часто используемыми вызовами, без сомнения, являются READ и WRITE. Вызов READ мы уже обсуждали, WRITE имеет те же самые параметры.

Несмотря на то что большинство программ читает и записывает файлы с помощью последовательного доступа, некоторым прикладным программам необходима возможность доступа к любой, случайно выбранной части файла. Связанный с каждым файлом указатель содержит текущую позицию в файле. Когда чтение (запись) осуществляется последовательно, он обычно указывает на байт, который должен быть прочитан (записан) следующим. Вызов LSEEK может изменить значение позиции указателя, так что следующий вызов READ или WRITE начнет операцию где-либо в другой части файла.

У вызова LSEEK есть три параметра: первый — это идентификатор файла, второй — позиция в файле, а третий говорит, является ли второй параметр позицией в файле относительно начала файла (абсолютная позиция), относительно текущей позиции или относительно конца файла. Вызов LSEEK возвращает абсолютную позицию в файле после изменения указателя.

Для каждого файла MINIX хранит следующие данные: тип файла (обычный, специальный, каталог и т. д.), размер, время последнего изменения и другую информацию. Программа может запросить эту информацию через системные вызо-

вы STAT и FSTAT. Они различаются только тем, что первый из них требует имени файла, а второй полагается на дескриптор файла и полезен для открытых файлов, особенно для файлов стандартного ввода и вывода, имена которых не всегда известны. У обоих вызовов второй параметр указывает на структуру, куда нужно поместить информацию. Эта структура показана на листинге 1.2.

Листинг 1.2. Структура, используемая для получения информации от системных вызовов STAT и FSTAT

```
struct stat {
   short st_dev;              /*устройство, которому принадлежит inode*/
   unsigned short st_ino;     /*номер inode*/
   unsigned short st_mode;    /*режим доступа*/
   short st_nlink;            /*число ссылок на файл*/
   short st_uid;              /*идентификатор пользователя*/
   short st_gid;              /*идентификатор группы*/
   short st_rdev;             /*основное и вторичное */
                              /*устройство для специальных файлов*/
   long st_size;              /*размер файла*/
   long st_atime;             /*время последнего обращения*/
   long st_mtime;             /*время последнего изменения*/
   long st_ctime;             /*время последнего изменения inode*/
};
```

При работе с файловыми дескрипторами изредка может оказаться полезным системный вызов DUP. Например, рассмотрим программу, которая закрывает стандартный вывод (дескриптор 1), подставляет в качестве стандартного вывода другой файл, вызывает функцию, которая что-то пишет в этот файл, а затем восстанавливает исходное состояние. Если просто закрыть стандартный вывод и открыть новый файл, то новый файл станет стандартным выводом (если стандартный ввод, дескриптор которого равен 0, используется), но восстановить состояние будет невозможно. Решение дает следующий оператор, который использует системный вызов DUP:

```
fd = dup(1);
```

При его выполнении в переменную fd помещается новый дескриптор, который будет соответствовать тому же файлу, что и стандартный вывод (1). Затем стандартный вывод можно закрыть, после чего открыть новый файл и использовать его. Когда понадобится восстановить исходное состояние, нужно закрыть файловый дескриптор 1 и выполнить код:

```
n = dup(fd);
```

В результате самый меньший из файловых дескрипторов, а именно 1, станет соответствовать тому же файлу, что и fd. Наконец, fd можно закрыть, в результате чего мы вернемся к той же ситуации, с которой начали.

У системного вызова dup есть второй вариант, с помощью которого можно связать неиспользованный дескриптор с уже открытым файлом. Он записывается так:

```
dup2(fd, fd2);
```

Здесь fd это дескриптор открытого файла, а fd2 — не связанный ни с каким файлом дескриптор, который после выполнения вызова будет ссылаться на тот

же файл. Таким образом, если fd ссылается на стандартный ввод (дескриптор равен 0), а дескриптор fd2 равен 4, то после выполнения вызова и 0, и 4 будут соответствовать стандартному вводу.

Для обеспечения взаимодействия между процессами в MINIX можно использовать каналы (pipes), как это уже было сказано выше. Когда пользователь дает оболочке команду

```
cat file1 file2 | sort
```

то оболочка создает канал и соединяет стандартный вывод первого процесса со входом канала, а стандартный ввод второго процесса с выходом. Чтобы создать канал, применяется системный вызов pipe, возвращающий два файловых дескриптора: один для чтения из канала и другой для записи в него. Обычно после этого делается вызов fork, и родительский процесс закрывает дескриптор для чтения, а дочерний процесс — дескриптор для записи (или наоборот), чтобы один процесс мог писать в канал, а другой — читать из него.

В листинге 1.3 приведен скелет процедуры, создающей два процесса таким образом, что выход первого из них передается через канал во второй (более реалистичный пример делал бы проверку ошибок и обработку аргументов). Сначала создается поток, затем делается вызов fork, и родительский процесс становится первым процессом в канале, а дочерний процесс — вторым. Так как запускаемые файлы, process1 и process2, ничего не знают о том, что они соединяются каналом, для работы программы необходимо, чтобы стандартный вывод первого процесса был соединен со стандартным вводом второго процесса каналом. Сначала родительский процесс закрывает дескриптор для чтения из канала. Затем он закрывает стандартный вывод и делает вызов dup, после которого стандартным выводом становится вход канала. Важно понимать, что dup всегда возвращает наименьший возможный дескриптор, в данном случае это будет 1. Наконец, исходный дескриптор для записи в канал закрывается.

Листинг 1.3. Скелет процедуры, создающей конвейер из двух процессов

```
#define STD_INPUT 0          /* Дескриптор файла для стандартного устройства ввода */
#define STD_OUTPUT 1         /* Дескриптор файла для стандартного устройства вывода*/

pipeline(process1, process2) /* Указатели на имена программ */
char *process1, *process2;
{
  int fd[2];

  pipe(&fd[0]);              /* создать конвейер */
  if(fork() !=0) {
     /* Эти выражения исполняются родительским процессом */
     close(fd[0]);           /* Процесс 1 не нуждается в считывании из конвейера */
     close(STD_OUTPUT);      /* Подготовка нового стандартного устройства вывода */
     dup(fd[1]);             /* Сделать стандартным устройством вывода fd[1] */
     close(fd[1]);           /* Этот дескриптор файла больше не нужен */
     execl(process1,process1,0);
  } else {
     /* Эти выражения исполняются процессом-наследником */
     close(fd[1]);           /* Процесс 2 не нуждается в записи в конвейер */
     close(STD_INPUT);       /* Подготовка нового стандартного устройства ввода */
     dup(fd[0]);             /* Сделать стандартным устройством ввода fd[0] */
```

```
    close(fd[0]);           /* Этот дескриптор файла больше не нужен */
    execl(process2,process2,0);
  }
}
```

После вызова exec стартует процесс, у которого дескрипторы 0 и 2 остались без изменений, а дескриптор 1 соответствует записи в канал. Код дочернего процесса аналогичен. Значение параметра функции execl повторяется потому, что первым ее параметром должно быть имя программы, а вторым — значение первого аргумента запускаемой программы, которое для большинства программ должно совпадать с именем.

Следующий системный вызов, ioctl, потенциально применим ко всем специальным файлам. Например, он используется драйверами блочных устройств, таких как SCSI драйвера для работы с ленточными накопителями и CD-ROM. Тем не менее он в основном применяется к символьным специальным файлам, особенно к терминалам. В стандарте POSIX определено несколько функций, которые транслируются библиотекой в вызовы ioctl. С помощью функций tcgetattr и tcsetattr можно изменить характеристики терминала, такие как коррекция ошибок, режим и т. д. Эти функции используют вызов ioctl.

Режим с обработкой (cooked mode) — это нормальный режим работы терминала, в котором удаление символов работает нормально, символы Ctrl+S и Ctrl+Q соответственно останавливают и запускают вывод информации на терминал, Ctrl+D означает конец файла, а Del генерирует сигнал прерывания. Нажатие Ctrl+\ в этом режиме генерирует сигнал, по которому процесс принудительно прерывается, и выводится дамп памяти.

В *«сыром» режиме* (raw mode) вся эта дополнительная обработка отключается, и символы передаются программе напрямую. Более того, в этом режиме терминал не ждет окончания ввода строки, а передает символы программе сразу.

Режим cbreak (cbreak mode) — промежуточный между двумя предыдущими. В этом режиме при редактировании не работают символы удаления и Ctrl+D, но Ctrl+S, Ctrl+Q, Del и Ctrl+\ работают. Как и в «сыром» режиме, символы передаются программам сразу, не дожидаясь окончания ввода строки (так как редактирование строк не работает, не обязательно дожидаться окончания ввода, потому что пользователь не сможет передумать и удалить введенные символы, как в режиме с обработкой).

Термины cooked mode, raw mode и cbreak mode в стандарте POSIX не используются. Вместо этого POSIX определяет *канонический режим* (canonical mode), соответствующий режиму с обработкой. В нем определено одиннадцать специальных символов, и ввод ведется построчно. В *неканоническом режиме* (noncanonical mode) ввод данных определяется минимальным воспринимаемым количеством символов и временем (в десятых долях секунды). Стандарт POSIX очень гибок и определяет множество различных флагов, управляя которыми можно приблизить неканонический режим как к «сырому» режиму, так и к режиму cbreak. Старые термины более содержательны, поэтому мы неформально будем придерживаться их.

У вызова ioctl есть три параметра. Например, вызов функции *tcsetattr* преобразуется в следующий вызов:

```
ioctl(fd, TCSETS, &termios);
```

Первый параметр задает файл, второй — выполняемую операцию, а третий — адрес структуры POSIX, содержащей флаги и массив управляющих символов. Другие коды операций позволяют отложить изменения до завершения вывода, считать текущие значения параметров и отбросить не до конца считанные данные.

Системный вызов access позволяет узнать, разрешено ли системой ограничения доступа обращение к определенному файлу. Этот вызов необходим потому, что некоторые программы могут работать под другим идентификатором пользователя. Для этого программами используется механизм setuid, который будет описан далее.

Чтобы присвоить файлу новое имя, применяется системный вызов rename. Его параметры задают старое и новое имя файла.

Наконец, для управления файлами служит вызов fcntl, в чем-то подобный вызову ioctl (иными словами, оба этих вызова — грязный хак). У этого вызова есть несколько параметров, самые важные из которых служат для управления захватом файла. С помощью fcntl можно захватывать и освобождать отдельные части файлов, а также определять, захвачен ли нужный участок. Этот вызов никак не определяет семантику захвата файла. Программы должны сами решать, что делать.

1.4.4. Системные вызовы для управления каталогами

В этом разделе мы рассмотрим некоторые системные вызовы, относящиеся скорее к каталогам и файловой системе в целом, нежели просто к определенному файлу, как в предыдущем разделе. Первые два вызова, mkdir и rmdir, соответственно создают и удаляют пустые каталоги. Следующий вызов — link. Он разрешает одному файлу появляться под двумя или более именами, часто в разных каталогах. Этот вызов обычно используется, когда несколько программистов, работающих в одной команде, должны совместно использовать один общий файл. Тогда этот файл может появиться в каталоге у каждого из программистов, возможно, под другим именем. Разделение (совместное использование) файла — это не то же самое, что копирование файла для каждого члена команды. При разделении файла изменения, производимые одним программистом, немедленно становятся видимыми для остальных — все происходит в одном файле. А при создании копии файла последующие изменения не влияют на другие копии этого файла.

Чтобы увидеть, как работает вызов link, рассмотрим ситуацию на рис. 1.10, *а*. Два пользователя, *ast* и *jim*, имеют свои собственные каталоги ast и jim с файлами. Если теперь пользователь *ast* запустит программу, содержащую системный вызов

```
link("/usr/jim/memo", "/usr/ast/note");
```

то файл memo в каталоге Джима появится в каталоге Аста под названием note. Соответственно, /usr/jim/memo и /usr/ast/note теперь будут ссылаться на один и тот же файл. Хранятся ли каталоги пользователей в каталоге /usr, /user, /home или где-либо еще, определяется локальным системным администратором.

Возможно, станет понятнее, что делает системный вызов link, если разобраться в том, как он работает. Каждый файл в MINIX имеет уникальный номер —

свой i-номер, который идентифицирует файл (identification — идентификация). i-номер — это индекс в таблице *i-узлов* (i-nodes), содержащей по одному идентификатору на файл. Каждый i-узел включает в себя информацию о хозяине файла, о том, какие блоки на диске он занимает и т. д. Каталог представляет собой просто файл, содержащий набор пар (i-номер, ASCII-имя). На рис. 1.10, *б* два элемента имеют одинаковый i-номер (70) и, таким образом, ссылаются на один и тот же файл. Если впоследствии один из них будет удален с помощью системного вызова `unlink`, другой элемент останется. Если будут удалены оба файла, MINIX увидит, что больше нет записей, соответствующих этому файлу (поле в таблице i-узлов хранит данные с номером элемента каталога, указывающие на файл), и удалит файл с диска.

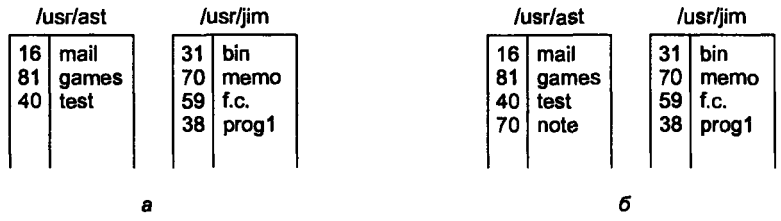


Рис. 1.10. а — два каталога до присоединения /usr/jim/memo к каталогу ast; *б* — те же каталоги после вызова link

Как упоминалось выше, системный вызов `mount` позволяет объединять в одну две файловых системы. Обычная ситуация такова: на жестком диске находится корневая файловая система, содержащая двоичные (исполняемые) версии общих команд и наиболее часто использующиеся файлы. При этом пользователь может вставить в дисковод гибкий диск для чтения файлов.

При помощи системного вызова `mount` файловую систему с гибкого диска можно присоединить к корневой файловой системе, как показано на рис. 1.11. Типичный оператор на языке C, выполняющий монтирование, выглядит так:

```
mount("/dev/fd0", "/mnt", 0);
```

где первым параметром является имя специального блочного файла на диске 0, второй параметр — это место в дереве, куда будет вмонтирована файловая система, а третий параметр говорит о том, монтируется ли встроенная файловая система для чтения и записи или только для чтения.

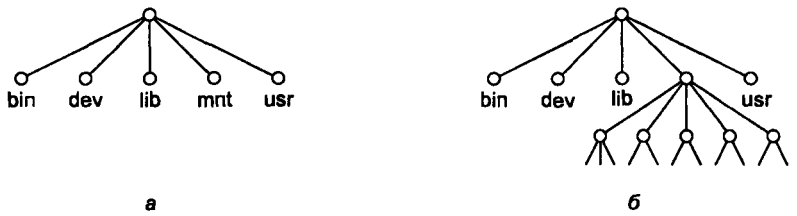


Рис. 1.11. Файловая система: *а* — до вызова mount; *б* — после вызова mount

После вызова mount доступ к файлу на диске 0 можно получить, просто указав его путь из корневого или рабочего каталога, независимо от того, на каком диске он находится. В действительности второй, третий и четвертый диски тоже можно встроить в любое удобное место в дереве. Вызов mount позволяет объединить съемные носители в единую интегрированную файловую структуру, не заботясь о том, на каком из устройств фактически находится файл. Хотя в нашем примере рассматривались гибкие диски, жесткие диски или их части (часто называемые *разделами* (partition) или *младшими устройствами* (minor devices)) монтируются аналогично. Когда файловая система более не нужна, ее можно демонтировать с помощью системного вызова umount.

MINIX кэширует в памяти последние блоки, к которым были совершены обращения, чтобы избежать слишком частых обращений к диску. Если блок в памяти модифицирован (например, вызовом write) и система рухнет до того, как обновленный блок будет записан на диск, файловая система будет повреждена. Чтобы уменьшить возможный риск повреждения, важно периодически сбрасывать содержимое кэша на диск, чтобы в случае сбоя системы терялось только небольшое количество данных. Системный вызов sync указывает MINIX сбросить на диск кэшированные блоки. Обычно при старте MINIX вместе с системой запускается фоновая программа *update*, каждые 30 секунд вызывающая sync.

Для работы с директориями служат еще два вызова, chdir и chroot. Первый из них меняет текущую директорию, а второй меняет корневую. После вызова

```
chdir("/usr/ast/test");
```

процедура открытия файла *xyz* откроет */usr/ast/test/xyz*. Использование понятия рабочего каталога избавляет от необходимости постоянно набирать длинные абсолютные пути файлов.

1.4.5. Системные вызовы для защиты

В MINIX для каждого файла определен 11-битный режимный код файла, иногда также называемый модой (mode), используемый для его защиты. Режимный код включает в себя 9 бит чтения-записи-выполнения для владельцев, группы и других пользователей. Системный вызов chmod предоставляет возможность изменения режимного кода файла. Например, следующий вызов предоставит всем, кроме владельца, доступ к файлу только для чтения; владелец же сможет еще и выполнять файл:

```
chmod("file", 0644);
```

Другие два бита зашиты, 02000 и 04000, называются соответственно битами SETGID (set-group-id, устанавливать идентификатор группы) и SETUID (set-user-id, устанавливать идентификатор пользователя). Когда пользователь запускает программу, у которой установлен бит SETUID, то на время выполнения процесса идентификатор пользователя меняется на идентификатор владельца программы. Эта специальная возможность широко используется для того, чтобы позволить пользователям выполнять функции, доступные только суперпользователям, такие как создание директорий. А именно, для создания директории требуется

вызов mknod, доступный только суперпользователю. Если же владельцем программы *mkdir* будет суперпользователь и для нее будут установлены права доступа 04755, обычные пользователи смогут запускать ее и тем самым вызывать mknod, но весьма ограниченным образом.

Когда процесс исполняет файл, в разрешениях которого выставлены биты SETUID или SETGID, эффективный идентификатор пользователя или группы отличаются от настоящих значений. Но иногда для процесса важно знать, чему равны эффективные и реальные значения uid и gid. Чтобы получить эту информацию, процесс может делать системные вызовы getuid и getgid. Оба этих вызова возвращают одновременно и эффективный, и реальный идентификаторы, а чтобы получать эти значения по отдельности, служат четыре библиотечные процедуры: *getuid*, *getgid*, *geteuid*, *getegid*. Первые две возвращают реальные значения, вторые две — эффективные.

Для обычных пользователей единственный способ изменить свой идентификатор — запустить программу, у которой установлен бит SETUID. Но для суперпользователей существует и другая возможность, предоставляемая системным вызовом setuid, устанавливающим одновременно реальное и эффективное значение uid. Вызов setgid, соответственно, устанавливает реальное и эффективное значение gid. Кроме того, суперпользователь может менять владельца файла при помощи системного вызова chown. Другими словами, у суперпольвователя есть множество возможностей нарушать все возможные правила защиты. Это объясняет, почему многие студенты посвящают столь много времени попыткам стать суперпользователем.

Два последних системных вызова из данной категории могут делаться и обычными процессами. Первый из них, umask, устанавливает системную битовую маску, применяемую для маскирования битов прав доступа к файлу при его создании. Если сделать вызов

```
umask(022);
```

то при вызовах creat или mknod в правах доступа к создаваемому объекту будут маскироваться биты 022. Иначе говоря, вызов

```
creat("file", 0777);
```

создаст файл с правами доступа 0755, а не 0777. Кроме того, битовая маска наследуется дочерними процессами, поэтому, если оболочка сразу после входа сделает вызов umask, ни одна из запущенных пользователем в этой сессии программ не сможет создать файл, который может изменить другой пользователь.

Когда владельцем программы является суперпользователь и у нее установлен бит SETUID, то она может обращаться к любому файлу. Но часто программе нужно знать, имеет ли вызвавший ее пользователь разрешение обращаться к данному файлу. Простая попытка обращения к файлу ничего не даст, так как она всегда завершится успехом.

Следовательно, необходимо иметь возможность узнать, разрешен ли доступ для реального (а не эффективного) uid. Это можно сделать с помощью системного вызова access. Чтобы проверить возможность чтения, параметр *mode* должен быть равен 4, записи — 2 и исполнения — 1. Эти значения можно комбинировать. Например, если mode равен 6, то вызов вернет 0, если разрешены и запись,

и чтение. В противном случае вызов вернет –1. Если параметр *mode* равен 0, то делается проверка того, что файл существует и возможен обзор ведущих к нему директорий.

1.4.6. Системные вызовы для работы со временем

Для работы с часами у MINIX есть четыре системных вызова. Вызов `time` возвращает текущее время в секундах, причем нулем считается полночь 1 января 1970 года (имеется в виду начало дня, а не его конец). Кроме того, чтобы считывать значение системного таймера, нужно иметь возможность сначала это значение установить. Возможность установить часы дает вызов `stime`, доступный только суперпользователю. Третий вызов — `utime`, он позволяет владельцу файла изменить значение времени, записанное в i-узле файла. У этого вызова довольно ограниченная область применения, но некоторым программам он нужен, например, программа *touch* устанавливает время в i-узле файла в соответствие с текущим значением времени.

Наконец, есть вызов `times`, позволяющий узнать, сколько времени процессор провел, исполняя процесс, и сколько времени потрачено на систему (то есть на обработку системных вызовов). Кроме того, суммируется и возвращается общее время системы и процесса для всех дочерних процессов.

1.5. Структура операционной системы

Теперь, когда мы уже видели, как выглядят операционные системы снаружи (то есть мы знакомы с программным интерфейсом), самое время заглянуть внутрь. Чтобы получить представление обо всем спектре возможных вариантов, в следующих разделах мы исследуем четыре различные использующиеся (или использовавшихся ранее) структуры. Исследование это нельзя назвать всесторонним, здесь лишь рассматриваются несколько моделей, применявшихся на практике в разных системах. Их четыре — монолитные системы, многоуровневые системы, виртуальные машины и модель клиент-сервер.

1.5.1. Монолитные системы

В общем случае организация монолитной системы представляет собой «большой беспорядок». То есть структура как таковая отсутствует. Операционная система написана в виде набора процедур, каждая из которых может вызывать другие, когда ей это нужно. При использовании такой техники каждая процедура системы имеет строго определенный интерфейс в терминах параметров и результатов, и каждая имеет возможность вызвать любую другую для выполнения некоторой необходимой для нее работы.

Для построения монолитной системы необходимо скомпилировать все отдельные процедуры, а затем связать их в единый объектный файл с помощью компоновщика. Здесь, по существу, полностью отсутствует сокрытие деталей реализации —

каждая процедура видит любую другую процедуру (в отличие от структуры, содержащей модули, в которых большая часть информации является локальной для модуля, и процедуры модуля можно вызвать только через специально определенные точки входа).

Однако даже такие монолитные системы могут иметь некоторую структуру. При обращении к системным вызовам, поддерживаемым операционной системой, параметры помещаются в строго определенные места — регистры или стек, после чего выполняется специальная команда прерывания, известная как *вызов ядра* или *вызов супервизора*.

Эта команда переключает машину из режима пользователя в режим ядра и передает управление операционной системе, что видно на шаге 1 рис. 1.12 (у большинства процессоров есть два режима работы: режим ядра, предназначенный для ОС, и пользовательский режим, в котором запрещен ввод/вывод и некоторые другие инструкции).

Затем операционная система проверяет параметры вызова, чтобы определить, какой системный вызов должен быть выполнен (шаг 2). После этого операционная система обращается к таблице как к массиву с номером системного вызова в качестве индекса. В k-м элементе таблицы содержится ссылка на процедуру обработки системного вызова k (шаг 3 на рис. 1.12). После того, как работа завершена, управление возвращается в пользовательскую программу, которая продолжит свою работу со следующего оператора (шаг 4).

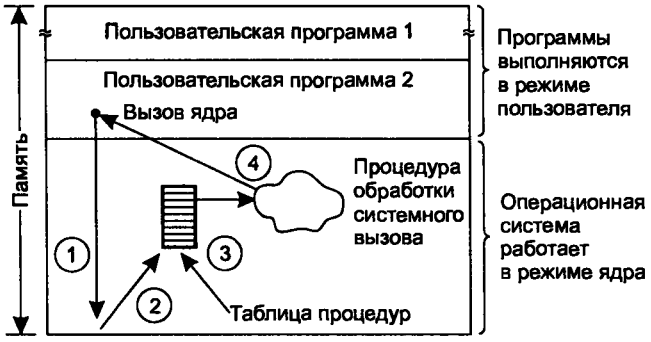


Рис. 1.12. Выполнение системного вызова: *1* — пользовательская программа вызывает прерывание; *2* — операционная система определяет номер процедуры обработчика; *3* — Операционная система вызывает обработчик; *4* — управление возвращается в основную программу

Такая организация операционной системы предполагает следующую структуру:

1. Главная программа, которая вызывает требуемую служебную процедуру.
2. Набор служебных процедур, выполняющих системные вызовы.
3. Набор утилит, обслуживающих служебные процедуры.

В этой модели для каждого системного вызова имеется одна служебная процедура. Утилиты выполняют функции, которые нужны нескольким служебным процедурам. Деление процедур на три уровня показано на рис. 1.13.

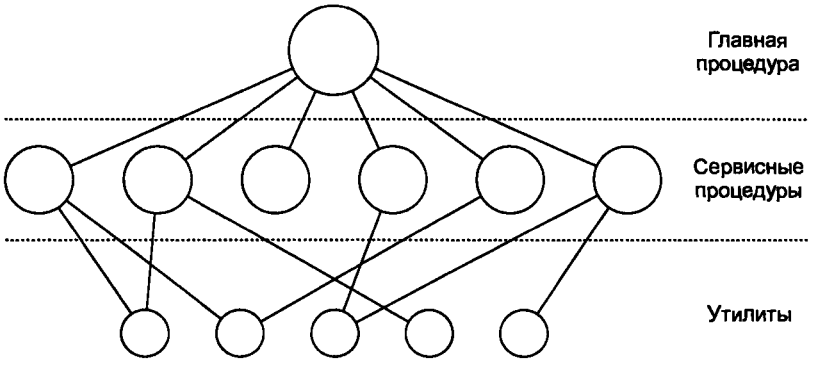


Рис. 1.13. Простая модель монолитной системы

1.5.2. Многоуровневые системы

Обобщением подхода, изображенного на рис. 1.13, является организация операционной системы в виде иерархии уровней. Первой системой, построенной таким образом, была система THE, созданная в Technische Hogeschool Eindhoven (Нидерланды) Э. Дейкстрой (E. W. Dijkstra) и его студентами в 1968 году. Она была простой пакетной системой для голландского компьютера Electrologica X8, память которого состояла из 32 K 27-разрядных слов.

Система включала 6 уровней, как показано в табл. 1.2. Уровень 0 занимался распределением времени процессора, переключая процессы при возникновении прерывания или при срабатывании таймера. Над уровнем 0 система состояла из последовательных процессов, каждый из которых можно было запрограммировать, не заботясь о том, что на одном процессоре запущено несколько процессов. Другими словами, уровень 0 обеспечивал базовую многозадачность процессора.

Таблица 1.2. Структура операционной системы THE

Уровень	Функция
5	Оператор
4	Программы пользователя
3	Управление вводом/выводом
2	Связь оператор-процесс
1	Управление памятью и барабаном
0	Распределение процессора и многозадачность

Уровень 1 управлял памятью. Он выделял процессам пространство в оперативной памяти и на магнитном барабане объемом 512 K слов для тех частей процессов (страниц), которые не помещались в оперативной памяти. Процессы более высоких уровней не заботились о том, находятся ли они в данный момент в памяти или на барабане. Программное обеспечение уровня 1 обеспечивало попадание страниц в оперативную память по мере необходимости.

Уровень 2 управлял связью между консолью оператора и процессами. Таким образом, все процессы выше этого уровня имели свою собственную консоль оператора. Уровень 3 управлял устройствами ввода/вывода и буферизовал потоки информации к ним и от них. Любой процесс выше уровня 3 вместо того, чтобы работать с конкретными устройствами, с их разнообразными особенностями, мог обращаться к абстрактным устройствам ввода/вывода, обладающим удобными для пользователя характеристиками. На уровне 4 работали пользовательские программы, которым не надо было заботиться ни о процессах, ни о памяти, ни о консоли, ни об управлении устройствами ввода/вывода. Процесс системного оператора размещался на уровне 5.

Дальнейшее обобщение многоуровневой концепции было сделано в операционной системе MULTICS. В ней уровни представляли собой серию концентрических колец, где внутренние кольца являлись более привилегированными, чем внешние. Когда процедура внешнего кольца хотела вызвать процедуру кольца, лежащего внутри, она должна была выполнить эквивалент системного вызова, то есть команду TRAP, параметры которой тщательно проверяются перед тем, как выполняется вызов. Хотя операционная система в MULTICS являлась частью адресного пространства каждого пользовательского процесса, аппаратура обеспечивала защиту данных на уровне сегментов памяти, разрешая или запрещая доступ к индивидуальным процедурам (в действительности к сегментам памяти) для записи, чтения или выполнения.

Стоит отметить, что в системе THE многоуровневая схема представляла собой исключительно конструкционное решение и все части системы были, в конечном счете, связаны в один объектный файл, а в MULTICS механизм разделения колец действовал во время исполнения на аппаратном уровне.

Преимущество подхода MULTICS заключается в том, что его можно расширить и на структуру пользовательских подсистем. Например, профессор может написать программу для тестирования и оценки студенческих программ и запустить ее в кольце n, в то время как студенческие программы будут работать в кольце $n + 1$, так что они не смогут изменить свои оценки.

1.5.3. Виртуальные машины

Исходная версия OS/360 была системой исключительно пакетной обработки. Однако множество пользователей OS/360 желали работать в системе с разделением времени, поэтому различные группы программистов как в самой корпорации IBM, так и вне ее решили написать для этой машины системы с разделением времени. Официальная система с разделением времени от IBM, которая называлась TSS/360, поздно вышла в свет и оказалась настолько громоздкой и медленной, что на нее перешли немногие. В конечном счете от нее отказались, но уже после того, как ее разработка потребовала около 50 млн долларов [37]. Группа из научного центра IBM в Кембридже, штат Массачусетс, разработала в корне отличающуюся от нее систему, которую IBM в результате приняла как законченный продукт. Сейчас она широко используется на еще оставшихся мэйнфреймах.

Эта система, в оригинале называвшаяся CP/CMS, а позже переименованная в VM/370, была основана на следующем проницательном наблюдении: система с разделением времени обеспечивает (1) многозадачность и (2) расширенную машину с более удобным интерфейсом, чем тот, что предоставляется оборудованием напрямую. VM/370 основана на полном разделении этих двух функций.

Сердце системы, называемое *монитором виртуальной машины*, работает с оборудованием и обеспечивает многозадачность, предоставляя верхнему слою не одну, а несколько виртуальных машин, как показано на рис. 1.14. Но, в отличие от всех других операционных систем, эти виртуальные машины не являются расширенными. Они не поддерживают файлы и прочие удобства, а представляют собой точные копии голой аппаратуры, включая режимы ядра и пользователя, ввод/вывод данных, прерывания и все остальное, присутствующее на реальном компьютере.

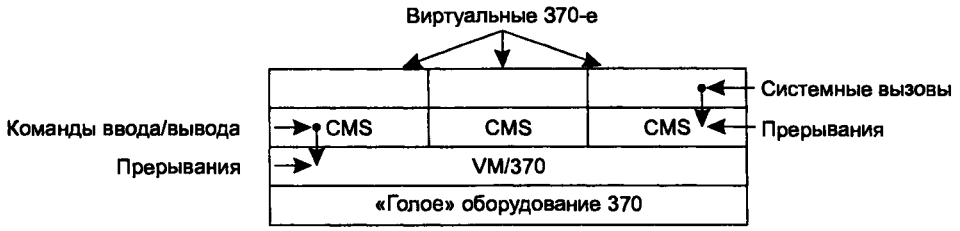


Рис. 1.14. Структура VM/370 с системой CMS

Поскольку каждая виртуальная машина идентична настоящему оборудованию, на каждой из них может работать любая операционная система, которая запускается прямо на аппаратуре. На разных виртуальных машинах могут (а зачастую так и происходит) функционировать различные операционные системы. На некоторых из них для обработки пакетов и транзакций работают потомки OS/360, а на других для интерактивного разделения времени пользователей работает однопользовательская интерактивная система *CMS* (Conversational Monitor System — система диалоговой обработки).

Когда программа операционной системы CMS выполняет системный вызов, он прерывает операционную систему на своей собственной виртуальной машине, а не на VM/370, как произошло бы, если бы он работал на реальной машине вместо виртуальной. Затем CMS выдает обычные команды ввода/вывода для чтения своего виртуального диска или другие команды, которые ей могут понадобиться для выполнения вызова. Эти команды ввода/вывода перехватываются VM/370, которая выполняет их в рамках моделирования реального оборудования. При полном разделении функций многозадачности и предоставления расширенной машины каждая часть может быть намного проще, гибче и удобней для обслуживания.

Идея виртуальной машины очень часто используется в наши дни, но в несколько другом контексте: для работы старых программ, написанных для системы MS-DOS на Pentium (или на других 32-разрядных процессорах Intel). При разработке компьютера Pentium и его программного обеспечения обе компании, Intel и Microsoft, понимали, что возникнет острая потребность в работе старых программ

на новом оборудовании. Поэтому корпорация Intel создала на процессоре Pentium режим виртуального процессора 8086. В этом режиме машина действует как 8086 (которая с точки зрения программного обеспечения идентична 8088), включая 16-разрядную адресацию памяти с ограничением объема памяти в 1 Мбайт.

Такой режим используется системой Windows и другими операционными системами для запуска программ MS-DOS. Программы запускаются в режиме виртуального процессора 8086. Пока они выполняют обычные команды, они работают напрямую с оборудованием. Но когда программа пытается обратиться по прерыванию к операционной системе, чтобы сделать системный вызов, или пытается напрямую осуществить ввод/вывод данных, происходит прерывание с переключением на монитор виртуальной машины.

Возможны два варианта устройства. Первый: сама система MS-DOS загружена в адресное пространство виртуальной машины 8086, так что монитор виртуальной машины только отсылает прерывания назад к MS-DOS, как это происходит на реальной 8086. Когда затем MS-DOS пытается самостоятельно осуществить ввод/вывод, операция перехватывается и выполняется монитором виртуальной машины.

В другом варианте монитор виртуальной машины перехватывает первое прерывание и сам выполняет ввод/вывод, так как он знает все системные вызовы MS-DOS и имеет представление о том, что должно делать каждое прерывание. Этот вариант не столь безупречен, как первый, потому что, в отличие от первого варианта, он корректно моделирует только MS-DOS и никакие другие операционные системы. С другой стороны, он намного быстрее работает, так как избегает проблем запуска MS-DOS для выполнения ввода/вывода. Существует еще один недостаток фактического запуска MS-DOS в режиме виртуальной 8086: MS-DOS очень часто оперирует флагом разрешения/запрещения прерывания, а моделирование этого требует больших затрат.

Стоит отметить, что ни один из двух описанных методов в действительности не является те же самым, чем была VM/370, потому что смоделированная машина представляет собой только 8086, а не полноценный Pentium. В системе VM/370 можно было запустить на виртуальной машине саму VM/370. На Pentium нельзя запустить, скажем, операционную систему Windows на виртуальной 8086, потому что не существует версий Windows, работающих на этой машине. Даже для самых старых версий Windows необходим как минимум 286-й процессор, а моделирование 286 не поддерживается (не говоря уже об эмуляции Pentium).

В системе VM/370 каждый пользователь получает точную копию настоящей машины. На Pentium, в режиме виртуальной машины 8086, каждый пользователь получает точную копию другой машины. Шагнув несколько дальше, исследователи из Массачусетсского технологического института изобрели систему, которая обеспечивает каждого пользователя абсолютной копией реального компьютера, но с подмножеством ресурсов [30]. Например, одна виртуальная машина может получить блоки на диске с номерами от 0 до 1023, следующая — от 1024 до 2047 и т. д.

На нижнем уровне в режиме ядра работает программа, которая называется *экзоядро* (exokernel). В ее задачу входит распределение ресурсов для виртуальных

машин, а после этого проверка их использования (то есть отслеживание попыток машин использовать чужой ресурс). Каждая виртуальная машина на уровне пользователя может работать с собственной операционной системой, как на VM/370 или виртуальных 8086-х для Pentium, с той разницей, что каждая машина ограничена набором ресурсов, которые она запросила и которые ей были предоставлены.

Преимущество схемы экзоядра заключается в том, что она позволяет обойтись без уровня отображения. При других методах работы каждая виртуальная машина считает, что она использует свой собственный диск с нумерацией блоков от 0 до некоторого максимума. Поэтому монитор виртуальной машины должен поддерживать таблицы преобразования адресов на диске (и всех других ресурсов). Необходимость преобразования отпадает при наличии экзоядра, которому нужно только хранить запись о том, какой виртуальной машине выделен данный ресурс. Такой подход имеет еще одно преимущество: он отделяет многозадачность (в экзоядре) от операционной системы пользователя (в пространстве пользователя) с меньшими затратами, так как для этого ему необходимо всего лишь не допускать вмешательства одной виртуальной машины в работу другой.

1.5.4. Модель клиент-сервер

Система VM/370 сильно выигрывает в простоте благодаря переносу значительной части кода традиционной операционной системы (обеспечивающего расширенную машину) в верхний уровень, систему CMS. Однако VM/370 и при этом останется сложной комплексной программой, потому что моделирование нескольких виртуальных 370-х машин само по себе не так просто (особенно если вы хотите сделать это достаточно эффективно).

В развитии современных операционных систем наблюдается тенденция в сторону дальнейшего переноса кода в верхние уровни и удалении при этом всего, что только возможно, из режима ядра, оставляя минимальное *микроядро*. Обычно это осуществляется перекладыванием выполнения большинства задач операционной системы на средства пользовательских процессов. Получая запрос на какую-либо операцию, например чтение блока файла, пользовательский процесс (теперь называемый *обслуживаемым процессом* или *клиентским процессом*) посылает запрос *серверному* (обслуживающему) *процессу*, который его обрабатывает и высылает назад ответ.

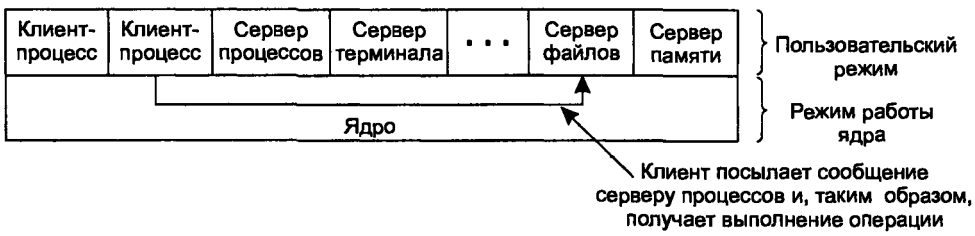


Рис. 1.15. Модель клиент-сервер

В модели, показанной на рис. 1.15, в задачу ядра входит только управление связью между клиентами и серверами. Благодаря разделению операционной системы на части, каждая из которых управляет всего одним элементом системы (файловой системой, процессами, терминалом или памятью), все части становятся маленькими и управляемыми. К тому же, поскольку все серверы работают как процессы в режиме пользователя, а не в режиме ядра, они не имеют прямого доступа к оборудованию. Поэтому если происходит ошибка на файловом сервере, может разрушиться служба обработки файловых запросов, но это обычно не приводит к остановке всей машины целиком.

Другое преимущество модели клиент-сервер заключается в ее простой адаптации к использованию в распределенных системах (рис. 1.16). Если клиент общается с сервером, посылая ему сообщения, клиенту не нужно знать, обрабатывается ли его сообщение локально на его собственной машине, или оно было послано по сети серверу на удаленной машине. С точки зрения клиента происходит одно и то же в обоих случаях: запрос был послан и на него получен ответ.

Рассказанная выше история о ядре, управляющем передачей сообщений от клиентов к серверам и назад, не совсем реалистична. Некоторые функции операционной системы, такие как загрузка команд в регистры физических устройств ввода/вывода, трудно, если вообще возможно, выполнить из программ в пространстве пользователя. Есть два способа разрешения этой проблемы. Первый заключается в том, что некоторые критические серверные процессы (например, драйверы устройств ввода/вывода) действительно запускаются в режиме ядра, с полным доступом к аппаратуре, но при этом общаются с другими процессами при помощи обычной схемы передачи сообщений.

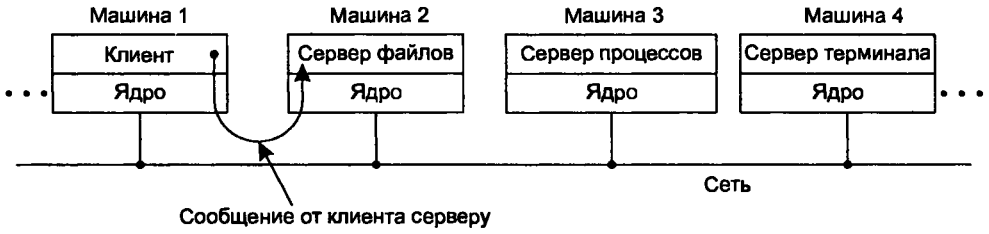


Рис. 1.16. Модель клиент-сервер в распределенной системе

Второй способ состоит в том, чтобы встроить минимальный *механизм обработки информации* в ядро, но оставить принятие *политических* решений за серверами в пользовательском пространстве [53]. Например, ядро может опознавать сообщения, посланные по определенным адресам. Для ядра это означает, что нужно взять содержимое сообщения и загрузить его, скажем, в регистры ввода/вывода некоторого диска для запуска операции чтения диска. В этом примере ядро даже может не обследовать байты сообщения, если они оказались допустимы или осмысленны; оно может вслепую копировать их в регистры диска. (Очевидно, должна использоваться некоторая схема, ограничивающая круг процессов, имеющих право отправлять подобные сообщения). Разделение между

механизмом и политикой является очень важным понятием, встречающимся в операционных системах в различном контексте постоянно.

1.6. Краткий обзор следующих глав

Типичные операционные системы состоят из четырех основных компонентов: управление процессами, устройствами ввода/вывода, памятью и файлами. Следующие четыре главы будут посвящены этим четырем составным частям, по одной главе на часть. В главе 6 содержится перечень рекомендуемой литературы и библиографии.

Главы о процессах, вводе/выводе, управлении памятью и файловых системах имеют одну и ту же общую структуру. Сначала излагаются общие принципы. Затем идет обзор соответствующей секции MINIX (эта информация применима и к UNIX). Наконец, подробно рассматривается реализация в MINIX. Часть, посвященную реализации, можно без потери целостности изложения пропустить или коснуться ее вскользь. Те же читатели, которые заинтересованы в изучении работы реальной ОС, должны прочитать все части.

Резюме

Операционную систему можно рассматривать с двух точек зрения: как менеджер ресурсов и как расширенную машину. Как менеджер ресурсов операционная система рационально управляет различными частями системы. С точки зрения расширенной машины, работа операционной системы состоит в предоставлении пользователям виртуальной машины, более удобной, чем настоящий компьютер.

Операционные системы имеют достаточно долгую историю развития, которая начинается с тех дней, когда операционные системы заменили оператора, и продолжается до современных многозадачных систем. Большое значение имеют ранние системы пакетной обработки, многозадачные системы и системы для персональных компьютеров.

Сердцем любой операционной системы является набор системных вызовов, которые она может обработать. Они говорят о том, что реально делает операционная система. Мы рассмотрели шесть групп системных вызовов для MINIX. Первая группа работает с созданием и завершением процессов. Вторая группа предназначена для работы с сигналами. Третья — для чтения и записи файлов. Четвертая нужна для управления каталогами. Пятая включила в себя управление защитой, а шестая — работу со временем.

Операционная система может быть структурирована несколькими способами. Наиболее общими выделяемыми при структурировании понятиями являются: монолитные системы, иерархия слоев, система виртуальных машин, экзоядро или использование модели клиент-сервер.

Вопросы

1. Каковы две главные функции операционной системы?
2. Что такое многозадачность?
3. Что такое подкачка данных (spooling)? Как вы считаете, будут ли передовые персональные компьютеры иметь в будущем подкачку данных в качестве стандартного элемента?
4. На ранних компьютерах чтение или запись каждого байта данных управлялось напрямую центральным процессором (то есть тогда не было прямого доступа к памяти — DMA). Какой смысл имеет это понятие для многозадачности?
5. Почему системы с разделением времени не были широко распространены на компьютерах второго поколения?
6. Какая из следующих команд должна быть разрешена только в режиме ядра:
   1) отключение всех прерываний;
   2) чтение счетчика даты/времени;
   3) изменения счетчика даты/времени;
   4) изменение схемы распределения памяти.
7. Перечислите основные различия между операционной системой для персонального компьютера и для мэйнфрейма.
8. У файла в MINIX идентификатор владельца равен 12 и идентификатор группы равен 1. Файлу присвоены следующие разрешения: *rwxr-x---*. К этому файлу пытается обратиться другой пользователь, у которого uid = = 6, а gid = 1. Что произойдет?
9. Как в свете того, что само существование суперпользователю может привести ко множеству проблем с безопасностью, объяснить существование этой концепции?
10. Модель клиент-сервер популярна в распределенных системах. Может ли она использоваться в системах из одного компьютера?
11. Почему в системах разделения времени необходима таблица процессов? Нужна ли она также в системах на персональных компьютерах, где существует только один процесс, и этот процесс завладевает всей машиной до тех пор, пока не завершится?
12. В чем заключается существенная разница между блоковым специальным файлом и символьным специальным файлом?
13. Что случится, если в MINIX пользователь 2 создаст ссылку на файл, которым владеет пользователь 1, затем пользователь 1 удалит файл, и, наконец, пользователь 2 попытается прочитать файл?
14. Почему системный вызов `chroot` разрешено выполнять только суперпользователю (подсказка: подумайте о проблемах безопасности)?
15. Зачем в MINIX все время работает фоновая программа *update*?

16. Имеет ли смысл игнорировать сигнал `sigalarm`?
17. Напишите программу (или набор программ), чтобы протестировать все системные вызовы MINIX. Произведите каждый вызов с разными параметрами, в том числе и с некорректными, чтобы увидеть реакцию системы на ошибки.
18. Напишите оболочку, подобную той, что показана в листинге 1.1, но достаточно полную, чтобы ее можно было протестировать. Можно добавить некоторые дополнительные возможности, например перенаправление ввода и вывода, создание каналов, запуск фоновых задач.

Глава 2
Процессы и нити

Теперь мы детально рассмотрим устройство и работу операционных систем, в частности MINIX. Основным понятием, связанным с операционными системами, является *процесс* — абстрактное понятие, описывающее работу программы. Все остальное базируется на нем, поэтому представляется крайне важным, чтобы разработчики операционных систем (а также студенты) получили полное представление о концепции процесса как можно раньше.

2.1. Процессы

Все современные компьютеры могут делать одновременно несколько дел. Например, одновременно с запущенной пользователем программой может выполняться чтение с диска и вывод текста на экран монитора или на принтер. В многозадачной системе процессор переключается между программами, предоставляя каждой от десятков до сотен миллисекунд персонального времени. При этом в каждый конкретный момент времени процессор занят только одной программой, но за секунду он успевает поработать с несколькими, создавая у пользователей иллюзию параллельной работы со всеми программами. Иногда в этом контексте говорят о *псевдопараллелизме*, в отличие от настоящего параллелизма в *многопроцессорных системах* (в которых установлено два и более процессора, разделяющих между собой общую физическую память). Следить за работой параллельно протекающих процессов достаточно трудно, поэтому со временем разработчики операционных систем разработали концептуальную модель логически упорядоченных процессов, упрощающую эту работу. В данной главе мы опишем применение этой модели, а также некоторые результаты ее применения.

2.1.1. Модель процесса

В этой модели все функционирующее на компьютере программное обеспечение, иногда включая собственно операционную систему, организовано в виде набора *логически упорядоченных процессов*, или, для краткости, просто *процессов*. Процессом является выполняемая программа, вместе с текущими значениями счетчика команд, регистров и переменных. С позиций данной абстрактной модели, у каждого процесса есть собственный виртуальный центральный процессор. На самом деле, разумеется, реальный процессор переключается с процесса на процесс, но для лучшего понимания системы значительно проще рассматривать набор процессов, выполняемых параллельно (псевдопараллельно), чем пытаться представить себе процессор, переключающийся от программы к программе. Как

мы уже знаем из первой главы, это переключение и называется *многозадачностью* или *мультипрограммированием*.

На рис. 2.1, *а* представлена схема компьютера, работающего с четырьмя программами. На рис. 2.1, *б* показаны четыре процесса, каждый со своей управляющей логикой (то есть логическим счетчиком команд), независимые друг от друга. Конечно, на самом деле существует только один физический счетчик команд, подменяемый логическим счетчиком команд текущего процесса. Когда время, отведенное текущему процессу, заканчивается, значение физического счетчика сохраняется в логическом счетчике команд, то есть в памяти процесса. На рис. 2.1, *в* видно, что за достаточно большой промежуток времени изменилось состояние всех четырех процессов, но в каждый конкретный момент в действительности работает только один процесс.

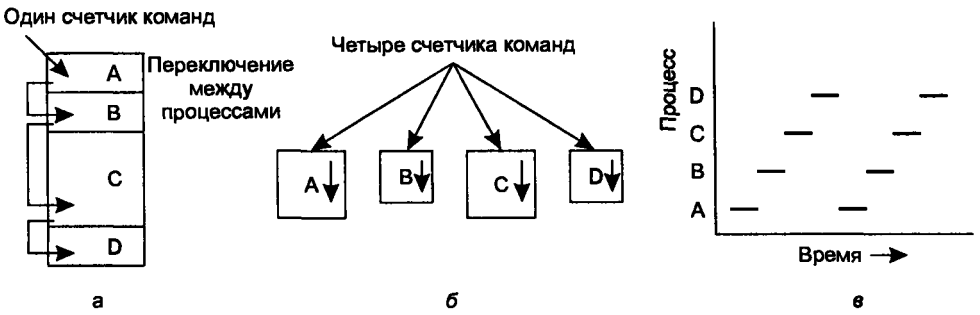


Рис. 2.1. Схема работы с четырьмя программами: *а* — четыре программы в многозадачном режиме; *б* — принципиальная модель четырех независимых логически упорядоченных процессов; *в* — в каждый момент времени активна только одна программа

Поскольку процессор переключается между программами, скорость, с которой процессор производит свои вычисления, будет непостоянной и, возможно, даже иной при каждом новом запуске процесса. Поэтому не следует программировать процессы исходя из каких-либо жестко заданных временных предположений. Представьте себе, например, процесс ввода/вывода, запускающий накопитель на магнитной ленте для восстановления упакованных файлов. Процесс выполняет холостой цикл задержки из 10 000 итераций, чтобы дать время накопителю разогнаться, а затем дает команду считать первый сектор. Если во время холостого цикла процессор решит переключиться на другую задачу, может случиться так, что работающий с магнитофоном процесс запустится снова уже после того, как считывающая головка пройдет первую запись. Если у процесса есть критические временные рамки такого рода, то есть отдельные события должны укладываться в заданное количество миллисекунд, необходимы специальные меры, позволяющие удостовериться в завершенности события. Однако обычно многозадачный режим, а также относительные скорости различных процессов не влияют на работу большинства отдельных процессов.

Различие между процессом и программой трудноуловимо, но тем не менее имеет принципиальное значение. Воспользуемся следующей аналогией: пред-

ставьте себе программиста, имеющего познания в кулинарии и своими руками выпекающего торт на день рождения собственной дочери. В его распоряжении есть рецепт торта, кухня, оборудованная всем необходимым, и ингредиенты для торта: мука, яйца, сахар, ванилин и т. п. Согласно этой аналогии, рецепт — это программа (то есть алгоритм), программист исполняет роль процессора, а ингредиенты торта представляют собой входные данные. Процессом является следующая последовательность действий: программист читает рецепт, смешивает продукты и печет торт.

Теперь представьте, что на кухню прибегает плачущий сын программиста и кричит, что его ужалила пчела. Программист отмечает, на чем он остановился (сохраняет текущее состояние процесса), находит справочник по оказанию первой помощи и действует в соответствии с инструкцией. Таким образом, наш процессор переключился с одного процесса (выпечка торта) на другой, с большим приоритетом (оказание первой помощи), и каждый процесс связан со своей программой (рецепт торта и справочник по оказанию первой помощи). После проведения всех необходимых процедур по борьбе с укусом пчелы программист возвращается к торту, продолжая с той операции, на которой он прервался.

Мы привели такую аналогию с целью показать, что процесс — это активность некоторого рода. У него есть программа, входные и выходные данные, а также состояние. Один процессор может переключаться между различными процессами, опираясь на некий алгоритм планирования для определения момента переключения от одного процесса к другому.

Иерархия процессов

Операционной системе нужен способ, позволяющий удостовериться в наличии всех обслуживаемых процессов. В простейших системах, а также системах, разработанных для выполнения одного-единственного приложения (например контроллер микроволновой печи), нетрудно реализовать такую ситуацию, в которой все процессы, которые когда-либо могут понадобиться, уже присутствуют в системе при ее загрузке. В универсальных системах требуется способ создания и прерывания процессов по мере необходимости. В MINIX процессы создаются при помощи системного вызова fork, создающего точную копию вызвавшего процесса. Дочерние процессы также могут вызывать fork, таким образом, образуется целое дерево процессов. В других операционных системах также существуют системные вызовы для создания процесса, загрузки его в память и запуска. Какова бы ни была природа системного вызова, процессам нужен способ запуска других процессов. Обратите внимание на то, что у каждого процесса есть всего один родительский, но может быть сколько угодно дочерних процессов (в том числе они могут вообще отсутствовать).

Как простой пример использования дерева процессов, мы рассмотрим инициализацию MINIX после ее загрузки. В загрузочном образе имеется специальный процесс под именем init. Когда этот процесс запускается, он сначала узнает из файла конфигурации, сколько имеется терминалов. Затем на каждый терминал при помощи fork создается новый процесс. Каждый из этих процессов ожидает начала сеанса работы пользователя, и, если пользователь успешно вошел

в систему, для него запускается оболочка, способная выполнять пользовательские команды. Эти команды могут, в свою очередь, запускать новые процессы и т. д. Таким образом, все процессы в системе образуют единое дерево, корнем которого является процесс init.

Состояния процессов

Несмотря на то что процесс является независимым объектом, со своим счетчиком команд и внутренним состоянием, существует необходимость его взаимодействия с другими процессами. Например, выходные данные одного процесса могут служить входными данными для другого процесса. В команде оболочки

```
cat chapter1 chapter2 chapter3 | grep tree
```

первый процесс, исполняющий файл cat, объединяет и распечатывает три файла. Второй процесс, исполняющий файл grep, отбирает все строки, содержащие слово «tree». В зависимости от относительных скоростей процессов (скорости зависят от относительной сложности программ и процессорного времени, предоставляемого каждому процессу) может получиться, что grep уже готов к запуску, но входных данных для его процесса еще нет. В этом случае процесс *блокируется* до поступления входных данных.

Процесс блокируется, поскольку с точки зрения логики он не в состоянии продолжать свою работу (обычно это связано с отсутствием входных данных, ожидаемых процессом). Также возможна ситуация, когда активный и работоспособный процесс останавливается, так как операционная система решила предоставить на время процессор другому процессу. Эти ситуации принципиально различны. В первом случае приостановка выполнения является внутренней проблемой (поскольку невозможно обработать командную строку пользователя до того, как она была введена). Во втором случае проблема является технической (нехватка процессоров для каждого процесса). На рис. 2.2 представлена схема состояний, показывающая три возможных состояния процесса:

1. Активный (в этот конкретный момент использующий процессор).
2. Готовый к работе (процесс временно приостановлен, чтобы позволить выполняться другому процессу).
3. Заблокированный (процесс не может быть запущен прежде, чем произойдет некое внешнее событие).

С позиций логики первые два состояния одинаковы. В обоих вариантах процесс может быть запущен, только во втором случае недоступен процессор. Третье состояние отличается тем, что запустить процесс невозможно, какой бы ни была загруженность процессора.

Как показано на рисунке, между этими тремя состояниями допустимы четыре перехода. Переход 1 происходит, когда процесс обнаруживает, что продолжение работы невозможно. В некоторых системах процесс должен выполнить системный запрос, например block, чтобы оказаться в состоянии блокировки. В других системах, как в MINIX, процесс автоматически блокируется, если при считыва-

нии из канала или специального файла (предположим, терминала) входные данные не были обнаружены.

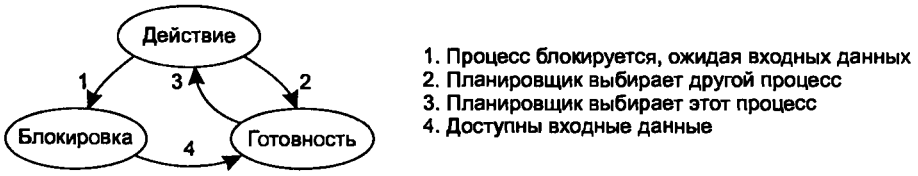


Рис. 2.2. Процесс может находиться в рабочем, готовом и заблокированном состоянии. Стрелками показаны возможные переходы между состояниями

Переходы 2 и 3 вызываются частью операционной системы, называемой планировщиком процессов, так что сами процессы даже не знают о существовании этих переходов. Переход 2 происходит, если планировщик решил, что пора предоставить процессор следующему процессу. Переход 3 имеет место, когда все остальные процессы уже исчерпали предоставленное им время и процессор снова возвращается к первому процессу. Вопрос планирования (когда следует запустить очередной процесс и на какое время) сам по себе достаточно важен, и мы вернемся к нему позже в этой главе. Было разработано множество алгоритмов с целью сбалансировать требования по эффективности для системы в целом и для каждого процесса в отдельности. Мы также рассмотрим некоторые из них ниже в этой главе.

Переход 4 происходит с появлением внешнего события, ожидавшегося процессом (например поступления входных данных). Если в этот момент не запущен какой-либо другой процесс, то срабатывает переход 3 и процесс запускается. В противном случае процессу придется некоторое время находиться в состоянии готовности, пока не освободится процессор.

Модель процессов упрощает представление о внутреннем поведении системы. Некоторые процессы запускают программы, выполняющие команды, введенные с клавиатуры пользователем. Другие процессы являются частью системы и обслуживают такие задачи, как выполнение запросов файловой подсистемы, управление запуском диска или магнитного накопителя. В случае дискового прерывания система останавливает текущий процесс и запускает дисковый процесс, который был заблокирован в ожидании этого прерывания. Вместо прерываний мы можем представлять себе дисковые процессы, процессы пользователя, терминала и т. п., блокирующиеся на время ожидания событий. Когда событие произошло (информация прочитана с диска или клавиатуры), блокировка снимается и процесс вправе быть запущен.

Рассмотренный подход описывается моделью, представленной на рис. 2.3. Нижний уровень операционной системы — это планировщик, выше него — все множество процессов. За обработку прерываний и детали, связанные с остановкой и запуском процессов, отвечает то, что мы назвали планировщиком, который является, по сути, совсем небольшой программой. Вся остальная часть операци-

онной системы упорядочена в виде набора процессов. Очень немногие существующие системы структурированы столь удобно.

Рис. 2.3. Нижний уровень операционной системы отвечает за прерывания и планирование. Выше расположены логически упорядоченные процессы

2.1.2. Реализация процессов

Для реализации модели процессов операционная система поддерживает таблицу (массив структур), называемую *таблицей процессов*, с одним элементом для каждого процесса. (Некоторые авторы называют эти элементы *блоками управления процессом*.) Элемент таблицы содержит информацию о состоянии процесса, счетчике команд, указателе стека, распределении памяти, состоянии открытых файлов, об использовании и распределении ресурсов, а также всю остальную информацию, которую необходимо сохранять при переключении в состояние *готовности* или *блокировки* для последующего запуска, — как если бы процесс не останавливался.

В MINIX за управление процессами, памятью и файлами ответственны различные модули из состава системы, поэтому таблица разбита на разделы, где каждый модуль поддерживает относящиеся к нему поля. В табл. 2.1 представлены некоторые наиболее важные поля типичной системы. Поля в первой колонке относятся к управлению процессом. Остальные колонки описывают управление памятью и файлами.

Теперь, после знакомства с таблицей процессов, время сказать несколько слов о том, как создается иллюзия параллельности процессов на машине с одним процессором и несколькими устройствами ввода/вывода. Затем последует описание того, как работает планировщик в MINIX (см. рис. 2.3), но это же относится и к большинству других современных ОС. С каждым классом устройств ввода/вывода (гибкий диск, жесткий диск, таймер, терминал) связана область памяти (обычно расположенная в нижних адресах), называемая *вектором прерываний*. Вектор прерываний содержит адрес процедуры обработки прерываний. Представьте, что в момент дискового прерывания работал пользовательский процесс 3. Содержимое счетчика команд процесса, слово состояния программы и, возможно, один или несколько регистров записываются в (текущий) стек аппаратными средствами. Затем происходит переход по адресу, указанному в векторе прерывания диска. Вот и все, что делают аппаратные средства. С этого момента вся остальная обработка прерывания производится программным обеспечением, обычно стандартной процедурой-обработчиком.

Таблица 2.1. Некоторые поля типового элемента таблицы процессов

Управление процессом	Управление памятью	Управление файлами
Регистры	Указатель на текстовый сегмент	Маска UMASK
✦ Счетчик команд	✦ Указатель на сегмент данных	✦ Корневой каталог
✦ Слово состояния программы	✦ Указатель на сегмент стека	✦ Рабочий каталог
✦ Указатель стека	✦ Статус завершения	✦ Дескрипторы файла
✦ Состояние процесса	✦ Состояние сигналов	✦ Эффективный идентификатор пользователя
✦ Приоритет	✦ Идентификатор процесса	✦ Эффективный идентификатор группы
✦ Использованное процессорное время	✦ Родительский процесс	✦ Параметры системного вызова
✦ Процессорное время дочернего процесса	✦ Реальный UID	✦ Различные битовые флаги
✦ Время следующего аварийного сигнала	✦ Эффективный GID	
✦ Указатели очереди сообщений	✦ Реальный GID	
✦ Биты действующих сигналов	✦ Эффективный GID	
✦ Идентификатор процесса	✦ Битовые маски для сигналов	
✦ Различные флаги	✦ Различные битовые флаги	

Обработка любого прерывания начинается с сохранения регистров, часто в блоке управления текущим процессом в таблице процессов. Затем информация, помещенная в стек прерыванием, удаляется, и указатель стека переставляется на временный стек, используемый программой обработки процесса. Такие действия, как сохранение регистров и установка указателя стека, невозможно даже выразить на языке высокого уровня (например, на C). Поэтому они выполняются небольшой программой на ассемблере, обычно одинаковой для всех прерываний, поскольку процедура сохранения регистров не зависит от причины возникновения прерывания.

Взаимодействие между процессами в MINIX организуется при помощи сообщений, таким образом, следующий шаг — формирование уведомления дисковому процессу, который ожидает информации от системы в заблокированном состоянии. В сообщении указывается, что произошло именно прерывание, чтобы его можно было отличить от сообщений других пользовательских процессов, запрашивающих чтение дисковых блоков и т. п. После этого с процесса снимается блокировка и делается вызов планировщика. В MINIX у процессов могут быть разные приоритеты, с целью дать обработчикам ввода/вывода преимущество перед обычными пользовательскими программами. Соответственно, если в данный момент у дискового процесса наибольший приоритет, для запуска будет выбран этот процесс. Если же приоритет прерванного процесса не меньше, то будет запущен он, а дисковому придется немного подождать.

По завершении своей работы эта программа вызывает процедуру на языке C, которая выполняет все остальные действия, связанные с конкретным прерыва-

нием. (Мы предполагаем, что операционная система написана на С, что является стандартным решением для всех существующих ОС.) Схема обработки прерывания нижним уровнем операционной системы и действия планировщика представлены ниже. Следует отметить, что частности могут несколько варьироваться от системы к системе.

1. Аппаратное обеспечение сохраняет в стеке счетчик команд и т. п.
2. Аппаратное обеспечение загружает новый счетчик команд из вектора прерываний.
3. Процедура на ассемблере сохраняет регистры.
4. Процедура на ассемблере устанавливает новый стек.
5. Запускается программа обработки прерываний на С. (Она обычно считывает и буферизует входные данные.)
6. Планировщик выбирает следующий процесс.
7. Программа на С передает управление процедуре на ассемблере.
8. Процедура на ассемблере запускает новый процесс.

2.1.3. Нити

В традиционных процессах, о которых мы только что говорили, есть только один поток управления и один счетчик команд. Но в некоторых современных операционных системах существует возможность создать несколько потоков управления в одном процессе. Такие потоки обычно называются просто *нитями* или, иногда, *легковесными процессами.*

На рис. 2.4, *а* представлены три обычных процесса, у каждого из которых есть собственное адресное пространство и единственный поток управления. На рис. 2.4, *б* представлен один процесс с тремя потоками управления. В обоих случаях присутствуют три нити, но на рис. 2.4, *а* каждая из них имеет собственное адресное пространство, а на рис. 2.4, *б* нити разделяют память вместе.

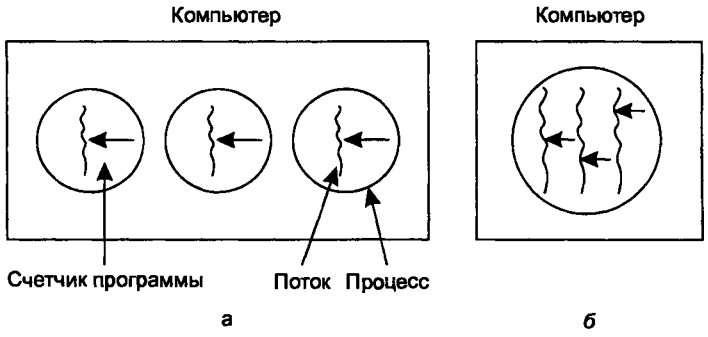


Рис. 2.4. *а* — три процесса с одиночными потоками управления; *б* — один процесс с тремя потоками управления

В качестве примера приложения, рассчитанного на многопоточность, рассмотрим файловый сервер. Процесс сервера получает запрос на чтение или запись файла, а также отсылает считанные данные или, наоборот, принимает данные для записи. Чтобы увеличить производительность, сервер поддерживает в памяти кэш последних обращений к файлам и, по возможности, пытается работать с ним.

Такая ситуация естественным образом ведет к рис. 2.4, *б*. Поступивший запрос обрабатывается отдельной нитью. Если эта нить по той или иной причине блокируется (например из-за обращения к диску), другие нити могут продолжать работу. Изображенная на рис. 2.4, *а* модель не подходит, так как здесь отдельные нити не в силах использовать общий кэш в памяти.

Другой пример приложения, в котором необходимы потоки управления, — это браузеры, наподобие Netscape или Internet Explorer. Многие веб-страницы оформлены с помощью большого числа маленьких изображений. Чтобы загрузить каждое из них, браузеру необходимо каждый раз создавать новое соединение, и львиная доля времени уходит на процесс установки и завершения этих соединений. Если же один браузер будет использовать несколько потоков, он сможет запрашивать несколько картинок одновременно, что иногда приводит к заметному увеличению скорости, так как для небольших изображений время собственно передачи данных невелико.

Когда в памяти имеется несколько нитей, то некоторые из полей, показанных в табл. 2.1, относятся уже не к процессам, а к отдельным нитям. Поэтому необходима дополнительная таблица, каждая запись в которой будет описывать отдельный поток управления. Среди прочих в этой таблице должны быть такие поля, как счетчик команд, регистры и состояние потока. Счетчик команд нужен потому, что нити, как и процессы, могут приостанавливаться и возобновлять свою работу. Поля регистров необходимы по той причине, что значения регистров приостановленной нити необходимо сохранять. Наконец, нити, как и процессы, могут находиться в состоянии выполнения, готовности, а также быть заблокированными.

Иногда операционная система не заботится о потоках управления. Другими словами, управление нитями происходит целиком в пользовательском режиме. Например, когда нить блокируется, то, перед тем как остановиться, она решает, какая нить получит управление дальше, и запускает ее. Существуют и широко используются несколько библиотек для поддержки пользовательских нитей, в том числе пакеты POSIX P-Threads и Mach C-Threads.

В других системах ОС учитывает существование множества потоков, и, когда одна нить переходит в состояние блокировки, система выбирает следующую, которая получит управление, в том же самом процессе или в другом. Чтобы поддерживать такую функциональность, ядро системы должно хранить таблицу всех нитей в системе, наподобие таблицы процессов.

Хотя, на первый взгляд, оба варианта могут показаться равносильными, между ними есть заметная разница в производительности. Переключение потоков происходит гораздо быстрее, когда оно делается без участия ядра. Этот факт — серьезный аргумент за пользовательские нити. С другой стороны, когда одна из

пользовательских нитей блокируется системой (например, она ожидает завершения операции ввода/вывода или должна обработать ошибку отсутствия страницы), то блокируется весь процесс, поскольку ядро не подозревает о существовании нескольких последовательностей команд. Это — сильный аргумент за потоки, управляемые ядром. Как следствие, используются обе системы, и существует множество гибридных схем.

Несмотря на то что нити часто бывают полезными, они существенно усложняют программную модель. Представьте себе системный вызов fork в UNIX. Если родительский процесс состоял из множества нитей, должно ли это свойство распространяться на дочерний процесс? Если нет, то ожидаемо неправильное функционирование процесса, поскольку все нити могут оказаться необходимы.

Но что произойдет, если поток родительского процесса будет блокирован вызовом read с клавиатуры, а у дочернего процесса столько же нитей, сколько у родительского? Будут ли теперь блокированы две нити — одна из родительского процесса, другая из дочернего? И если с клавиатуры поступит строка, получат ли обе нити ее копию? Или только одна — тогда какая? Эта же проблема возникает при работе с открытыми сетевыми соединениями.

Другой класс проблем связан с тем, что нити совместно используют большое количество структур данных. Что произойдет, если одна нить закроет файл в то время, когда другая считывает из него данные? Представьте себе, что одной нити стало недостаточно памяти, и она просит выделить дополнительную. На полпути происходит переключение потоков, и теперь новая нить также замечает, что ей не хватает памяти и запрашивает ее для себя. В этой ситуации память может быть выделена дважды.

Еще одна проблема связана с сообщениями об ошибках. В UNIX, после системного вызова, система помещает информацию о состоянии операции в глобальную переменную, errno. А что произойдет, если сразу после того, как первая нить сделала системный вызов, управление было передано другой, которая также сделает системный вызов и затрет значение глобальной переменной?

Теперь рассмотрим сигналы. Одни из них замыкаются на нити, тогда как другие — нет. Например, если нить выполняет запрос alarm, результирующий сигнал по логике должен вернуться к этой нити. Однако когда нити реализованы в пространстве пользователя, ядро ничего не знает об их существовании и вряд ли направит сигнал по правильному назначению. Ситуация еще больше усложняется, если у процесса может быть только один необработанный аварийный сигнал, а несколько нитей выполняют запрос alarm независимо друг от друга.

Другие сигналы, такие как прерывание с клавиатуры, не связаны с нитями. Кто должен их перехватывать? Одна назначенная нить? Все нити? Специально созданная «всплывающая» нить? Что случится, если одна нить изменит обработчик сигнала, не предупредив об этом остальные нити?

Последняя проблема, порождаемая нитями, — управление стеками. Во многих системах при переполнении стека процесса ядро автоматически увеличивает его. Если у процесса несколько нитей, стеков тоже должно быть несколько. Если ядро не знает о существовании этих стеков, оно не может их автоматически

наращивать при переполнении. Ядро может даже не связать ошибки памяти с переполнением стеков.

Разумеется, эти проблемы не являются непреодолимыми, но на их примере хорошо видно, что введение нитей в существующую систему неразумно без тщательной и продуманной реконструкции всей системы. По крайней мере, придется изменить семантику системных запросов и переписать библиотеки. И результат ваших трудов должен быть совместим с существующими программами для процессов с одним потоком управления. Дополнительную информацию о нитях см. в [40, 59].

2.2. Межпроцессное взаимодействие

Процессам часто бывает необходимо взаимодействовать между собой. Например, в конвейере ядра выходные данные первого процесса должны передаваться второму и т. д. по цепочке. Поэтому одной из главных задач становится правильно организованное взаимодействие между процессами, по возможности не использующее прерываний. В этом разделе мы рассмотрим некоторые аспекты *межпроцессного взаимодействия* (IPC, InterProcess Communication).

Говоря кратко, проблема разбивается на три компонента. Первый мы уже упомянули: передача информации от одного процесса к другому. Второй связан с контролем над деятельностью процессов: как гарантировать, что два процесса не «пересекутся» в критических ситуациях (представьте себе два процесса, каждый из которых пытается завладеть последним мегабайтом памяти). Третий касается согласования действий процессов: если процесс `A` отвечает за поставку данных, а процесс `B` за их вывод на печать, то процесс `B` должен подождать и не начинать печатать, пока не поступят данные от процесса `A`. Все три вопроса будут нами разобраны в следующем подразделе.

2.2.1. Состояние состязания

В некоторых операционных системах процессы, работающие совместно, сообща используют некое общее хранилище данных. Каждый из процессов может считывать что-либо из общего хранилища данных и записывать туда информацию. Это хранилище представляет собой участок в основной памяти (возможно, в структуре данных ядра) или файл общего доступа. Местоположение разделяемой памяти не влияет на суть взаимодействия и возникающие проблемы. Рассмотрим межпроцессное взаимодействие на простом, но очень распространенном примере: систему буферизации — «спулер» — печати. Если процессу требуется вывести на печать файл, он помещает имя файла в специальный *каталог спулера*. Другой процесс, *демон печати*, периодически проверяет наличие поданных на печать файлов, печатает их и удаляет их имена из каталога.

Представьте, что каталог спулера состоит из большого числа сегментов, пронумерованных последовательно (0, 1, 2, ...), в каждом их которых может храниться имя файла. Также есть две совместно используемые переменные: `out`,

указывающая на следующий файл для печати, и in, указывающая на следующий свободный сегмент. Эти две переменные можно хранить в одном файле (состоящем из двух слов), доступном всем процессам. Пусть в данный момент сегменты с 0 по 3 пусты (эти файлы уже напечатаны), а сегменты с 4 по 6 заняты (эти файлы ждут своей очереди на печать). Более или менее одновременно процессы A и B решают поставить файл в очередь на печать. Описанная ситуация схематически изображена на рис. 2.5.

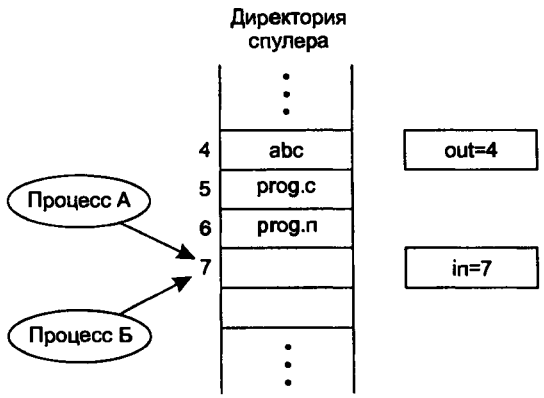


Рис. 2.5. Два процесса хотят одновременно получить доступ к совместно используемой памяти

В соответствии с законом Мерфи (он звучит примерно так: «Если что-то плохое может случиться, оно непременно случится»), возможна следующая ситуация. Процесс A считывает значение (7) переменной in и сохраняет его в локальной переменной next_free_slot. После этого происходит прерывание по таймеру, и процессор переключается на процесс B. Процесс B, в свою очередь, считывает значение переменной in и сохраняет его (опять 7) в своей локальной переменной next_free_slot. В данный момент оба процесса считают, что следующий свободный сегмент — седьмой.

Процесс B сохраняет в каталоге спулера имя файла и заменяет значение in на 8, затем продолжает заниматься своими задачами, не связанными с печатью.

Наконец, управление переходит к процессу A, и он продолжает с того места, на котором остановился. Он обращается к переменной next_free_slot, считывает ее значение и записывает в седьмой сегмент имя файла (разумеется, удаляя при этом имя файла, помещенное туда процессом B). Затем он заменяет значение in на 8 (next_free_slot + 1 = 8). Структура каталога спулера не нарушена, поэтому демон печати не заподозрит ничего плохого, но файл процесса B не будет напечатан. Пользователь, связанный с процессом B, может в этой ситуации полдня описывать круги вокруг принтера, ожидая результата. Ситуации, в которых два (и более) процесса считывают или записывают данные одновременно и конечный результат зависит от того, какой из них был первым, называются *состояниями состязания*. Отладка программы, в которой вероятно возникновение состояния состязания, вряд ли может доставить удовольствие. Результаты боль-

шинства тестов будут хорошими, но изредка будет происходить нечто странное и необъяснимое.

2.2.2. Критические области

Как избежать состязания? Основным способом предотвращения проблем в этой и любой другой ситуации, связанной с конкурентным использованием памяти, файлов и чего-либо еще, является запрет одновременной записи и чтения данных более чем одним процессом. Говоря иными словами, необходимо *взаимное исключение*. То есть в тот момент, когда один процесс использует общие данные, другому процессу это делать будет запрещено. Проблема, описанная в предыдущем разделе, возникла из-за того, что процесс B начал работу с одной из совместно используемых переменных до того, как процесс A ее закончил. Выбор подходящей простейшей операции, реализующей взаимное исключение, является серьезным моментом разработки операционной системы, и мы рассмотрим его подробно в дальнейшем.

Проблему исключения состояний состязания можно сформулировать на абстрактном уровне. Некоторый промежуток времени процесс занят внутренними расчетами и другими задачами, не приводящими к состояниям состязания. В другие моменты времени процесс обращается к совместно используемым данным или выполняет какое-то иное действие, чреватое состязанием. Часть программы, в которой происходит обращение к совместно используемым данным, называется *критической областью* или *критической секцией*. Если нам удастся избежать одновременного нахождения двух процессов в критических областях, мы сможем избежать состязаний.

Несмотря на то что поставленное требование исключает состязание, его недостаточно для правильной совместной работы параллельных процессов и эффективного использования общих данных. Для этого необходимо выполнение четырех условий.

1. Два процесса не должны одновременно находиться в критических областях.
2. В программе не должно быть предположений о скорости или количестве процессоров.
3. Процесс, в состоянии вне критической области, не может блокировать другие процессы.
4. Недопустима ситуация, в которой процесс вечно ждет попадания в критическую секцию.

2.2.3. Взаимное исключение с активным ожиданием

В этом разделе мы рассмотрим различные способы реализации взаимного исключения с целью избежать вмешательства в критическую область одного процесса при нахождении там другого и связанных с этим проблем.

Запрет на прерывания

Самое простое решение состоит в запрете всех прерываний при входе процесса в критическую область и разрешение прерываний по выходу из нее. Если прерывания запрещены, невозможно прерывание по таймеру. Отключение прерываний исключает передачу процессора другому процессу. Таким образом, запретив прерывания, процесс может спокойно считывать и сохранять совместно используемые данные, не опасаясь вмешательства конкурентов.

И все же было бы неразумно давать пользовательскому процессу полномочия запрета прерываний. Представьте себе, что процесс отключил все прерывания и в результате какого-либо сбоя, не включил их обратно. Операционная система на этом может закончить свое существование. К тому же, в многопроцессорной системе запрет прерываний повлияет только на тот процессор, который выполнит инструкцию disable. Остальные процессоры продолжат работу и сохранят доступ к общим данным.

С другой стороны, для ядра характерна блокировка прерываний для некоторых команд при работе с переменными или списками. Возникновение прерывания в момент, когда, например, список готовых процессов находится в неопределенном состоянии, могло бы привести к состоянию состязания. Итак, запрет прерываний бывает полезным в самой операционной системе, но это решение неприемлемо в качестве механизма взаимного исключения для пользовательских процессов.

Переменные блокировки

Теперь попробуем найти программное решение. Рассмотрим одну совместно используемую переменную блокировки, изначально равную 0. Если процесс хочет попасть в критическую область, он предварительно считывает значение переменной блокировки. Если переменная равна 0, процесс изменяет ее на 1 и входит в критическую область. Если же переменная равна 1, то процесс ждет, пока ее значение сменится на 0. Таким образом, 0 означает, что ни одного процесса в критической области нет, а 1 свидетельствует, что какой-либо процесс уже находится в этом состоянии.

К сожалению, у предложенного метода те же проблемы, что и в примере с каталогом буферизации печати. Представьте, что один процесс считывает переменную блокировки, обнаруживает, что она равна 0, но прежде чем он успевает изменить ее на 1, управление получает другой процесс, успешно изменяющий ее на 1. Когда первый процесс снова получит управление, он тоже установит переменную блокировки в 1, и два процесса одновременно окажутся в критических областях.

Можно подумать, что проблема решается повторной проверкой значения переменной, до ее замены, но это не так. Второй процесс может получить управление как раз после того, как первый процесс закончил вторую проверку, но еще не заменил значение переменной блокировки.

Строгое чередование

Третий метод реализации взаимного исключения иллюстрирован на рис. 2.6. Этот фрагмент программного кода, как и многие другие в данной книге, написан на C. Язык C был выбран, поскольку практически все существующие операционные системы написаны на C (или C++), а не на Java, Modula 3, Pascal и т. п. Язык C обладает всеми необходимыми свойствами для написания операционных систем: это мощный, эффективный и предсказуемый язык программирования. А язык Java, например, не является предсказуемым, поскольку у программы, написанной на нем, может в критический момент закончиться свободная память, и она вызовет «сборщика мусора» в исключительно неподходящее время. В случае C это невозможно, поскольку в C процедура «сбора мусора» в принципе отсутствует.

```
while(TRUE) {
      while(turn!=0)      /*wait*/;
      critical_region();
      turn=1;
      noncritical_region();
}
```

а

```
while(TRUE) {
      while(turn!=1)      /*wait*/;
      critical_region();
      turn=0;
      noncritical_region();
}
```

б

Рис. 2.6. Предлагаемое решение проблемы критической области: *а* — процесс 0; *б* — процесс 1. В обоих случаях необходимо удостовериться в наличии точки с запятой, ограничивающей цикл while

На рис. 2.6 целая переменная turn, изначально равная 0, фиксирует, чья очередь входить в критическую область. Вначале процесс 0 проверяет значение turn, считывает 0 и входит в критическую область. Процесс 1 также проверяет значение turn, считывает 0 и после этого крутится в цикле, непрерывно проверяя, когда же значение turn будет равно 1. Постоянная проверка значения переменной в ожидании некоторого значения называется *активным ожиданием*. Подобного способа следует избегать, поскольку он является бесцельной тратой времени процессора. Активное ожидание используется только в случае, когда есть уверенность в незначительности времени ожидания.

Когда процесс 0 покидает критическую область, он изменяет значение turn на 1, позволяя процессу 1 завершить цикл. Предположим, что процесс 1 быстро покидает свою критическую область, так что оба процесса теперь находятся в обычном состоянии, и значение turn равно 0. Теперь процесс 0 выполняет весь цикл быстро, выходит из критической области и устанавливает значение turn равным 1. В итоге в этот момент значение turn равно 1, и оба процесса находятся вне критической области.

Неожиданно процесс 0 завершает работу вне критической области и возвращается к началу цикла. Но на большее он не способен, поскольку значение turn равно 1, и процесс 1 находится вне критической области. Процесс 0 «зависнет» в своем цикле while, ожидая, пока процесс 1 изменит значение turn на 0. Получа-

ется, что метод поочередного доступа к критической области не слишком удачен, если один процесс существенно медленнее другого.

Эта ситуация нарушает третье из сформулированных нами условий: один процесс блокирован другим, не находящимся в критической области. Возвратимся к примеру с каталогом спулера: если заменить критическую область процедурой считывания/записи в каталог спулера, процесс 0 не сможет послать файл на печать, поскольку процесс 1 занят чем-то другим.

Фактически этот метод требует, чтобы два процесса попадали в критические области строго по очереди. Ни один из них не сможет войти в критическую область (например послать файл на печать) два раза подряд. Хотя этот алгоритм и исключает состояния состязания, его нельзя рассматривать всерьез, поскольку он нарушает третье условие успешной работы двух параллельных процессов с совместно используемыми данными.

Алгоритм Петерсона

Датский математик Деккер (T. Dekker) был первым, кто разработал программное решение проблемы взаимного исключения, не требующее строгого чередования. Подробное изложение алгоритма можно найти в [7].

В 1981 году Петерсон (G. L. Peterson) придумал существенно более простой алгоритм взаимного исключения. С этого момента вариант Деккера стал считаться устаревшим. Алгоритм Петерсона, представленный в листинге 2.1, состоит из двух процедур, написанных на ANSI C, что предполагает необходимость прототипов для всех определяемых и используемых функций. В целях экономии места мы не будем приводить прототипы для этого и последующих примеров.

Листинг 2.1. Решение Петерсона для взаимного исключения

```
#define FALSE 0
#define TRUE 1
#define N 2                                     /* Количество процессов */
int turn;                                       /* Чья сейчас очередь? */
int interested[N];                              /* Все переменные изначально
                                                /* равны 0 (FALSE) */

void enter_region(int process);                 /* Процесс 0 или 1 */
{
   int other;                                   /* Номер второго процесса */
   other = 1 - process;                         /* "противоположный" процесс */
   interested[process] = TRUE;                  /* Индикатор интереса */
   turn = process;                              /* Установка флага */
   while (turn == process && interested[other] == TRUE) /* Пустой цикл */;
}

void leave_region(int process)                  /* Процесс, покидающий
                                                /* критическую область */
{
   interested[process] = FALSE;                 /* Индикатор выхода из
                                                /* критической области */
}
```

Прежде чем обратиться к совместно используемым переменным (то есть перед тем, как войти в критическую область), процесс вызывает процедуру enter_region

со своим номером (0 или 1) в качестве аргумента. Поэтому процессу при необходимости придется подождать, прежде чем входить в критическую область. После выхода из критической области процесс вызывает процедуру leave_region, чтобы обозначить свой выход и тем самым разрешить другому процессу вход в критическую область.

Рассмотрим работу алгоритма более подробно. Исходно оба процесса находятся вне критических областей. Процесс 0 вызывает enter_region, задает элементы массива и устанавливает переменную turn равной 0. Поскольку процесс 1 не заинтересован в попадании в критическую область, происходит возврат из процедуры. Теперь, если процесс 1 вызовет enter_region, ему придется подождать, пока interested [0] примет значение FALSE, а это произойдет только в тот момент, когда процесс 0 вызовет процедуру leave_region при покидании критической области.

Представьте, что оба процесса вызвали enter_region практически одновременно. Оба запомнят свои номера в turn. Но сохранится номер того процесса, который был вторым, а предыдущий номер будет утерян. Предположим, что вторым был процесс 1, отсюда значение turn равно 1. Когда оба процесса дойдут до конструкции while, процесс 0 войдет в критическую область, а процесс 1 останется в цикле и будет ждать, пока процесс 0 выйдет из нее.

Команда TSL

Рассмотрим решение, требующее участия аппаратного обеспечения. Многие компьютеры, особенно разработанные с расчетом на несколько процессоров, имеют команду Test and Set Lock (*TSL*) — «проверить и заблокировать», которая действует следующим образом. В регистр считывается содержимое слова памяти, а затем в этой ячейке памяти сохраняется ненулевое значение. Гарантируется, что операция считывания слова и сохранения *неделима* — другой процесс не может обратиться к слову в памяти, пока команда не выполнена. Процессор, выполняющий команду tsl, блокирует шину памяти, препятствуя обращениям к памяти со стороны остальных процессоров.

Воспользуемся командой tsl. Пусть разделяемая переменная lock управляет доступом к общей памяти. Если значение lock равно 0, любой процесс может изменить его на 1 и обратиться к общей памяти, а затем вернуть его обратно в 0, пользуясь обычной командой типа move.

Как использовать команду tsl для реализации взаимного исключения? Решение приведено в листинге 2.2. Здесь представлена подпрограмма из четырех команд, написанная на некотором обобщенном (но типичном) ассемблере. Первая команда копирует старое значение lock в регистр и затем устанавливает значение переменной в 1. Потом старое значение сравнивается с нулем. Если оно ненулевое, значит, блокировка уже была произведена, и проверка начинается сначала. Рано или поздно значение окажется нулевым (это означает, что процесс, находившийся в критической области, покинул ее), и подпрограмма вернет управление в вызвавшую программу, установив блокировку. Сброс блокировки не представляет собой ничего сложного — просто помещается 0 в переменную lock. Специальной команды процессора не требуется.

Листинг 2.2. Вход и выход из критической области с помощью команды tsl

```
enter_region:
   tsl register,lock   /* значение lock копируется в регистр */
                       /* значение переменной устанавливается равным 1 */
   cmp register,#0     /* Старое значение lock сравнивается с нулем */
   jne enter_region    /* Если оно ненулевое, значит, блокировка уже была */
                       /* установлена, поэтому цикл */
   ret                 /* Возврат в вызывающую программу */
                       /* процесс вошел в критическую область */
leave_region:
   move lock,#0        /* Сохранение 0 в переменной lock */
   ret
```

Одно решение проблемы критических областей теперь очевидно. Прежде чем попасть в критическую область, процесс вызывает процедуру enter_region, которая выполняет активное ожидание вплоть до снятия блокировки, затем она устанавливает блокировку и возвращает управление. По выходу из критической области процесс вызывает процедуру leave_region, помещающую 0 в переменную lock. Как и во всех остальных решениях проблемы критической области, для корректной работы процесс должен вызывать эти процедуры своевременно, в противном случае исключить взаимное исключение не удастся.

2.2.4. Примитивы межпроцессного взаимодействия

Оба решения — Петерсона и с помощью команды TSL — корректны, но они обладают одним и тем же недостатком: наличием активного ожидания. В сущности, оба они реализуют следующий алгоритм: перед входом в критическую область процесс проверяет, можно ли это сделать. Если нельзя, процесс попадает в цикл, ожидая разрешения на вход в критическую область.

Этот алгоритм не только бесцельно расходует время процессора, но, кроме этого, он может иметь некоторые неожиданные последствия. Рассмотрим два процесса: H, с высоким приоритетом, и L, с низким приоритетом. Правила планирования в этом случае таковы, что процесс H запускается немедленно, как только он оказывается в состоянии ожидания. В какой-то момент, когда процесс L находится в критической области, процесс H оказывается в состоянии ожидания (например, он закончил операцию ввода/вывода). Процесс H попадает в состояние активного ожидания, но поскольку процессу L при условии работающего процесса H процессорное время никогда не будет предоставлено, у процесса L не будет возможности выйти из критической области, и процесс H навсегда останется в цикле. Эту ситуацию иногда называют *проблемой инверсии приоритета*.

Теперь рассмотрим некоторые примитивы межпроцессного взаимодействия, применяемые вместо циклов ожидания (в которых лишь напрасно расходуется процессорное время). Эти примитивы блокируют процессы в случае запрета на вход в критическую область. Одной из простейших является пара примитивов sleep и wakeup. Примитив sleep — системный запрос, в результате которого вызывающий процесс блокируется, пока его не запустит другой процесс. Запрос wakeup ожидает один аргумент — идентификатор запускаемого процесса. Также

возможно наличие одного аргумента у обоих запросов (ожидания и запуска) — адреса ячейки памяти, предназначенной для их согласования.

Проблема производителя и потребителя

В качестве примера использования этих примитивов рассмотрим проблему *производителя и потребителя*, также известную как проблема *ограниченного буфера*. Два процесса совместно используют буфер ограниченного размера. Один из них, производитель, помещает данные в этот буфер, а другой, потребитель, считывает их оттуда. (Можно обобщить задачу на случай m производителей и n потребителей, но мы разберем случай с одним производителем и одним потребителем, поскольку это существенно упрощает решение.)

Трудности начинаются в тот момент, когда производитель желает поместить в буфер очередную порцию данных и обнаруживает, что тот полон. Для производителя решением является ожидание, пока потребитель полностью или частично не очистит буфер. Аналогично, если потребитель хочет забрать данные из буфера, а буфер пуст, потребитель переключается в состояние ожидания и выходит из него, как только производитель положит что-нибудь в буфер и разбудит спящего.

Это решение кажется достаточно простым, но оно приводит к состояниям состязания, как и пример с каталогом спулера. Нам нужна переменная count для отслеживания количества элементов в буфере. Если максимальное число элементов, хранящихся в буфере, равно N, программа производителя должна проверить, не равно ли N значение count, прежде чем поместить в буфер следующую порцию данных. Если значение count равно N, производитель переключается в состояние ожидания; в противном случае он помещает данные в буфер и увеличивает значение count.

Код программы потребителя прост: сначала проверить, не равно ли значение count нулю. Если равно, то уйти в состояние ожидания; иначе забрать порцию данных из буфера и уменьшить значение count. Каждый из процессов также должен проверять, не следует ли активизировать другой процесс, и в случае необходимости проделывать это. Программы обоих процессов представлены в листинге 2.3.

Листинг 2.3. Проблема производителя и потребителя с неустранимым состоянием соревнования

```
#define N 100                   /* Максимальное количество элементов
                                /* в буфере */
int count = 0;                  /* Текущее количество элементов в буфере */

void producer(void)
{
   int item;

   while (TRUE) {               /* Повторять вечно */
      item = produce_item();    /* Сформировать следующий элемент */
      if (count == N) sleep();  /* Если буфер полон, уйти в состояние
                                /* ожидания */
      insert_item(item);        /* Поместить элемент в буфер */
```

```
      count = count + 1;                    /* Увеличить количество элементов в буфере */
      if (count == 1) wakeup(consumer);     /* Был ли буфер пуст? */
   }
}
void consumer(void)
{
   int item;
   while (TRUE) {                           /* Повторять вечно */
      if (count == 0) sleep();              /* Если буфер пуст, уйти в состояние
                                               ожидания */
      item = remove_item( );                /* Забрать элемент из буфера */
      count = count - 1;                    /* Уменьшить счетчик элементов в буфере */
      if (count == N - 1) wakeup(producer); /* Был ли буфер полон? */
      consume_item(item);                   /* Отправить элемент на печать */
   }
}
```

Для описания на языке C системных вызовов sleep и wakeup мы представили их в виде вызовов библиотечных процедур. В стандартной библиотеке C их нет, но они будут доступны в любой системе, в которой присутствуют такие системные вызовы. Процедуры insert_item и remove_item помещают элементы в буфер и извлекают их оттуда.

Теперь давайте вернемся к состоянию состязания. Его возникновение вероятно, поскольку доступ к переменной count не ограничен. Может возникнуть следующая ситуация: буфер пуст, и потребитель только что считал значение переменной count, чтобы проверить, не равно ли оно нулю. В этот момент планировщик передал управление производителю, производитель поместил элемент в буфер и увеличил значение count, убедившись, что теперь оно стало равно 1. Зная, что ранее значение было равно 0, а потребитель находился в состоянии ожидания, производитель активизирует потребителя с помощью вызова wakeup.

Но потребитель не был в состоянии ожидания, следовательно, сигнал активизации пропал впустую. Когда управление перейдет к потребителю, он вернется к считанному когда-то значению count, обнаружит, что оно равно 0, и уйдет в состояние ожидания. Рано или поздно производитель наполнит буфер и также перейдет в состояние ожидания. Оба процесса так и останутся в состоянии простоя.

Суть проблемы в данном случае состоит в том, что сигнал активизации, поступивший процессу, не находящемуся в состоянии ожидания, уходит в никуда. Если бы не это, проблемы бы не было. Быстрым решением может быть добавление *бита ожидания активизации*. Если сигнал активизации послан процессу, не находящемуся в состоянии ожидания, этот бит устанавливается. Позже, когда процесс пытается перейти в состояние ожидания, бит ожидания активизации сбрасывается, но процесс остается активным. Этот бит исполняет роль копилки сигналов активизации.

Несмотря на то что введение бита ожидания запуска спасло положение в нашем примере, легко представить ситуацию с несколькими процессами, в которой одного бита будет недостаточно. Мы можем добавить еще один бит, или 8, или 32, но это не решит проблему.

2.2.5. Семафоры

В 1965 году Дейкстра (E. W. Dijkstra) предложил использовать целую переменную для подсчета сигналов запуска, сохраненных на будущее. Им был предложен новый тип переменных, так называемые *семафоры*, значение которых может быть нулем (в случае отсутствия сохраненных сигналов активизации) или некоторым положительным числом, соответствующим количеству отложенных сигналов.

Дейкстра предложил две операции, down и up (обобщения sleep и wakeup). Операция down сравнивает значение семафора с нулем. Если значение семафора больше нуля, операция down уменьшает его (то есть расходует один из сохраненных сигналов активизации) и просто возвращает управление. Если значение семафора равно нулю, процедура down не возвращает управление процессу, а процесс переводится в состояние ожидания. Все операции проверки значения семафора, его изменения и перевода процесса в состояние ожидания выполняются как единое и неделимое *элементарное действие*. Тем самым гарантируется, что после начала операции ни один процесс не получит доступа к семафору до окончания или блокирования операции. Атомарность операции чрезвычайно важна для разрешения проблемы синхронизации и предотвращения состояния состязания.

Операция up увеличивает значение семафора. Если с этим семафором связаны один или несколько ожидающих процессов, которые не могут завершить более раннюю операцию down, один из них выбирается системой (например, случайным образом) и ему разрешается завершить свою операцию down. Таким образом, после операции up, примененной к семафору, связанному с несколькими ожидающими процессами, значение семафора так и останется равным 0, но число ожидающих процессов уменьшится на единицу. Операция увеличения значения семафора и активизации процесса тоже неделима. Ни один процесс не может быть блокирован во время выполнения операции up, как ни один процесс не мог быть блокирован во время выполнения операции wakeup в предыдущей модели.

В оригинале Дейкстра употреблял вместо down и up обозначения P и V соответственно. Мы не будем в дальнейшем использовать оригинальные обозначения, поскольку тем, кто не знает голландского языка, эти обозначения ничего не говорят (да и тем, кто знает язык, говорят немного). Впервые обозначения down и up появились в языке Алгол 68.

Решение проблемы производителя и потребителя с помощью семафоров

Как показано в листинге 2.4, проблему потерянных сигналов запуска можно решить с помощью семафоров. Очень важно, чтобы они были реализованы неделимым образом. Стандартным способом является реализация операций down и up в виде системных запросов, с запретом операционной системой всех прерываний на период проверки семафора, изменения его значения и возможного перевода процесса в состояние ожидания. Поскольку для выполнения всех этих действий требуется всего лишь несколько команд процессора, запрет прерываний не при-

носит никакого вреда. Если используются несколько процессоров, каждый семафор необходимо защитить переменной блокировки посредством команды tsl, чтобы гарантировать одновременное обращение к семафору только одного процессора. Необходимо понимать, что использование команды tsl принципиально отличается от активного ожидания, при котором производитель или потребитель ждут наполнения или опустошения буфера. Операция с семафором займет несколько микросекунд, тогда как активное ожидание может затянуться на существенно больший промежуток времени.

Листинг 2.4. Решение проблемы производителя и потребителя с помощью семафоров

```
#define N 100                   /* Количество сегментов в буфере */
typedef int semaphore;          /* Семафоры – особый вид целочисленных
                                /* переменных */
semaphore mutex = 1;            /* Контроль доступа в критическую область */
semaphore empty = N;            /* Число пустых сегментов буфера */
semaphore full = 0;             /* Число полных сегментов буфера */

void producer(void)
{
   int item;

   while (TRUE) {               /* TRUE – константа, равная 1*/
      item = produce_item();    /* Создать данные, помещаемые в буфер */
      down(&empty);             /* Уменьшить счетчик пустых сегментов буфера */
      down(&mutex);             /* Вход в критическую область */
      insert_item(item);        /* Поместить в буфер новый элемент */
      up(&mutex);               /* Выход из критической области */
      up(&full);                /* Увеличить счетчик полных сегментов буфера */
   }
}

void consumer(void)
{
   int item;
   while (TRUE) {               /* Бесконечный цикл */
      down(&full);              /* Уменьшить число полных сегментов буфера */
      down(&mutex);             /* Вход в критическую область */
      item = remove_item();     /* Удалить элемент из буфера */
      up(&mutex);               /* Выход из критической области */
      up(&empty);               /* Увеличить счетчик пустых сегментов буфера */
      consume_item(item);       /* Обработка элемента */
   }
}
```

В представленном решении используются три семафора: один для подсчета заполненных сегментов буфера (full), другой для подсчета пустых сегментов (empty), а третий предназначен для исключения одновременного доступа производителя и потребителя (mutex) к буферу. Значение счетчика full исходно равно нулю, счетчик empty равен числу сегментов в буфере, а mutex равен 1. Семафоры, исходное значение которых установлено в 1, предназначенные для исключения одновременного нахождения в критической области двух процессов, называются *двоичными семафорами*. Взаимное исключение обеспечивается, если каждый

процесс выполняет операцию down перед входом в критическую область и up — после выхода из нее.

Теперь, когда у нас есть примитивы межпроцессного взаимодействия, вернемся к последовательности обработки прерываний, показанной в конце раздела 2.1.2. В системах, использующих семафоры, естественным способом скрыть прерывание будет связать с каждым устройством ввода/вывода семафор, изначально равный нулю. Сразу после включения устройства ввода/вывода управляющий процесс выполняет операцию down на соответствующем семафоре, тем самым входя в состояние блокировки. В случае прерывания обработчик прерывания выполняет up на соответствующем семафоре, переводя процесс в состояние готовности. В такой модели пятый шаг в алгоритме из раздела 2.1.2 заключается в выполнении up на семафоре устройства, чтобы следующим шагом планировщик смог запустить программу, управляющую устройством. Разумеется, если в этот момент несколько процессов находятся в состоянии готовности, планировщик вправе выбрать другой, более значимый процесс. Мы рассмотрим некоторые алгоритмы планирования позже в этой главе.

В примере, представленном в листинге 2.4, семафоры использовались двояко. Это различие достаточно ощутимо, чтобы сказать о нем особо. Семафор mutex предназначен для реализации взаимного исключения, то есть для исключения одновременного обращения к буферу и к связанным переменным двух процессов. Мы уже рассмотрели взаимное исключение и методы его реализации в предыдущем разделе.

Остальные семафоры введены с целью *синхронизации*. Семафоры full и empty необходимы, чтобы гарантировать, что определенные последовательности событий происходят или не происходят. В нашем случае они дают гарантию, что производитель прекращает работу, когда буфер полон, а потребитель прекращает ее, когда буфер пуст.

2.2.6. Мониторы

Межпроцессное взаимодействие с применением семафоров выглядит довольно просто, не правда ли? Эта простота кажущаяся. Взгляните внимательнее на порядок выполнения процедур down перед помещением или удалением элементов из буфера в листинге 2.4. Представьте себе, что две процедуры down в программе производителя поменялись местами, так что значение mutex было уменьшено раньше, чем empty. Если буфер был заполнен, производитель блокируется, сбросив mutex в 0. Соответственно, в следующий раз, когда потребитель обратится к буферу, он выполнит down с переменной mutex, равной 0, и тоже заблокируется. Оба процесса заблокированы навсегда. Эта неприятная ситуация называется взаимоблокировкой, и мы вернемся к ней в главе 3.

Вышеизложенная ситуация показывает, с какой аккуратностью нужно обращаться с семафорами. Одна маленькая ошибка, и все останавливается. Это напоминает программирование на ассемблере, но на самом деле еще сложнее, поскольку такие ошибки приводят к абсолютно невоспроизводимым и непредсказуемым состояниям состязания, взаимоблокировкам и т. п.

Чтобы упростить написание программ, в 1974 году Хоар (Hoare) [43] и Бринч Хансен (Brinch Hansen) предложили примитив синхронизации более высокого уровня, называемый *монитором*. Их предложения несколько отличались друг от друга, как мы увидим дальше. Монитор — набор процедур, переменных и других структур данных, объединенных в особый модуль или пакет. Процессы могут вызывать процедуры монитора, но у процедур, объявленных вне монитора, нет прямого доступа к внутренним структурам данных монитора. В листинге 2.5 представлен монитор, написанный на воображаемом языке, некоем «местечковом диалекте» — «пиджин» Pascal.

Листинг 2.5. Монитор

```
monitor example
   integer i;
   condition c;

   procedure producer();
   .
   .
   .
   end;

   procedure consumer();

   end;
end monitor;
```

Реализации взаимных исключений способствует важное свойство монитора: при обращении к монитору в любой момент времени активным может быть только один процесс. Мониторы являются структурным компонентом языка программирования, поэтому компилятор знает, что обрабатывать вызовы процедур монитора следует иначе, чем вызовы остальных процедур. Обычно при вызове процедуры монитора первые несколько команд процедуры проверяют, нет ли в мониторе активного процесса. Если таковой есть, вызывающему процессу придется подождать, в противном случае запрос удовлетворяется.

Реализация взаимного исключения зависит от компилятора, но обычно используется двоичный семафор. Поскольку взаимное исключение обеспечивает компилятор, а не программист, вероятность ошибки гораздо меньше. В любом случае программист, пишущий код монитора, не должен задумываться о том, как компилятор организует взаимное исключение. Достаточно знать, что обеспечив попадание в критические области через процедуры монитора, можно не бояться нахождения в критических областях двух процессов одновременно.

Хотя мониторы предоставляют простой способ реализации взаимного исключения, этого недостаточно. Необходим также способ блокировки процессов, которые не могут продолжать свою деятельность. В случае проблемы производителя и потребителя достаточно просто поместить все проверки буфера на не-пустоту и пустоту в процедуры монитора, но как процесс заблокируется, обнаружив полный буфер?

Решение заключается в *переменных состояния* и двух операциях, wait и signal. Когда процедура монитора обнаруживает, что она не в состоянии продолжать

работу (например, производитель выясняет, что буфер заполнен), она выполняет операцию wait на какой-либо переменной состояния, скажем, full. Это приводит к блокировке вызывающего процесса и позволяет другому процессу войти в монитор.

Другой процесс, в нашем примере потребитель, может активизировать ожидающего напарника, например, выполнив операцию signal на той переменной состояния, на которой он был заблокирован. Чтобы в мониторе не оказалось двух активных процессов одновременно, нам необходимо правило, определяющее последствия операции signal. Хоар предложил запуск «разбуженного» процесса и остановку второго. Бринч Хансен придумал другое решение: процесс, выполнивший signal, должен немедленно покинуть монитор. Иными словами, операция signal выполняется только в самом конце процедуры монитора. Мы будем использовать это решение, поскольку оно в принципе проще и к тому же легче в реализации. Если операция signal выполнена на переменной, с которой связаны несколько заблокированных процессов, планировщик выбирает и «оживляет» только один из них.

Кроме этого, существует третье решение, не основывающееся на предположениях Хоара и Хансена: позволить процессу, выполнившему signal, продолжать работу и запустить ждущий процесс только после того, как первый процесс покинет монитор.

Переменные состояния не являются счетчиками. В отличие от семафоров они не аккумулируют сигналы, чтобы впоследствии воспользоваться ими. Это означает, что в случае выполнения операции signal на переменной состояния, с которой не связано ни одного блокированного процесса, сигнал будет утерян. Проще говоря, операция wait должна выполняться прежде, чем signal. Последнее правило существенно упрощает реализацию. На практике оно не создает проблем, поскольку отслеживать состояния процессов при необходимости не очень трудно. Процесс, который собирается выполнить signal, может оценить необходимость этого действия по значениям переменных.

В листинге 2.6 представлена схема решения проблемы производителя и потребителя с применением мониторов, написанная на «пиджин» Pascal. В данной ситуации этот суррогат языка удобен своей простотой, а также тем, что он позволяет в точности следовать моделям Хоара и Хансена. В каждый момент времени активна только одна процедура монитора. Буфер состоит из N сегментов.

Листинг 2.6. Схема решения проблемы производителя и потребителя с применением мониторов

```
monitor ProducerConsumer
      condition full, empty;
      integer count;

      procedure insert(item: integer);
      begin
         if count = N then wait(full);
         insert_item(item);
         count := count+1;
         if count = 1 then signal(empty)
      end;
```

```
     function remove: integer;
     begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count-1;
        if count = N-1 then signal(full)
     end;
     count := 0;
end monitor;

procedure producer;
begin
     while true do
     begin
        item = produce_item;
        ProducerConsumer.insert(item)
     end
end;

procedure consumer;
begin
     while true do
     begin
        item = ProducerConsumer.remove;
        consume_item(item)
     end
end;
```

Можно подумать, что операции wait и signal похожи на sleep и wakeup, которые приводили к неустранимым состояниям конкуренции. Они действительно похожи, но с одним существенным отличием: неудачи применения операций sleep и wakeup были связаны с тем, что один процесс пытался уйти в состояние ожидания, в то время как другой процесс предпринимал попытки активизировать его. С мониторами такого произойти не может. Автоматическое взаимное исключение, реализуемое процедурами монитора, гарантирует: если производитель, находящийся в мониторе, обнаружит полный буфер и решит выполнить операцию wait, можно не опасаться, что планировщик передаст управление потребителю раньше, чем операция wait будет завершена. Потребитель даже не сумеет попасть в монитор, пока операция wait не будет выполнена и производитель не прекратит работу.

Благодаря автоматизации взаимного исключения применение мониторов сделало параллельное программирование значительно менее подверженным ошибкам, в отличие от применения семафоров. Но и у мониторов тоже есть свои недостатки. Недаром два примера мониторов, которые мы рассмотрели, были написаны на «пиджин» Pascal, а не на C, как все остальные примеры этой книги. Как мы уже говорили, мониторы являются структурным компонентом языка программирования, и компилятор должен их распознавать и организовывать взаимное исключение. В Pascal, C и многих других языках нет мониторов, поэтому странно было бы ожидать от их компиляторов выполнения правил взаимного исключения. И в самом деле, как отличить компилятор процедуры монитора от остальных?

Другая проблема, связанная с мониторами и семафорами, состоит в том, что они были разработаны для решения задачи взаимного исключения в системе с одним или несколькими процессорами, имеющими доступ к общей памяти. Помещение семафоров в разделенную память с защитой в виде команд TSL может исключить состояния состязания. Эти примитивы будут неприменимы в распределенной системе, состоящей из нескольких процессоров с собственной памятью у каждого, связанных локальной сетью. Вывод из всего вышесказанного следующий: семафоры являются примитивами слишком низкого уровня, а мониторы применимы только в некоторых языках программирования. Примитивы не подходят и для реализации обмена информацией между компьютерами — нужно что-то другое.

2.2.7. Передача сообщений

В роли чего-то другого выступает *передача сообщений*. Межпроцессное взаимодействие такого рода строится на двух примитивах: send и receive, которые скорее являются системными вызовами, нежели структурными компонентами языка (что отличает их от мониторов и делает похожим на семафоры). Поэтому их легко можно инкапсулировать в библиотечные процедуры, например:

```
send(destination, &message);
receive(source, &message);
```

Первый запрос посылает сообщение заданному адресату, а второй получает сообщение от указанного источника (или от любого источника, если это не имеет значения). Если сообщения нет, второй запрос блокируется до поступления сообщения либо немедленно возвращает код ошибки.

Разработка систем передачи сообщений

С системами передачи сообщений связано большое количество сложных проблем и конструктивных вопросов, которых не возникает в случае семафоров и мониторов. Особенно много сложностей появляется в случае взаимодействия процессов, проистекающих на различных компьютерах, соединенных сетью. Так, сообщение может затеряться в сети. Чтобы избежать потери сообщений, отправитель и получатель договариваются, что при получении сообщения получатель посылает обратно *подтверждение* приема. Если отправитель не получает подтверждение через некоторое время, он отсылает сообщение еще раз.

Теперь представим, что сообщение получено, но подтверждение до отправителя не дошло. Отправитель направит сообщение еще раз, и до получателя оно дойдет дважды. Крайне важно, чтобы получатель мог отличить копию предыдущего сообщения от нового. Обычно проблема решается с помощью занесения порядкового номера сообщения в тело самого сообщения. Если к получателю приходит письмо с номером, совпадающим с номером предыдущего письма, письмо классифицируется как копия и игнорируется. Решение проблемы успешного обмена информации в условиях ненадежной передачи сообщений составляет основу изучения компьютерных сетей.

Для систем обмена сообщениями также важен вопрос названий процессов. Необходимо однозначно определять процесс, указанный в запросе send или receive. Кроме того, встает вопрос *аутентификации*: каким образом клиент может определить, что он взаимодействует с настоящим файловым сервером, а не с самозванцем?

Помимо этого существуют конструктивные проблемы, существенные при расположении отправителя и получателя на одном компьютере. Одной из таких проблем является производительность. Копирование сообщений из одного процесса в другой происходит гораздо медленнее, чем операция на семафоре или вход в монитор. Было проведено множество исследований с целью увеличения эффективности передачи сообщений. В [10], например, предлагалось ограничивать размер сообщения до размеров регистра и передавать сообщения через регистры.

Решение проблемы производителя и потребителя путем передачи сообщений

Теперь рассмотрим решение проблемы производителя и потребителя посредством передачи сообщений и без использования общей памяти. Решение представлено в листинге 2.7. Мы предполагаем, что все сообщения имеют одинаковый размер, и сообщения, которые посланы, но еще не получены, автоматически помещаются операционной системой в буфер. В этом решении используются N сообщений, по аналогии с N сегментами в буфере. Потребитель начинает с того, что посылает производителю N пустых сообщений. Как только у производителя оказывается элемент данных, который он может предоставить потребителю, он берет пустое сообщение и отсылает назад полное. Таким образом, общее число сообщений в системе постоянно, и их можно хранить в заранее отведенной области памяти.

Листинг 2.7. Решение проблемы производителя и потребителя с использованием N сообщений

```
#define N 100                       /* Количество сегментов в буфере */
void producer(void)
{
     int item;
     message m;                     /* Буфер для сообщений */
     while (TRUE) {
        item = produce_item();      /* Сформировать нечто, чтобы заполнить
                                    /* буфер */
        receive(consumer, &m);      /* Ожидание прибытия пустого сообщения */
        build_message(&m, item);    /* Сформировать сообщение для отправки */
        send(consumer, &m);         /* Отослать элемент потребителю */
     }
}
void consumer(void)
{
     int item, i;
     message m;
     for (i = 0; i < N; i++)
     {
        send(producer, &m);
```

```
    }                              /* Отослать N пустых сообщений */
    while (TRUE) {
        receive(producer, &m);     /* Получить сообщение с элементом */
        item = extract_item(&m);   /* Извлечь элемент из сообщения */
        send(producer, &m);        /* Отослать пустое сообщение */
        consume_item(item);        /* Обработка элемента */
    }
}
```

Если производитель работает быстрее, чем потребитель, все сообщения будут ожидать потребителя в цельном виде. При этом производитель блокируется в ожидании пустого сообщения. Если потребитель работает быстрее, ситуация инвертируется: все сообщения будут пустыми, а потребитель будет блокирован в ожидании полного сообщения.

Передача сообщений реализуется по-разному. Рассмотрим способ адресации сообщений. Можно присвоить каждому из процессов уникальный адрес и адресовать сообщение непосредственно процессам. Другой подход состоит в использовании новой структуры данных, называемой *почтовым ящиком*. Почтовый ящик — это буфер для определенного количества сообщений, тип которых задается при создании ящика. При использовании почтовых ящиков в качестве параметров адреса send и receive задаются почтовые ящики, а не процессы. Если процесс пытается послать сообщение в переполненный почтовый ящик, ему приходится подождать, пока хотя бы одно сообщение не будет оттуда удалено.

В задаче производителя и потребителя оба они создадут почтовые ящики, достаточно большие, чтобы хранить N сообщений. Производитель будет посылать сообщения с данными в почтовый ящик потребителя, а потребитель станет отправлять пустые сообщения в почтовый ящик производителя. В случае почтовых ящиков способ буферизации очевиден: в почтовом ящике получателя хранятся сообщения, которые были посланы процессу-получателю, но еще не получены.

Другой крайностью при использовании почтовых ящиков является принципиальное отсутствие буферизации. При таком подходе, если send выполняется раньше, чем receive, процесс-отправитель блокируется до выполнения receive, когда сообщение может быть напрямую скопировано от отправителя к получателю без промежуточной буферизации. Если receive выполняется раньше, чем send, процесс-получатель блокируется до выполнения send. Этот метод часто называется *рандеву*, он легче реализуется, чем схема буферизации сообщений, но менее гибок, поскольку отправитель и получатель должны работать в режиме жесткой синхронизации.

В операционной системе MINIX взаимодействие между процессами происходит посредством каналов, которые по своей сути являются почтовыми ящиками. Единственная разница между механизмом каналов и настоящими почтовыми ящиками в том, что в каналах нет разграничения сообщений. Другими словами, если источник поместит в канал 10 сообщений по 100 байт, а приемник прочитает 1000 байт, то он сразу прочитает все 0 сообщений. Конечно, если существует договоренность всегда использовать сообщения одного и того же размера, то подобных проблем не возникает. Кроме того, можно ввести специальный символ

для разделения сообщений (скажем, перевод строки). Например, в процессах, образующих ОС MINIX, для обмена информацией применена схема с сообщениями фиксированного размера.

2.3. Классические проблемы межпроцессного взаимодействия

Литература по операционным системам охватывает множество интересных проблем, которые широко обсуждались и анализировались с применением различных методов синхронизации. В этом разделе мы рассмотрим три наиболее известных проблемы.

2.3.1. Проблема обедающих философов

В 1965 году Дейкстра сформулировал и решил проблему синхронизации, названную им *проблемой обедающих философов*. С тех пор каждый, кто изобретал еще один новый примитив синхронизации, считал своим долгом продемонстрировать достоинства нового примитива на примере проблемы обедающих философов. Задачу можно сформулировать следующим образом: пять философов сидят за круглым столом и у каждого есть тарелка со спагетти. Спагетти настолько скользкие, что каждому философу нужны две вилки, чтобы управиться с яством. Но между каждыми двумя тарелками лежит одна вилка (рис. 2.7).

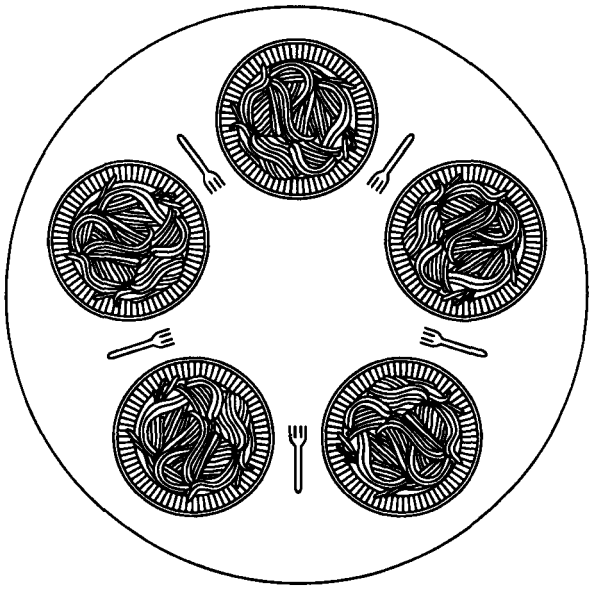


Рис. 2.7. Время обеда на факультете философии

Жизнь философа состоит из чередующихся периодов поглощения пищи и размышлений. (Разумеется, это абстракция, даже применительно к философам, но остальные процессы жизнедеятельности для нашей задачи несущественны.) Когда философ голоден, он пытается получить две вилки, левую и правую, в любом порядке. Если ему удалось завладеть двумя вилками, он некоторое время ест, затем кладет вилки обратно и продолжает размышления. Вопрос состоит в следующем: можно ли написать алгоритм, который моделирует эти действия для каждого философа и никогда не застревает? (Кое-кто считает, что необходимость двух вилок выглядит несколько искусственно. Возможно, нам следует заменить итальянскую пищу блюдами китайской кухни, спагетти — рисом, а вилки — соответствующими палочками.)

В листинге 2.8 представлено очевидное решение проблемы. Процедура take_fork ждет, пока указанная вилка не освободится, и берет ее. К сожалению, это решение неверно — представьте себе, что все пять философов возьмут одновременно свои левые вилки. Каждый останется без правой вилки, и произойдет взаимоблокировка.

Листинг 2.8. Неверное решение проблемы обедающих философов

```
#define N 5                     /* Количество философов */

void philosospher(int i)        /* i - номер философа, от 0 до 4 */
{
     while(TRUE) {
        think();                /* Философ размышляет */
        take_fork(i);           /* Берет левую вилку */
        take_fork((i+1) % N);   /* Берет правую вилку */
        eat();                  /* Спагетти, ням-ням */
        put_fork(i);            /* Кладет на стол левую вилку */
        put_fork((i+1) % N);    /* Кладет на стол правую вилку */
     }
}
```

Можно изменить программу так, чтобы после получения левой вилки проверялась доступность правой. Если правая вилка недоступна, философ отдает левую обратно, ждет некоторое время и повторяет весь процесс. Этот подход также не будет работать, хотя и по другой причине. Если не повезет, все пять философов могут начать процесс одновременно, взять левую вилку, обнаружить отсутствие правой, положить левую обратно на стол, одновременно взять левую вилку, и так до бесконечности. Ситуация, в которой все программы продолжают работать сколь угодно долго, но не могут добиться хоть какого-то прогресса, называется *зависанием*. (По-английски starvation буквально «умирание от голода».)

Вы можете подумать: «Если философы будут размышлять в течение некоторого случайно выбранного промежутка времени после неудачной попытки взять правую вилку, вероятность того, что все процессы будут продолжать топтаться на месте хотя бы в течение часа, невелика». Это правильно, и для большинства приложений повторение попытки спустя некоторое время снимает проблему. Например, в локальной сети Ethernet в ситуации, когда два компьютера посылают пакеты одновременно, каждый должен подождать случайно заданное время и повторить попытку — на практике это решение хорошо работает. Тем не менее

в некоторых приложениях предпочтительным является другое решение, работающее всегда и не зависящее от случайных чисел (например, в приложении для обеспечения безопасности на атомных электростанциях).

В листинг 2.8 можно внести улучшение, исключающее взаимоблокировку и зависание процесса: защитить пять операторов, следующих за запросом think, бинарным семафором. Тогда философ должен будет выполнить операцию down на переменной mutex прежде, чем потянуться к вилкам. А после возврата вилок на место ему следует выполнить операцию up на переменной mutex. С теоретической точки зрения решение вполне подходит. С позиций практики возникают проблемы с эффективностью: в каждый момент времени может есть спагетти только один философ. Но вилок пять, поэтому необходимо разрешить есть в каждый момент времени двум философам.

Решение, представленное в листинге 2.9, исключает взаимоблокировку и позволяет реализовать максимально возможный параллелизм для любого числа философов. Здесь используется массив state для отслеживания душевного состояния каждого философа: он либо ест, либо размышляет, либо голодает (пытаясь получить вилки). Философ может начать есть, только если ни один из его соседей не ест. Соседи философа *i* определяются макросами LEFT и RIGHT (то есть если *i* = 2, то LEFT = 1 и RIGHT = 3).

Листинг 2.9. Решение задачи обедающих философов

```
#define N         5          /* Количество философов */
#define LEFT      (i+N.1)%N  /* Номер левого соседа философа с номером i */
#define RIGHT     (i+1)%N    /* Номер правого соседа философа с номером i */
#define THINKING  0          /* Философ размышляет */
#define HUNGRY    1          /* Философ пытается получить вилки */
#define EATING    2          /* Философ ест */
typedef int semaphore;       /* Семафоры – особый вид целочисленных
                             /* переменных */
int state[N];                /* Массив для отслеживания состояния каждого
                             /* философа */
semaphore mutex = 1;         /* Взаимоисключение для критических областей */
semaphore s[N];              /* Каждому философу по семафору */

void philosopher(int i)      /* i – номер философа. от 0 до N-1 */
{
      while (TRUE) {         /* Повторять до бесконечности */
         think();            /* Философ размышляет */
         take_forks(i);      /* Получает две вилки или блокируется */
         eat();              /* Спагетти. ням-ням */
         put_forks(i);       /* Кладет на стол обе вилки */
      }
}

void take_forks(int i)       /* i – номер философа. от 0 до N-1*/
{
      down(&mutex);          /* Вход в критическую область */
      state[i] = HUNGRY;     /* Фиксация наличия голодного философа */
      test(i);               /* Попытка получить две вилки */
      up(&mutex);            /* Выход из критической области */
      down(&s[i]);           /* Блокировка, если вилок не досталось */
}
```

```
void put_forks(i)              /* i - номер философа, от 0 до N-1*/
{
      down(&mutex);            /* Вход в критическую область */
      state[i] = THINKING;     /* Философ перестал есть */
      test(LEFT);              /* Проверить, может ли есть сосед слева */
      test(RIGHT);             /* Проверить, может ли есть сосед справа */
      up(&mutex);              /* Выход из критической области */
}

void test(i)                   /* i - номер философа, от 0 до N-1*/
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
         state[i] = EATING;
         up(&s[i]);
      }
}
```

В программе используется массив семафоров, по одному на философа, чтобы блокировать голодных философов, если их вилки заняты. Обратите внимание, что каждый процесс запускает процедуру philosopher в качестве своей основной программы, но остальные процедуры take_forks, put_forks и test являются обычными процедурами, а не отдельными процессами.

2.3.2. Проблема читателей и писателей

Проблема обедающих философов полезна для моделирования процессов, соревнующихся за монопольный доступ к ограниченному количеству ресурсов, например к устройствам ввода/вывода. Другой известной задачей является проблема читателей и писателей [17], моделирующая доступ к базе данных. Представьте себе базу данных бронирования билетов на самолет, к которой пытается получить доступ множество клиентов. Можно разрешить одновременное считывание данных из базы, но если процесс записывает информацию в базу, доступ остальных процессов должен быть прекращен, даже доступ на чтение. Как запрограммировать читателей и писателей? Одно из решений представлено в листинге 2.10.

Листинг 2.10. Решение проблемы читателей и писателей

```
typedef int semaphore;          /* Воспользуйтесь своим воображением */
semaphore mutex = 1;            /* Контроль доступа к rc */
semaphore db = 1;               /* Контроль доступа к базе данных */
int rc = 0;                     /* Количество процессов, читающих или желающих */
                                /* читать */

void reader(void)
{
      while (TRUE) {             /* Повторять до бесконечности */
         down(&mutex);           /* Получение монопольного доступа к rc */
         rc = rc+1;              /* Одним читающим процессом больше */
         if (rc == 1) down(&db); /* Если этот читающий процесс - первый... */
         up(&mutex);             /* Отказ от монопольного доступа к rc */
         read_data_base();       /* Доступ к данным */
         down(&mutex);           /* Получение монопольного доступа к rc */
```

```
        rc = rc-1;              /* Одним читающим процессом меньше */
        if (rc == 0) up(&db); /* Если этот читающий процесс -/*
                              /* последний... */
        up(&mutex);           /* Отказ от монопольного доступа к rc */
        use_data_read();      /* Вне критической области */
    }
}

void writer(void)
{
    while (TRUE) {            /* Повторять до бесконечности */
        think_up_data();      /* Вне критической области */
        down(&db);            /* Получение монопольного доступа */
        write_data_base();    /* Запись данных */
        up(&db);              /* Отказ от монопольного доступа */
    }
}
```

Первый читающий процесс выполняет операцию down на семафоре db, чтобы получить доступ к базе. Последующие читатели просто увеличивают значение счетчика rc. По мере уменьшения читающих из базы значение счетчика уменьшается, и последний читающий процесс выполняет на семафоре db операцию up, позволяя блокированному пишущему процессу получить доступ к базе.

В этом решении один момент требует комментариев. Представьте, что в то время как один читатель уже пользуется базой, другой читатель запрашивает к ней доступ. Доступ разрешается, поскольку читающие процессы друг другу не мешают. Доступ разрешается и третьему, и последующим читателям.

Затем доступ запрашивает пишущий процесс. Запрос отклонен, поскольку пишущим процессам необходим монопольный доступ, и пишущий процесс приостанавливается. Пока в базе есть хотя бы один активный читающий процесс, доступ остальным читателям разрешается, а они все приходят и приходят. Если, предположим, новый читающий процесс запрашивает доступ каждые 2 с, а для работы с базой ему надо 5 с, то пишущий процесс никогда в базу не попадет.

Чтобы избежать такой ситуации, нужно немного изменить программу: если пишущий процесс ждет доступа к базе, новый читающий процесс доступа не получает, а становится в очередь за пишущим процессом. Теперь пишущему процессу нужно подождать, пока базу покинут уже находящиеся в ней читающие процессы, но не нужно пропускать вперед читающие процессы, пришедшие к базе после него. Недостаток этого решения заключается в снижении производительности, вызванном смягчением конкуренции. В [17] представлено решение, в котором пишущим процессам предоставляется более высокий приоритет.

2.3.3. Проблема спящего брадобрея

Действие еще одной классической проблемной ситуации межпроцессного взаимодействия разворачивается в парикмахерской. В зале есть один брадобрей, его кресло и n стульев для посетителей. Если желающих воспользоваться его услугами нет, цирюльник сидит в своем кресле и спит (рис. 2.8). Когда в парикмахер-

скую приходит клиент, он должен разбудить спящего. Если клиент приходит и видит, что брадобрей занят, он либо садится на стул (если есть место), либо уходит (если места нет). Необходимо запрограммировать брадобрея и посетителей так, чтобы избежать состояния состязания. У этой задачи существует много аналогов в сфере массового обслуживания, это, например информационная служба, обрабатывающая одновременно ограниченное количество запросов, с компьютеризированной системой ожидания для запросов.

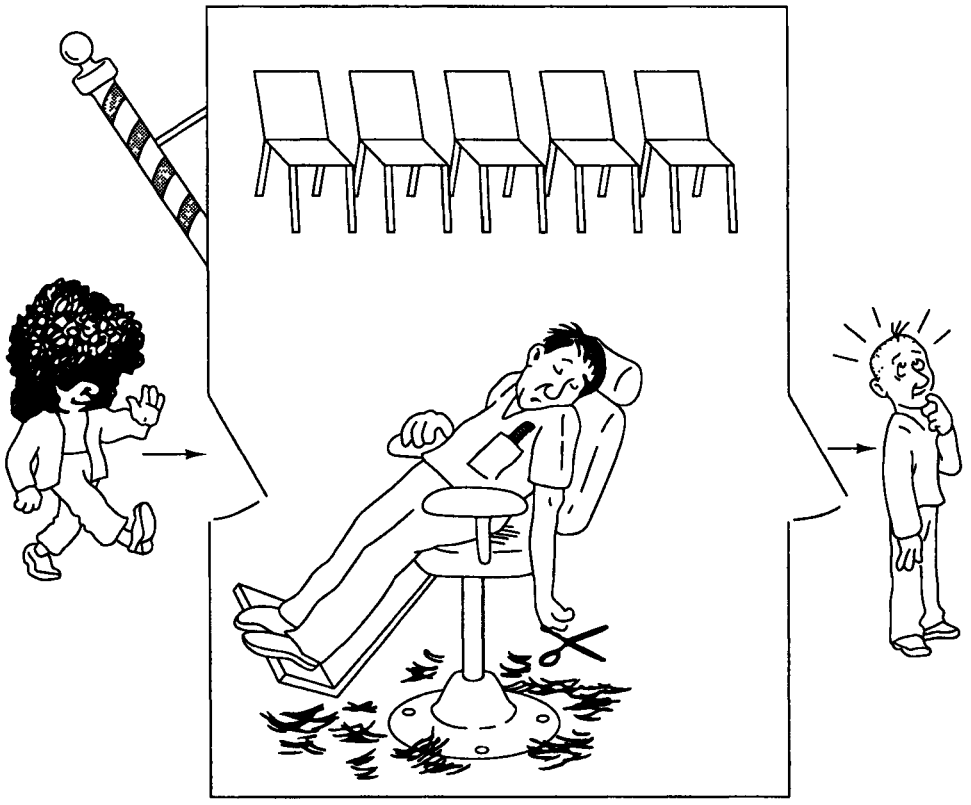


Рис. 2.8. Спящий брадобрей

В предлагаемом решении используются три семафора: customers, для подсчета ожидающих посетителей (клиент, сидящий в кресле брадобрея, не учитывается — он уже не ждет); barbers, количество брадобреев (0 или 1), простаивающих в ожидании клиента, и mutex для реализации взаимного исключения. Также используется переменная waiting, предназначенная для подсчета ожидающих посетителей. Она является копией переменной customers. Присутствие в программе этой переменной связано с тем фактом, что прочитать текущее значение семафора невозможно. В этом решении посетитель, заглядывающий в парикмахерскую,

должен сосчитать количество томящихся клиентов. Если посетителей меньше, чем стульев, новый посетитель остается, в противном случае он уходит.

Решение представлено в листинге 2.11. Когда брадобрей приходит утром на работу, он выполняет процедуру barber, блокируясь на семафоре customers, поскольку значение семафора равно 0. Затем брадобрей засыпает, как показано на рис. 2.8, и спит, пока не придет первый клиент.

Листинг 2.11. Решение проблемы спящего брадобрея

```
#define CHAIRS 5                /* Количество стульев для посетителей */

typedef int semaphore;          /* Догадайтесь сами */

semaphore customers = 0;        /* Количество ожидающих посетителей */
semaphore barbers = 0;          /* Количество брадобреев, ждущих клиентов */
semaphore mutex = 1;            /* Для взаимного исключения */
int waiting = 0;                /* Ожидающие (не обслуживаемые) посетители */

void barber(void)
{
     while (TRUE) {
        down(&customers);       /* Если посетителей нет, уйти в состояние ожидания */
        down(&mutex);           /* Запрос доступа к waiting */
        waiting = waiting - 1; /* Уменьшение числа ожидающих посетителей */
        up(&barbers);           /* Один брадобрей готов к работе */
        up(&mutex);             /* Отказ от доступа к waiting */
        cut_hair();             /* Клиента обслуживают (вне критической области)*/
     }
}

void customer(void)
{
     down(&mutex);              /* Вход в критическую область */
     if (waiting < CHAIRS) {    /* Если свободных стульев нет, придется уйти */
        waiting = waiting + 1; /* Увеличение числа ожидающих посетителей */
        up(&customers);         /* При необходимости разбудить брадобрея */
        up(&mutex);             /* Отказ от доступа к waiting */
        down(&barbers);         /* Если брадобрей занят, уйти в состояние ожидания */
        get_haircut();          /* Клиента усаживают и обслуживают */
     } else {
        up(&mutex);             /* Много посетителей, из парикмахерской придется уйти */
     }
}
```

Приходя в парикмахерскую, посетитель выполняет процедуру customer, запрашивая доступ к mutex для входа в критическую область. Если вслед за ним появится еще один посетитель, ему не удастся что-либо сделать, пока первый желающий постричься не освободит доступ к mutex. Затем посетитель проверяет наличие свободных стульев, в случае неудачи освобождает доступ к mutex и уходит.

Если свободный стул есть, посетитель увеличивает значение целочисленной переменной waiting. Затем он выполняет процедуру up на семафоре customers, тем самым активизируя поток брадобрея. В этот момент оба — посетитель и брадобрей — активны. Когда посетитель освобождает доступ к mutex, брадобрей

захватывает семафор, проделывает некоторые служебные операции и начинает стричь клиента.

По окончании стрижки посетитель выходит из процедуры и покидает парикмахерскую. В отличие от предыдущих программ цикла посетителя нет, поскольку каждого посетителя стригут только один раз. Цикл брадобрея существует, и брадобрей пытается найти следующего посетителя. Если ему это удается, он стрижет следующего посетителя, в противном случае брадобрей засыпает.

Стоит отметить, что, несмотря на отсутствие передачи данных в проблеме читателей и писателей и в проблеме спящего брадобрея, обе эти проблемы относятся к проблемам межпроцессного взаимодействия, поскольку требуют синхронизации нескольких процессов.

2.4. Планирование

Когда компьютер работает в многозадачном режиме, на нем могут быть активными несколько процессов, пытающихся одновременно получить доступ к процессору. Эта ситуация возникает при наличии двух и более процессов в состоянии готовности. Если доступен только один процессор, необходимо выбирать между процессами. Отвечающая за это часть операционной системы называется *планировщиком*, а используемый алгоритм — *алгоритмом планирования*. Рассмотрению задач, связанных с планированием, посвящены следующие разделы.

Давным-давно, во времена систем пакетной обработки, перфокарт на магнитной ленте в качестве устройства ввода, алгоритм планирования был прост: запустить следующую задачу на ленте. С появлением систем с разделением времени алгоритм планирования усложнился, поскольку теперь несколько задач одновременно ожидали обслуживания. На некоторых мэйнфреймах до сих пор совмещаются системы пакетной обработки и службы разделения времени. Даже на персональных компьютерах одновременно могут работать несколько пользовательских процессов, не говоря уже о фоновых службах, таких как сетевые или почтовые демоны.

Чтобы разработать алгоритм планирования, необходимо иметь представление о том, что должен делать хороший алгоритм. Некоторые задачи зависят от среды (системы пакетной обработки, интерактивные или реального времени), но есть одинаковые во всех системах, мы рассмотрим их ниже.

1. Равноправие — предоставление каждому процессу справедливой доли процессорного времени.
2. Использование процессора — поддержка постоянной занятости процессора.
3. Время отклика — быстрая реакция на запросы.
4. Оборотное время — минимизация времени, затрачиваемого на ожидание обслуживания и обработку задачи.
5. Пропускная способность — максимальное количество задач в час.

Несложное рассуждение показывает, что некоторые из перечисленных целей противоречат друг другу.

При всех обстоятельствах необходимо справедливое распределение процессорного времени. Сопоставимые процессы должны получать сопоставимое обслуживание. Выделить одному процессу намного больше ресурсов процессора, чем другому, эквивалентному, несправедливо. Разумеется, с различными категориями процессов следует обращаться весьма по-разному. Сравните, например, задачи обеспечения безопасности и начисления заработной платы в компьютерном центре атомной электростанции. Интерактивным пользователям, для которых важно небольшое время отклика, наверняка бы понравилось, если пакетные задачи не запускались бы вообще (ну, разве что, кроме времени с 3 часов ночи до 6 утра, когда все такие пользователи спят). Но тем, кто ставит задачи на обработку, такая идея придется не по вкусу, так как она нарушает пункт 4. Вообще говоря, можно показать, что алгоритм планирования, ориентированный на какой-то один класс задач, всегда плохо подходит для других классов. В конце концов, процессорное время ограничено. Чтобы дать одному пользователю больше, другому придется дать меньше. Такова жизнь.

1. Осложняет работу планировщика тот факт, что каждый процесс уникален и непредсказуем. Одни проводят много времени, ожидая окончания ввода/вывода, а другие, если дать им шанс, будут загружать процессор часами. Когда планировщик запускает какой-то процесс, он не может знать, сколько времени пройдет до его прерывания, будь причиной последнего ввод/вывод, семафор или что-либо другое. Во избежание таких злоупотреблений во всех компьютерах имеется встроенный электронный таймер, периодически вызывающий прерывание. Обычно это происходит 50 или 60 раз в секунду (50 или 60 Гц), но на многих компьютерах операционная система может произвольным образом менять эту частоту. При каждом прерывании операционная система решает, должен ли текущий процесс продолжить работу, или он получил уже достаточно процессорного времени и можно отдать процессор другому претенденту.

2. Стратегия, при которой приостанавливаются процессы, логически способные выполняться, называется стратегией с вытесняющим прерыванием, в противовес стратегии выполнения до завершения (иногда называемой невытесняющей стратегией), применявшейся в ранних пакетных системах. Как будет показано в этой главе, процесс может быть приостановлен в любой момент, без предупреждения. Это приводит к возникновению ситуаций состязания, и, как следствие, делает необходимыми семафоры, мониторы, сообщения и прочие запутанные механизмы. С другой стороны, если позволять процессу работать столько, сколько ему нужно, то, скажем, процесс, вычисляющий число π с точностью до миллиона знаков после запятой, надолго блокировал бы работу всех остальных.

3. Поэтому, хотя невытесняющие алгоритмы и просты в реализации, для многоцелевых систем, на которых могут одновременно работать несколько пользователей, этот механизм не подходит. И напротив, этот механизм может быть пригодным для специализированной системы, такой как сервер баз данных, где основной процесс вправе запускать дочерние и выполнять

их до тех пор, пока они не завершатся или не заблокируются. Основное отличие подобных систем от многоцелевых в том, что здесь все процессы находятся под присмотром основного, знающего, чем занимается каждый из потомков и сколько времени это должно занять.

2.4.1. Циклическое планирование

Рассмотрим некоторые основные алгоритмы планирования. Одним из наиболее старых, простых, справедливых и часто используемых является алгоритм *циклического планирования* (*карусельного*). Каждому процессу предоставляется некоторый интервал времени процессора, так называемый *квант* времени. Если к концу кванта времени процесс все еще работает, он прерывается, а управление передается другому процессу. Разумеется, если процесс блокируется или прекращает работу раньше, переход управления происходит в этот момент. Реализация циклического планирования проста. Планировщику нужно всего лишь поддерживать список процессов в состоянии готовности согласно рис. 2.9, *а*. Когда процесс исчерпал свой лимит времени, он отправляется в конец списка (рис. 2.9, *б*).

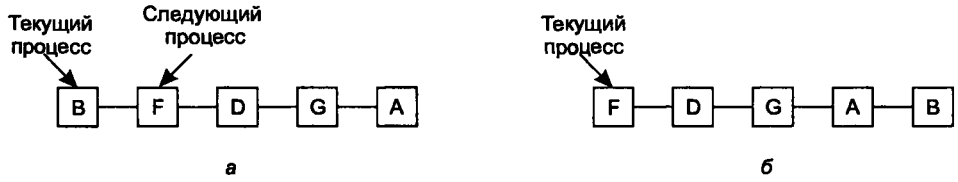


Рис. 2.9. Циклическое планирование: *а* — список процессов в состоянии готовности; *б* — список процессов в состоянии готовности после того, как процесс B исчерпал свой квант времени

Единственным интересным моментом этого алгоритма является длина кванта. Переключение с одного процесса на другой занимает некоторое время — необходимо сохранить и загрузить регистры и карты памяти, обновить таблицы и списки, сохранить и перезагрузить кэш памяти и т. п. Представим, что *переключение процессов* или *переключение контекста*, как его иногда называют, занимает 1 мс, включая переключение карт памяти, перезагрузку кэша и т. п. Пусть размер кванта установлен в 4 мс. В таком случае 20 % процессорного времени уйдет на администрирование — это слишком много.

Для увеличения эффективности назначим размер кванта, скажем, 100 мс. Теперь пропадает только 1 % времени. Но представьте, что будет в системе с разделением времени, если 10 пользователей одновременно нажмут клавишу возврата каретки. В список будет занесено 10 процессов. Если процессор был свободен, первый процесс будет запущен немедленно, второму придется ждать 100 мс и т. д. Последнему процессу, возможно, придется ждать целую секунду, если все остальные не блокируются за время кванта. Большинству пользователей секундная задержка вряд ли понравится.

Важен и тот фактор, что если установленное значение кванта больше среднего интервала работы процессора, переключение процессов будет происходить редко. Напротив, большинство процессов будут совершать блокирующую операцию прежде, чем истечет длительность кванта, вызывая переключение процессов. Устранение принудительных переключений процессов улучшает производительность системы, так как переключения процессов будут происходить только тогда, когда это логически необходимо, то есть когда процесс заблокировался и не может продолжать работу.

Вывод можно сформулировать следующим образом: слишком малый квант приведет к частому переключению процессов и небольшой эффективности, но слишком большой квант может привести к медленному реагированию на короткие интерактивные запросы. Значение кванта около 20–50 мс часто является разумным компромиссом.

2.4.2. Планирование согласно приоритетам

Алгоритм карусельного планирования базируется на важном допущении о том, что все процессы равнозначны. В ситуации компьютера с большим числом пользователей это может быть не так. Например, в университете прежде всего должны обслуживаться деканы, затем профессора, секретари, уборщицы и лишь потом студенты. Необходимость принимать во внимание подобные внешние факторы приводит к *планированию согласно приоритетам*. Основная идея проста: каждому процессу присваивается свой ранг в табеле и управление передается готовому к работе процессу с самым высоким приоритетом.

Даже на персональном компьютере с одним пользователем могут выполняться несколько процессов, отдельные из которых являются более важными, чем другие. Демон, отвечающий за пересылку электронной почты в фоновом режиме, имеет более низкий приоритет, чем процесс, отображающий на экране видеофильм в реальном времени.

Чтобы предотвратить бесконечную работу процессов с высоким приоритетом, планировщик может уменьшать приоритет процесса с каждым тактом часов (то есть при каждом прерывании по таймеру). Если в результате приоритет текущего процесса окажется ниже приоритета следующего процесса, произойдет переключение. Возможно предоставление каждому процессу максимального отрезка времени работы. Как только время кончилось, управление передается следующему по значимости процессу.

Приоритеты процессам могут присваиваться статически или динамически. На военной базе процессу, запущенному генералом, присваивается приоритет 100, полковником — 90, майором — 80, капитаном — 70, лейтенантом — 60 и т. д. А в коммерческом компьютерном центре выполнение заданий с высоким приоритетом может стоить 100 долларов в час, со средним — 75, с низким — 50. В системе UNIX есть команда nice, позволяющая пользователю добровольно снизить приоритет своих процессов, чтобы быть любезным по отношению к остальным пользователям. Этой командой никто никогда не пользуется.

Система вправе динамически назначать приоритеты для достижения своих целей. Например, некоторые процессы сильно ограничены возможностями устройств ввода/вывода и большую часть времени проводят в ожидании завершения соответствующих операций. Когда бы ни потребовался процессор такому процессу, его следует немедленно предоставить, чтобы процесс мог начать обрабатывать следующий запрос ввода/вывода, который будет выполняться параллельно с вычислениями другого процесса. Если заставить процесс, ограниченный возможностями устройств ввода/вывода, длительное время ждать доступа к процессору, он будет неоправданно долго находиться в памяти. Простой алгоритм обслуживания процессов, сдерживаемых возможностями устройств ввода/вывода, состоит в установке приоритета, равного $1/f$, где f — часть использованного в последний раз кванта. Процесс, утилизировавший всего 1 мс из 50 мс кванта, получит приоритет 50, процесс, использовавший 25 мс, получит приоритет 2, а процесс, задействовавший весь квант, получит приоритет 1.

Часто бывает удобно сгруппировать процессы в классы по приоритетам и использовать планирование согласно приоритетам среди классов, но циклическое планирование внутри каждого класса. На рис. 2.10 представлена система с четырьмя классами приоритетов. Алгоритм планирования выглядит следующим образом: пока в классе 4 есть готовые к запуску процессы, они запускаются один за другим согласно алгоритму циклического планирования, и каждому отводится квант времени. При этом классы с более низким приоритетом не будут их беспокоить. Если в классе 4 нет готовых к запуску процессов, запускаются процессы класса 3 и т. д. Если приоритеты постоянны, до процессов класса 1 процессор может не дойти никогда.

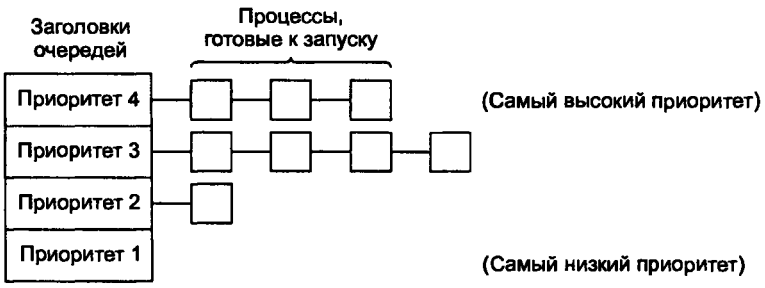


Рис. 2.10. Алгоритм планирования с четырьмя классами приоритетов

2.4.3. Планирование с несколькими очередями

Один из первых «приоритетных» планировщиков был реализован в системе CTSS (Compatible Time-Shared System — совместимая система с разделением времени) [75]. Основной проблемой системы CTSS стало слишком медленное переключение процессов, поскольку в памяти компьютера IBM 7094 мог находиться только один процесс. Каждое переключение означало выгрузку текущего процесса на диск и считывание нового процесса с диска. Разработчики CTSS быстро сообра-

зили, что эффективность будет выше, если процессам, ограниченным возможностями процессора, выделять больший квант времени, чем если предоставлять им небольшие кванты, но часто. С одной стороны, это уменьшит количество перемещений из памяти на диск, а с другой — приведет к ухудшению времени отклика, как мы уже видели. В результате было разработано решение с классами приоритетов. Процессам класса с высшим приоритетом выделялся один квант, процессам следующего класса — два кванта, следующего — четыре кванта и т. д. Когда процесс использовал все отведенное ему время, он переходил на класс ниже.

В качестве примера рассмотрим процесс, которому необходимо производить вычисления в течение 100 квантов. Вначале ему будет предоставлен один квант, затем процесс будет сброшен на диск. В следующий раз ему достанется 2 кванта, затем 4, 8, 16, 32, 64, хотя из 64 он использует только 37. В этом случае понадобятся всего 7 «перекачек» (включая первоначальную загрузку) вместо 100, которые понадобились бы в случае циклического алгоритма. Помимо того, по мере погружения в очередь приоритетов процесс будет все реже запускаться, в пользу более коротких процессов.

Чтобы дать процессу, который при запуске считался «долгоиграющим», но позже стал интерактивным, не погибнуть в недрах планирования, была разработана следующая стратегия. Как только с терминала приходит сигнал возврата каретки, процесс, соответствующий этому терминалу, переносится в класс высшего приоритета, поскольку предполагается, что он становится интерактивным. Однажды пользователь, запустивший процесс, сильно стесненный возможностями процессора, обнаружил, что бездумное нажимание клавиши Enter существенно уменьшает время отклика, и рассказал об этом друзьям. Мораль этой истории такова: осуществить задуманное на практике гораздо сложней, чем в теории.

Для разделения процессов по классам используются также многие другие алгоритмы. Например, в системе XDS 940, разработанной в Беркли [79], было четыре класса приоритетов, называвшихся: терминал, ввод/вывод, короткий квант и длинный квант. Когда запускался процесс, ожидающий вывода на терминал, он перемещался в класс высшего приоритета (терминал). Когда снималась блокировка процесса, ожидавшего доступа к диску, он переходил во второй класс. Если к концу отведенного времени процесс все еще работал, он сначала зачислялся в третий класс. Если процесс слишком много раз полностью задействовал свой квант времени, не блокируясь на терминале или другом устройстве ввода/вывода, он перемещался в последний класс. Этот метод практикуется во многих системах для предоставления преимущества интерактивным процессам по сравнению с фоновыми.

2.4.4. «Самый короткий процесс — следующий»

Описанные выше алгоритмы применимы, по большей части, к интерактивным системам. Теперь мы рассмотрим алгоритм, пригодный для планирования пакетных задач, время выполнения которых известно заранее. Рассмотрим еще один алгоритм без переключений для систем пакетной обработки, предполагающий, что временные отрезки работы известны заранее. Например, служащие страхо-

вой компании могут довольно точно предсказать, сколько времени займет обработка пакета из 1000 исков, поскольку они делают это каждый день. Если в очереди есть несколько одинаково важных задач, планировщик выбирает *первой самую короткую задачу*. Посмотрите на рис. 2.11. У нас есть четыре задачи: A, B, C и D, со временем выполнения 8, 4, 4 и 4 мин соответственно. Если мы запустим их в данном порядке, оборотное время задачи A будет 8 мин, B — 12 мин, C — 16 мин, D — 20 мин и среднее время будет равно 14 мин.

а *б*

Рис. 2.11. Пример алгоритма планирования «Кратчайшая задача — первая»: *а* — запуск четырех задач в исходном порядке; *б* — запуск в соответствии с алгоритмом

Запустим задачи в соответствии с алгоритмом, как показано на рис. 2.11, *б*. Теперь значения оборотного времени составляют 4, 8, 12 и 20 мин соответственно, а среднее время равно 11 мин. Алгоритм оптимизирует задачу. Рассмотрим четыре процесса со временем выполнения a, b, c и d. Первая задача выполняется за время a, вторая — за время $a + b$ и т. д. Среднее оборотное время будет равно $(4a + 3b + 2c + d)/4$. Очевидно, что вклад a в среднее время больше, чем всех остальных интервалов времени, поэтому первой должна выполняться самая быстрая задача, а последней — с наибольшей длительностью, вносящая вклад, равный собственному оборотному времени. Точно так же рассматривается система из любого количества задач.

Поскольку рассматриваемый алгоритм минимизирует среднее оборотное время в системах пакетной обработки, хотелось бы использовать его и в интерактивных системах. В известной степени это возможно. Интерактивные процессы чаще всего следуют схеме «ожидание команды, исполнение команды, ожидание команды, исполнение команды...» Если рассматривать выполнение каждой команды как отдельную задачу, можно свести к минимуму общее среднее время отклика, запуская первой самую короткую задачу. Единственная проблема состоит в том, чтобы понять, какой из ожидающих процессов самый короткий.

Один из методов привлекает к оценке «длину» процесса, базируясь на предыдущем его поведении. При этом запускается процесс, у которого оцененное время самое маленькое. Допустим, что предполагаемое время исполнения команды равно T_0 и предполагаемое время следующего запуска равно T_1. Можно улучшить оценку, взяв взвешенную сумму этих времен $aT_0 + (1 - a) \times T_1$. Выбирая соответствующее значение a, мы можем заставить алгоритм оценки быстро забывать о предыдущих запусках или, наоборот, помнить о них в течение долгого времени. Взяв $a = 1/2$, мы получим серию оценок:

$$T_0,\ T_0/2 + T_1/2,\ T_0/4 + T_1/4 + T_2/2,\ T_0/8 + T_1/8 + T_2/4 + T_3/2.$$

После трех запусков вес T_0 в оценке уменьшится до 1/8.

Метод оценки следующего значения серии через взвешенное среднее предыдущего значения и предыдущей оценки часто называют *старением*. Этот метод применим во многих ситуациях, где необходима оценка по предыдущим значениям. Проще всего реализовать старение при $a = 1/2$. На каждом шаге нужно всего лишь добавить к текущей оценке новое значение и разделить сумму пополам (со сдвигом вправо на 1 бит).

Следует отметить, что эта схема работает лишь в случае одновременного наличия задач. В качестве контрпримера можно рассмотреть пять задач, A, B, C, D и E, причем первые две доступны сразу же, а три оставшиеся — еще через три минуты. Время выполнения этих задач составляет 2, 4, 1, 1 и 1 мин соответственно.

Вначале можно выбрать только A или B, поскольку остальные недоступны. Если руководствоваться алгоритмом «Кратчайшая задача — первая», задачи будут запущены в следующем порядке: A, B, C, D, E и среднее оборотное время составит 4,6 мин. Если же запустить их в порядке B, C, D, E, A, оно будет равно 4,4 мин.

2.4.5. Гарантированное планирование

Принципиально другим подходом к планированию является предоставление пользователям реальных обещаний и затем их выполнение. Вот одно обязательство, которое легко произнести и легко выполнить: если вместе с вами процессором пользуются n пользователей, вам будет предоставлено $1/n$ мощности процессора. И в системе с одним пользователем и n запущенными процессорами каждому достанется $1/n$ циклов процессора.

Чтобы сдержать это обещание, система должна отслеживать распределение процессора между процессами с момента создания каждого процесса. Затем система рассчитывает количество ресурсов процессора, на которое процесс имеет право, например время с момента создания, деленное на n. Теперь можно сосчитать отношение времени, предоставленного процессу, к времени, на которое он имеет право. Полученное значение 0,5 означает, что процессу выделили только половину положенного, а 2,0 говорит о том, что процессу досталось в два раза больше его нормы. Далее запускается процесс, у которого это отношение наименьшее, пока оно не станет больше, чем у его ближайшего соседа.

2.4.6. Лотерейное планирование

Хотя идея обещаний пользователям и их выполнения хороша, но ее трудно реализовать. Для более простой реализации предсказуемых результатов применяется другой алгоритм, называемый *лотерейным планированием* [86].

В его основе лежит раздача процессам «лотерейных билетов» на доступ к различным ресурсам, в том числе и к процессору. Когда планировщику необходимо принять решение, выбирается случайным образом лотерейный билет, и его обладатель получает доступ к ресурсу. Что касается доступа к процессору, «лотерея» может происходить 50 раз/с, и победитель получает 20 мс времени процессора.

Если перефразировать Джорджа Оруэлла: «Все процессы равны, но некоторые равнее других». Более важным процессам можно раздать дополнительные билеты, чтобы увеличить вероятность выигрыша. Если тираж всего — 100 билетов и 20 из них находятся у одного процесса, то ему достанется 20 % времени процессора. В отличие от планирования по приоритетам, где очень трудно оценить, что означает, скажем, приоритет 40, в лотерейном планировании все очевидно. Каждый процесс получит процент ресурсов, примерно равный проценту имеющихся у него билетов.

Лотерейное планирование характеризуется несколькими интересными свойствами. Например, если при создании процессу достается несколько билетов, то уже в следующем розыгрыше его шансы на выигрыш пропорциональны количеству билетов. Другими словами, лотерейное планирование обладает высокой «отзывчивостью».

Взаимодействующие процессы могут при необходимости обмениваться билетами. Так, если клиентский процесс посылает сообщение серверному процессу и затем блокируется, он вправе передать все свои билеты серверному процессу, чтобы увеличить шанс запуска сервера. Когда серверный процесс заканчивает работу, ему ничего не стоит вернуть все билеты обратно. Действительно, если клиентов нет, то серверу билеты вовсе не нужны.

Лотерейное планирование позволяет решать задачи, которые не решить с помощью других алгоритмов. В качестве примера приведем видеосервер, на котором несколько процессов передают своим клиентам потоки видеоинформации с различной частотой кадров. Предположим, что процессы используют частоты 10, 20 и 25 кадров/с. Предоставив процессам соответственно 10, 20 и 25 билетов, можно реализовать загрузку процессора в желаемой пропорции 10:20:25.

2.4.7. Планирование в системах реального времени

В системах *реального времени*, что и следовало ожидать, существенную роль играет время. Чаще всего одно или несколько внешних физических устройств генерируют входные сигналы, и компьютер должен адекватно на них реагировать в течение заданного временного интервала. Например, компьютер в проигрывателе компакт-дисков получает биты от дисковода и должен за очень маленький промежуток времени преобразовать их в музыку. Если процесс преобразования будет слишком долгим, звук окажется искаженным. Подобные системы также используются для наблюдения за пациентами в палатах интенсивной терапии, в качестве автопилота самолета, для управления роботами на автоматизированном производстве. В любом из этих случаев запоздалая реакция ничуть не лучше, чем отсутствие реакции.

Системы реального времени делятся на *жесткие системы реального времени*, что означает наличие жестких сроков для каждой задачи (в них обязательно надо укладываться), и *гибкие системы реального времени*, в которых нарушения временного графика нежелательны, но допустимы. В обоих случаях реализуется

разделение программы на несколько процессов, каждый из которых предсказуем. Эти процессы чаще всего бывают короткими и завершают свою работу в течение секунды. Когда появляется внешний сигнал, именно планировщик должен обеспечить соблюдение графика.

Внешние события, на которые система должна реагировать, можно разделить на *периодические* (возникающие через регулярные интервалы времени) и *непериодические* (возникающие непредсказуемо). Возможно наличие нескольких периодических потоков событий, которые система должна обрабатывать. В зависимости от времени, затрачиваемого на обработку каждого из событий, может оказаться, что система не в состоянии своевременно обработать их все. Если в систему поступает m периодических событий, событие с номером i поступает с периодом P_i и на его обработку уходит C_i секунд работы процессора, своевременную обработку всех потоков обеспечивает только выполнение условия

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1.$$

Системы реального времени, удовлетворяющие этому условию, называются *поддающимися планированию* или *планируемыми*.

В качестве примера рассмотрим гибкую систему реального времени с тремя периодическими сигналами с периодами в 100, 200 и 500 мс соответственно. Если на обработку этих сигналов уходит, в том же порядке, 50, 30 и 100 мс, система является поддающейся планированию, поскольку 0,5 + 0,15 + 0,2 < 1. Даже при добавлении четвертого сигнала с периодом в 1 с системой все равно можно будет управлять через планирование, пока время обработки сигнала не превысит 150 мс. В этих расчетах существенным является предположение, что время переключения между процессами пренебрежимо мало.

Алгоритмы планирования для систем реального времени бывают как статическими, так и динамическими. В первом случае все решения планирования принимаются заранее, еще до запуска системы. Во втором случае решения выносятся по ходу дела. Рассмотрим в общих чертах несколько динамических алгоритмов планирования реального времени. Классический вариант носит название *алгоритма постоянной скорости* (rate monotonic algorithm, Liu и Layland, 1973). В нем процессам предварительно назначаются приоритеты, пропорциональные частоте возникновения переключений. Например, если один процесс работает каждые 50 мс, а второй каждые 100 мс, то первый может получить приоритет 50, а второй 10. В момент запуска планировщик всегда инициирует процесс, имеющий больший приоритет, прерывая, если нужно, текущий. Авторы алгоритма доказали, что он является оптимальным.

Другой популярный алгоритм планирования реального времени называется *планированием по ближайшему крайнему сроку* (earliest deadline first). Когда обнаруживается событие, соответствующий ему процесс заносится в список готовых. Процессы в этом списке упорядочены по предельному сроку выполнения, который для периодических событий равен следующему моменту возникновения события. Планировщик запускает первый процесс из списка, для этого процесса предельный срок наступит первым.

В третьем случае для каждого процесса сначала рассчитывается, сколько у него есть времени в запасе, этот параметр называется *неопределенностью* (буквально — расхлябанностью, laxity) процесса. Так, если на выполнение процесса требуется 200 мс времени, а выполнен он должен быть за 250 мс, неопределенность равна 50 мс. В алгоритме планирования с *наименьшей неопределенностью*), выбирается самый «собранный» процесс, с мимимумом времени, оставляемым для резерва.

Хотя, теоретически, с помощью одного из перечисленных алгоритмов можно превратить операционную систему общего назначения в систему реального времени, на практике оказывается, что накладные расходы на переключение контекстов в таких системах столь велики, что производительность реального времени достигается только для процессов с очень мягкими ограничениями. Вследствие этого для работы в реальном времени обычно применяются специальные операционные системы, обладающие некоторыми важными особенностями. Обычно это небольшой размер, быстрое выполнение прерываний, быстрое переключение контекстов, малая протяженность интервала времени, когда заблокированы прерывания, и возможность поддерживать несколько таймеров миллисекундной или микросекундной точности.

2.4.8. Двухуровневый механизм

До этого момента мы полагали, что все запускаемые процессы находятся в основной памяти. Если же памяти недостаточно, некоторые процессы будут размещаться на диске, частично или полностью. Эта ситуация имеет непосредственное отношение к планированию, так как переключение на процесс, расположенный на диске, гораздо более затратно, чем переключение на процесс в памяти.

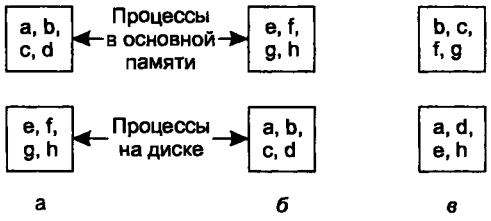


Рис. 2.12. Двухуровневый планировщик решает задачи планирования процессов в памяти и переноса их между памятью и областью подкачки (диском): *а* — загружается некоторое подмножество готовых к работе процессов; *б* — выгрузка старых и загрузка новых процессов; *в* — переключение контекста в основной памяти

Двухуровневый планировщик реализует практический способ работы с процессами, которые могут находиться в файле подкачки. Прежде всего, в основную память загружается некоторое подмножество готовых к работе процессов, как показано на рис. 2.12, *а*. После этого планировщик некоторое время выбирает процессы только из этого подмножества. Периодически запускается планировщик более высокого уровня, выгружающий из памяти процессы, которые нахо-

дились там достаточно долго, и помещающий на их место процессы с диска, как показано на рис. 2.12, *б*. После этого вновь вступает в дело планировщик нижнего уровня, переключающий только процессы в основной памяти.

Среди критериев, привлекаемых планировщиком, есть следующие:

1. Сколько времени прошло с тех пор, как процесс был выгружен на диск или загружен с диска?
2. Сколько времени процесс уже использовал процессор?
3. Каков «размер» процесса (маленькие процессы не мешают)?
4. Какова актуальность процесса?

Когда идет разговор о «планировщике», обычно имеется в виду именно *планировщик процессора*. В таком планировщике применяется любой подходящий к ситуации алгоритм, как с прерыванием, так и без. Некоторые из этих алгоритмов мы уже рассмотрели, а с другими еще ознакомимся.

2.4.9. Политика и механизм планирования

Вплоть до настоящего момента мы подразумевали, что процессы в системе принадлежат различным пользователям и конкурируют за доступ к процессору. Чаще всего именно так все и выглядит, но возможна ситуация, в которой под присмотром одного процесса выполняются много дочерних процессов. Например, у процесса, управляющего базой данных, может быть много дочерних процессов, обрабатывающих отдельные запросы или выполняющих конкретные функции (анализ запроса, доступ к диску и т. п.). Вполне вероятно, что родительский процесс лучше представляет, какой из его дочерних процессов более важен (или для которого фактор времени более критичен), а какой — менее. К сожалению, ни один из рассмотренных нами планировщиков не в силах получать информацию от пользовательских процессов и учитывать ее при принятии решений планирования. В результате планировщики редко выносят оптимальное решение.

Преодолеть проблему можно, выделив в ней *механизм планирования* и *политику планирования*. Таким образом, мы реализуем ситуацию, в которой алгоритм планирования будет каким-либо образом параметризован, но параметры вправе быть заданы процессом пользователя. Обратимся еще раз к примеру базы данных. Пусть ядро использует алгоритм приоритетного планирования, но существует системный запрос, посредством которого процесс может устанавливать (и менять) приоритеты своих дочерних процессов. В этом случае родительский процесс имеет право управления планированием дочерних процессов, хотя сам он планирования не осуществляет. Механизм определяется ядром, но политику задает процесс пользователя.

2.5. Обзор процессов в MINIX

Сейчас, завершив изучение принципов управления процессами, взаимодействия между ними и алгоритмы планирования, мы можем приступить непосредственно

к их реализации в MINIX. В отличие от UNIX, где ядро представляет собой монолитную программу, не разбитую на модули, ядро MINIX само является набором процессов, взаимодействующих с пользователем и между собой при помощи единственного примитива — передачи сообщений. Такой подход дает более гибкую и модульную структуру, позволяя, например, легко заменить всю файловую систему другой, не пересобирая ядро.

2.5.1. Внутренняя структура MINIX

Начнем наше изучение MINIX с обзора системы с высоты птичьего полета. Структурно система разбита на четыре уровня, каждый из которых выполняет строго определенные функции. Эти уровни показаны на рис. 2.13.

Нижний уровень обрабатывает прерывания и ловушки, занимается планированием, предоставляет вышележащим уровням модель независимых, логически упорядоченных процессов, обменивающихся информацией при помощи сообщений. Код данного уровня отвечает за две основные функции. Прежде всего, это перехват сообщений и ловушки, сохранение и восстановление регистров, планирование и прочие основные механизмы реализации абстракции процесса. Другая функция — обеспечение функциональности сообщений, то есть проверка правильности адресатов, обнаружение буферов приема и передачи в основной памяти и пересылка данных от источника к приемнику. Та часть кода, которая работает с низкоуровневой обработкой прерываний, написана на языке ассемблера. Остальная часть кода этого и других слоев реализована на языке C.

Второй слой составляют процессы ввода/вывода, по одному на отдельный тип устройств. Чтобы отличать такие процессы от обычных пользовательских процессов, мы будем называть их *задачами*, но различие между задачами и процессами минимально. В большинстве систем задачи ввода/вывода называются *драйверами устройств*. Мы будем использовать термины «задача» и «драйвер устройства» как эквивалентные. Каждому типу устройств, будь то диски, принтеры, терминалы, сетевые интерфейсы или таймеры, нужна своя задача. Одна задача, системная, несколько отличается от всех остальных, так как она не соответствует какому-либо устройству ввода/вывода. Эту задачу мы рассмотрим в следующей главе.

Все задачи, образующие второй слой, и весь код первого слоя связаны в одну двоичную программу, называемую *ядром*. У некоторых задач есть общие процедуры, но во всех отношениях они не зависят друг от друга, управляются планировщиком как отдельные процессы и взаимодействуют друг с другом при помощи сообщений. В процессорах Intel, начиная с 286, каждому процессу может быть назначен один из четырех уровней привилегий. При этом задачи и ядро работают с разными уровнями приоритета, хотя они и компилируются вместе. Настоящий код ядра может обращаться к любому адресу памяти и любому регистру, по сути, ядро вправе выполнить любую инструкцию с любыми данными. В отличие от ядра задачи не могут обращаться ко всем адресам памяти и всем регистрам процессора. Тем не менее им позволено обращаться к памяти менее привилегированных процессов, чтобы выполнять для них операции ввода/вывода.

Системная задача не занимается вводом-выводом в прямом смысле этого слова, ее цель — предоставление таких услуг, как копирование данных между различными областями памяти, для процессов, которые сами не умеют выполнять подобные действия. Конечно же, все эти ограничения не являются столь жесткими на машинах без нескольких уровней привилегий, например на некоторых старых машинах Intel.

Третий уровень населяют разнообразные службы, нужные пользовательским процессам. Эти процессы работают с меньшим уровнем привилегий, чем процессы ядра или задач, поэтому не в состоянии напрямую обращаться к портам ввода/вывода. Кроме того, они не могут обращаться к памяти вне выделенного для них сегмента. *Менеджер памяти* (memory manager, MM) ответственен за обработку всех системных вызовов MINIX, касающихся работы с памятью, например fork, exec и brk. Все файловые системные вызовы обрабатываются *файловой системой* (file system, FS), это, например, read, mount и chdir.

Как было замечено в начале первой главы, операционные системы выполняют две основные задачи: управляют ресурсами и реализуют при помощи системных вызовов интерфейс расширенной машины. В MINIX распределением ресурсов заведуют преимущественно первый и второй уровни, а обработкой системных вызовов занимается третий уровень. Файловая система была реализована как «сервер», поэтому она может быть размещена на удаленной машине практически в неизменном виде. То же касается и менеджера памяти, хотя удаленные серверы памяти не так полезны, как удаленные серверы файлов.

Кроме того, на третьем уровне могут существовать вспомогательные серверные процессы. На рис. 2.13 на этот уровень дополнительно помещен сетевой сервер, исходные коды которого входят в установочный пакет MINIX. Чтобы включить его, систему нужно просто перекомпилировать.

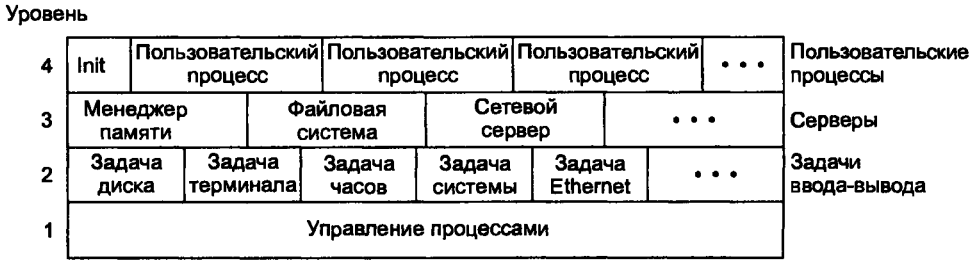


Рис. 2.13. Четыре уровня структуры MINIX

Здесь нужно отметить, что хотя различные серверы и являются независимыми процессами, они отличаются от пользовательских процессов тем, что запускаются вместе с системой, и пока система активна, они не могут быть завершены. Кроме того, хотя уровень привилегий для серверных процессов и ограничивает их теми же инструкциями, что и пользовательские программы, серверы выполняются с несколько большим приоритетом. Код запуска ядра помещает эти процессы в привилегированные ячейки таблицы процессов до того, как запустить

любую пользовательскую программу. Чтобы добавить в ядро новый сервер, ядро необходимо перекомпилировать.

Наконец, четвертый уровень содержит все пользовательские программы: оболочки, редакторы, компиляторы и творения самого пользователя. В работающей системе обычно имеется несколько процессов, запускаемых сразу после ее загрузки и работающих все время функционирования системы. Например, фоновый процесс, периодически запускающийся или работающий все время в ожидании какого-либо повеления (например, прихода пакета по сети), называется *демоном*. В действительности демон — это сервер, запускаемый независимо и функционирующий как пользовательский процесс. Но, в отличие от системных серверов, занимающих привилегированные ячейки в таблице процессов, демонам не дано нарушать системных ограничений, как это делают сервер памяти или файловый сервер.

2.5.2. Управление процессами в MINIX

В MINIX процессы соответствуют описанной несколько ранее обобщенной модели процесса. Процессы могут запускать процессы-потомки, те, в свою очередь, могут делать то же самое, образуя, таким образом, дерево процессов. По сути, все пользовательские процессы в системе — листы одного дерева, в корне которого находится init (см. рис. 2.13).

Как возникает такая ситуация? При включении компьютера первый сектор или первая дорожка загрузочного диска автоматически считываются в память, и хранимый там код исполняется. Детали этого процесса во многом зависят от того, производится загрузка с гибкого или жесткого диска. На дискете первый сектор содержит *программу начальной загрузки*. Это очень маленькая программа, так как она должна уместиться в один сектор диска. Затем в MINIX эта программа загружает и запускает более сложную программу, boot, а она, в свою очередь, загружает операционную систему.

При загрузке с жесткого диска требуется дополнительный промежуточный этап. Жесткие диски разбиваются на *разделы*, и первый сектор жесткого диска содержит небольшую программу и *таблицу разделов*, вместе они называются *главной загрузочной записью* (Master Boot Record, MBR). Программная часть считывает таблицу разделов и выбирает *активный* раздел. В первом секторе активного раздела расположена программа начальной загрузки, которая загружается, после чего действует так же, как при загрузке с дискеты, запуская программу boot.

В любом случае после запуска boot ищет на диске различные компоненты системы и размещает их по нужным адресам. Это касается ядра, менеджера памяти, файловой системы и программы init, корневого пользовательского процесса. Запуск системы — не простая операция. Ядру необходимо выполнять действия с диском и файловой системой до того, как эти части придут в готовность. В последующих разделах мы вернемся к вопросу запуска MINIX, а сейчас будет достаточно сказать, что, как только процесс загрузки завершается, ядро начинает работу.

В процессе инициализации ядро запускает задачи, затем менеджер памяти, файловую систему и прочие сервисы третьего уровня. Когда все они инициали-

зированы, они переходят в состояние блокировки, ожидая дальнейших событий. В этот момент, когда все задачи и серверы заблокированы, получает управление первый пользовательский процесс — init. Он уже находится в памяти, хотя, конечно, он мог бы быть загружен с диска или откуда-нибудь еще как отдельная программа, так как все остальное к моменту его запуска уже работает. Тем не менее, поскольку init запускается всего один раз, проще включить его в загрузочный образ системы вместе с ядром, задачами и серверами.

Став активным, init считывает файл /etc/ttytab, в котором перечислены все возможные устройства-терминалы. Если какое-либо устройство может быть использовано как терминал для входа в систему (в стандартной поставке это только консоль), то в /etc/ttytab у него присутствует запись в поле getty. Для каждого такого терминала init запускает дочерний процесс. Обычно каждый из этих дочерних процессов запускает программу /usr/bin/getty, которая печатает приглашение и ждет, когда пользователь введет свое имя. После этого вызывается программа /usr/bin/login, которой в качестве аргумента передается введенное имя. Если же для определенного терминала требуются какие-то специальные действия (например, это телефонное подключение), то в /etc/ttytab можно указать команду (например, /usr/bin/stty), которая будет выполнена перед запуском getty.

Если вход в систему осуществлен успешно, /bin/login запускает оболочку пользователя, имя которой указывается в файле /etc/passwd, обычно это /bin/sh или /usr/bin/ash. Оболочка воспринимает и интерпретирует вводимые пользователем команды, запуская для их исполнения новые процессы. Таким образом, оболочки являются прямыми потомками init, пользовательские процессы являются его «внуками», а все процессы вместе образуют единое дерево.

Для управления процессами служат два основных системных вызова: fork и exec. Вызов fork реализует единственный способ создать новый процесс. Вызов exec позволяет процессу запустить указанную программу. При запуске программы ей выделяется объем памяти, указанный в ее заголовке. Это количество памяти удерживается за программой все время работы, хотя распределение памяти между сегментом данных, сегментом стека и неиспользованным пространством может меняться динамически.

Вся информация о процессах хранится в таблице процессов, поля которой поделены между ядром, менеджером памяти и файловой системой. Когда появляется новый процесс (при помощи fork) или же когда процесс завершается (по системному вызову exit или по сигналу), то прежде всего свои поля в таблице процессов обновляет менеджер памяти. Затем он посылает ядру и файловой системе указания поступить таким же образом.

2.5.3. Взаимодействие между процессами в MINIX

Для отправки и приема сообщений предусмотрено три примитива. Им соответствуют следующие процедуры языка C:

- send(dest, &message); — отправляет сообщение процессу dest,

- receive(source, &message); — принимает сообщение от процесса source (или от ANY),
- send_rec(src_dst, &message); — отправляет сообщение и ждет ответа от процесса.

У всех трех вызовов во втором аргументе передается локальный адрес данных сообщения. Механизм передачи сообщений в ядре копирует эти данные из буфера отправителя в буфер приемника. При использовании send_rec ответ записывается поверх исходного сообщения. В принципе, механизм передачи сообщений в ядре можно было бы переделать так, чтобы обмениваться ими через сеть и тем самым реализовать распределенную систему. На практике это осложняется тем, что сообщения обычно содержат указатели на большие структуры данных и в распределенной системе необходимо реализовать механизм копирования этих данных через сеть.

Любой процесс или задача могут обмениваться сообщениями с процессами и задачами своего уровня, а также с процессами и задачами, находящимися строго одним уровнем ниже. Пользовательские процессы, таким образом, не в состоянии напрямую взаимодействовать с задачами ввода/вывода. Этот запрет накладывает сама система.

Когда процесс отправляет сообщение (что также касается и задач, как специального случая процесса), он блокируется до тех пор, пока адресат не сделает вызов receive. Другими словами, в MINIX во избежание проблем с буферизацией отправленных, но еще не принятых сообщений применяется метод рандеву. Эта схема обеспечивает меньшую гибкость, чем модель с буферизацией, но она вполне адекватна и гораздо проще в реализации, так как не требуется никакой работы с буферами.

2.5.4. Планирование процессов в MINIX

Мультипрограммную операционную систему движет система прерываний. Сделав запрос на ввод данных, процесс блокируется, позволяя выполняться другим процессам. Когда запрошенные данные становятся доступны, процесс прерывается диском, клавиатурой или другим оборудованием, для этого устройство ввода/вывода посылает процессу сообщение, пробуждая его. Кроме того, прерывания генерируются и таймером, это гарантирует, что не запрашивающий ввода процесс не будет работать слишком долго.

Каждый раз, как происходит прерывание процесса, будь его причина таймер или периферийное устройство, у системы появляется возможность принять решение, какому процессу больше требуется процессор. Конечно, это обязательно нужно делать и при завершении процесса, но в системах, подобных MINIX, переключение должно случаться чаще, чем завершения процессов. Планировщик MINIX использует многоуровневую систему очередей, соответствующую 2-му, 3-му и 4-му уровням, изображенным на рис. 2.13. При этом задачи и серверы выполняются до тех пор, пока они не попадут в состояние блокировки, а для планирования пользовательских процессов применяется циклическая схема.

Наибольший приоритет у задач, затем следуют менеджер памяти и файловая система, а на последнем месте находятся пользовательские процессы.

Прежде чем выбрать процесс для запуска, планировщик проверяет, нет ли готовых к работе задач. Если такие задачи есть, управление получает та из них, которая находится в начале очереди. В противном случае управление по возможности передается менеджеру памяти или файловой системе. Если это невозможно, запускается пользовательский процесс. Если же готовых к запуску пользовательских процессов также нет, выбирается процесс IDLE, который представляет собой цикл, выполняющийся до тех пор, пока не произойдет прерывание.

По каждому сигналу таймера делается проверка времени непрерывной работы текущего пользовательского процесса. Если это время превышает 100 мс, планировщик смотрит, не нужен ли процессор другой программе. Если такой процесс находится, он получает управление, а текущий помещается в конец очереди. Задачи, менеджер памяти и файловая система никогда не прерываются сигналами таймера, независимо от того, как долго они работают.

2.6. Реализация процессов в MINIX

Теперь мы близко подошли к рассмотрению реального кода. Но предварительно следует сказать несколько слов о терминах, которые вы встретите в этой главе. Термины «процедура» и «функция» означают то же самое, что и «подпрограмма». Имена переменных, процедур и файлов, а также системные вызовы записываются моноширинным шрифтом, например read.

Книга не попадает в печать в первый же день после последней точки, она и программное обеспечение постоянно развиваются. Поэтому между напечатанными листингами, примерами кода и версией на компакт-диске могут быть небольшие различия. Но обычно различия незначительны, не более одной или двух строк.

2.6.1. Структура исходного кода MINIX

Исходные коды системы логически разбиты на два каталога. В стандартной системе это /usr/include и /usr/src/. Реальное их размещение может быть другим, но структура в любой системе всегда одинакова. Ссылаясь на эти каталоги, мы будем называть их include/ и src/ соответственно.

В каталоге include/ расположено несколько стандартных заголовочных файлов POSIX. Кроме того, в нем есть три вложенных каталога:

1. sys/ — здесь содержатся дополнительные заголовочные файлы POSIX.
2. minix/ — заголовочные файлы операционной системы.
3. ibm/ — заголовочные файлы с определениями, специфичными для IBM PC.

Для поддержки расширения MINIX и программ, работающих в MINIX-среде, в каталог include/ включены и другие подкаталоги, имеющиеся на компакт-диске или в Интернете. Например, для поддержки сетевых расширений предусмотрен

каталог include/net и вложенный в него include/net/gen/. Но в данной книге обсуждаются только коды, необходимые для работы системы.

Каталог src/ содержит три важнейших вложенных каталога, где размещаются исходные коды операционной системы:

1. kernel/ — первый и второй уровни (процессы, сообщения и драйверы).
2. mm/ — коды менеджера памяти.
3. fs/ — коды файловой системы.

Кроме того, есть еще три каталога, содержимое которых в книге не рассматривается, но которые важны для получения работающей среды, это:

1. src/lib/ — исходные коды библиотечных процедур (например, open, read).
2. src/tools/ — коды программы init, используемой при загрузке системы.
3. src/boot/ — коды загрузки и установки MINIX.

В стандартной комплектации MINIX вы найдете еще несколько каталогов. Суть операционной системы в том, чтобы поддерживать некоторый набор команд (то есть программ), таких как cat, cp, date, ls, pwd. Исходные тексты этих программ расположены в каталоге src/commands/. Помимо того, MINIX является учебной системой, которую можно изменять, поэтому в каталоге src/test/ имеется набор программ для тестирования. Наконец, в каталоге src/inet/ находятся коды программ для компиляции MINIX с поддержкой сети.

Для удобства мы не будем указывать полное имя файла, кроме тех случаев, когда его нельзя понять из контекста. Но нужно заметить, что в разных каталогах присутствуют файлы с одинаковыми именами. Например, есть несколько файлов с названием const.h, в которых задаются константы для соответствующей части системы. Это не должно вызвать затруднения, так как файлы в одном каталоге будут рассматриваться совокупно.

Код первого и второго уровней системы находится в каталоге src/kernel/. В этой главе мы рассмотрим те файлы, которые относятся к управлению процессами, нижнему уровню MINIX на рис. 2.13. Этот уровень включает в себя функции для инициализации системы, обработки прерываний, передачи сообщений и планирования процессов. В третьей главе мы доберемся до остальных файлов этого каталога, ответственных за выполнение различных задач со второго уровня. В главе 4 мы рассмотрим менеджер памяти, коды которого расположены в src/mm/, и в 5-й главе изучим файловую систему (src/fs/).

При компиляции системы все файлы с исходными кодами преобразуются в объектные файлы. Затем объектные файлы из каталога src/kernel/ связываются в единый исполняемый файл kernel. Объектные файлы из src/mm/ также связываются в отдельный исполняемый файл, mm. Подобным образом получается и fs. Расширение системы достигается включением в нее дополнительных серверов. Например, чтобы добавить поддержку сетей, нужно изменить файл include/minix/config.h, включив компиляцию файлов из каталога src/inet/, которые образуют программу inet. Еще одна программа, init, строится в каталоге src/tools. Программа installboot, коды которой лежат в src/boot/, именует все эти программы, дополняет так, чтобы размер каждой был кратен длине сектора диска (чтобы было проще

загружать их по отдельности) и объединяет в единый файл. Последний и представляет собой двоичный код операционной системы. Его можно скопировать либо в корневой каталог, либо в каталог /minix/ на жестком диске или дискете. Впоследствии загрузчик будет загружать операционную систему из этого файла.

Распределение памяти после того, как слитые в один файл программы будут разделены и загружены, показано на рис. 2.14. Но, конечно же, подробности зависят от конфигурации системы. Пример, приведенный на рисунке, соответствует системе, оптимизированной под компьютер с несколькими мегабайтами оперативной памяти. Это позволяет выделять множество буферов для файловой системы, но приводит к тому, что все они не помещаются в нижнюю память (первые 640 Кбайт). Если же значительно уменьшить число буферов, то всю систему удается разместить в нижней памяти, причем остается еще место для нескольких пользовательских процессов.

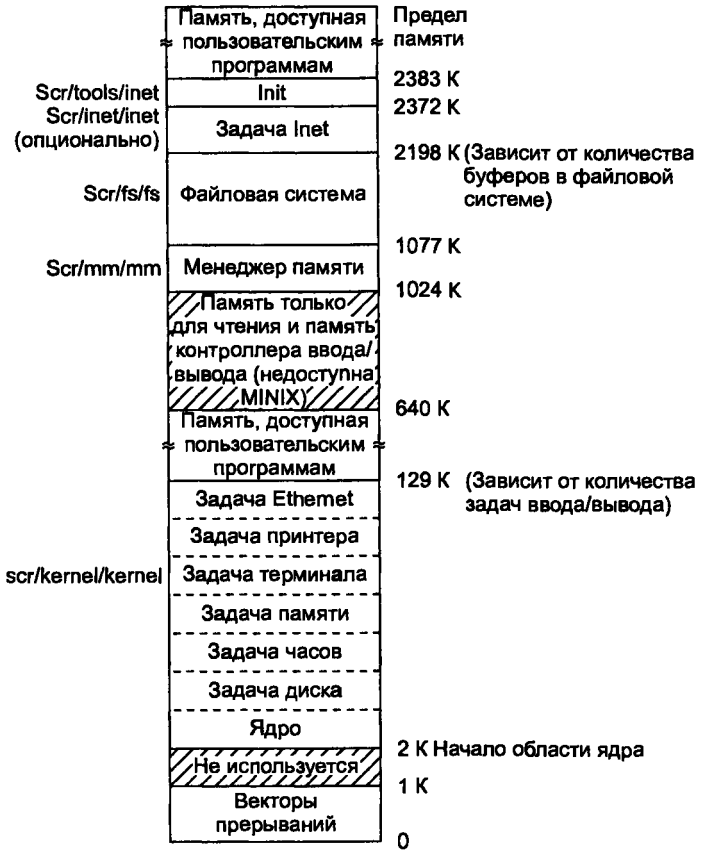


Рис. 2.14. Распределение памяти после загрузки MINIX. Четыре (или пять, если включена поддержка сети) отдельных блока полностью разделены

Важно понимать, что MINIX состоит из трех или более совершенно независимых программ, взаимодействующих при помощи сообщений. Поэтому процедура с именем panic в src/fs/ не конфликтует с процедурой painc в src/mm/. Единственные процедуры, видимые и доступные всем частям системы, расположены в каталоге lib/. Благодаря такой модульной структуре очень просто изменить одну ее часть, скажем, файловую систему, не затронув других частей. Модульность позволяет даже целиком перенести файловую систему на удаленную машину, сделав ее файловым сервером, который взаимодействует с другими пользовательскими машинами, обмениваясь сообщениями через сеть.

Другим примером модульности MINIX может служить то, что наличие или отсутствие поддержки сети в скомпилированной системе не оказывает никакого влияния на менеджер памяти, затрагивая только ядро системы (так как задача Ethernet входит в ядро, как и другие задачи устройств ввода/вывода). Сетевой сервер, если он нужен в системе, встраивается в MINIX точно так же, как сервер памяти и файловый сервер и работает с тем же уровнем приоритета. В его задачи входит передача больших объемов данных за небольшое время, поэтому ему требуется больший приоритет, чем может понадобиться пользовательским процессам. Но это интересует только задачу Ethernet, а многие другие сетевые функции могут быть реализованы на пользовательском уровне. Сетевые функции не являются традиционными системными функциями и детальное их рассмотрение не входит в задачи этой книги. В последующих главах и разделах мы будем изучать систему MINIX, скомпилированную без поддержки сети.

2.6.2. Общие заголовочные файлы

В каталоге include/ содержится набор файлов, определяющих общие константы, типы данных и макроопределения. Большинство этих определений обусловлены стандартом POSIX, который указывает, какое определение в каком файле каталога include/ или его подкаталоге include/sys/ находится. Файлы, которые присутствуют в этих каталогах, называются *заголовочными* или *включаемыми* файлами и имеют расширение .h. Подключаются они директивой #include языка C. Благодаря включаемым файлам упрощается поддержка большой системы.

Для компиляции пользовательских программ требуются по большей части заголовочные файлы из include/, а файлы из include/sys/ традиционно нужны для компиляции системных утилит. Но это не слишком важное ограничение, и в типичной программе, будь то пользовательская программа или часть системы, используются файлы из обоих каталогов. Здесь мы обсудим файлы, необходимые для компиляции MINIX, сначала те из них, которые лежат в include/, а затем из состава include/sys/. В следующем разделе мы рассмотрим файлы в каталогах include/minix/ и include/ibm/, которые соответственно содержат код, специфичный для системы MINIX и ее реализации на IBM-совместимых компьютерах.

Файлы в каталоге include/ в действительности предназначены для решения общих задач, поэтому в большинство модулей с исходным кодом системы они не включаются. Вместо этого они присоединяются в другие заголовочные файлы, например в главный заголовочный файл src/kernel/kernel.h, файл src/mm/mm.h

и src/fs/fs.h, подключаемые при каждой сборке системы. Главные заголовочные файлы каждой из трех составных частей системы служат каждый для своих целей, но начало у всех них одинаковое, оно приведено в листинге 2.12. Главные заголовочные файлы еще будут обсуждаться в этой книге, цель данного обзора — только подчеркнуть, что заголовочные файлы из различных каталогов используются совместно. В этом и следующем разделах мы упомянем каждый из перечисленных в листинге 2.12 файлов.

Листинг 2.12. Начальная часть всех главных заголовочных файлов, подключающая все прочие заголовочные файлы, нужные в коде

```
#include <minix/config.h>    /* Этот файл ДОЛЖЕН включаться первым */
#include <ansi.h>            /* Этот файл ДОЛЖЕН включаться вторым */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <limits.h>
#include <errno.h>
#include <minix.syslib.h>
```

Начнем с первого из файлов в каталоге include/ — ansi.h (строка 0000). Он необходим при компиляции любой из частей системы. Его назначение — удостовериться, что компилятор соответствует требованиям стандарта языка С, который определяется Международной организацией по стандартизации (International Organization for Standards). Этот стандарт также иногда называется ANSI C, поскольку изначально, до международного признания, он разрабатывался Американским национальным институтом стандартов (American National Standard Institute). Соответствующий стандарту компилятор должен определять несколько макросов, пригодных к использованию в компилируемых программах. Например, макрос __STDC__ у правильного компилятора должен иметь значение 1, как если бы последнему была подана строка

```
#define __STDC__ 1
```

Сейчас поставляемый с MINIX компилятор удовлетворяет стандарту, но более старые версии были разработаны до его принятия, поэтому MINIX все еще можно скомпилировать классическим компилятором С (Керниган и Ричи). MINIX создавалась как легко переносимая система, и возможность привлекать старые компиляторы — важная составная часть задуманного. Выражение

```
#define _ANSI
```

на строках 0023 и 0025 обрабатывается в том случае, если применяется стандартный компилятор. В ansi.h определяются несколько различных макросов, причем то, как это делается, зависит от того, определен ли макрос _ANSI.

Главный макрос в файле ansi.h — это макрос _PROTOTYPE. Он позволяет записывать прототип функции в таком виде:

```
_PROTOTYPE(тип-результата, имя-функции, (тип-аргумента аргумент, ...))
```

и, если компилятор соответствует стандарту, препроцессор приводит эту запись к следующему виду:

```
тип-результата имя-функции (тип-аргумента аргумент, ...)
```

Если же компилятор более старый, макрос транслируется так:

```
тип-результата имя-функции ()
```

Прежде чем переходить от ansi.h к другим файлам, обратим внимание еще на один момент. Все содержимое файла обрамлено строками:

```
#ifndef _ANSI_H
```

и

```
#endif
```

Кроме того, сразу после строки с #ifndef _ANSI_H определяется сам макрос _ANSI_H. Назначение этой конструкции в том, чтобы убедиться, что заголовочный файл будет включен только один раз. При повторном включении все его содержимое будет проигнорировано. Подобная техника используется во всех файлах из каталога include/.

Второй файл из include/, косвенно включаемый в каждый из файлов с кодами системы, — это limits.h (строка 0100). В нем объявлены основные ограничения, относящиеся к типам языка и системе. Например, число битов в целом числе или максимальная длина имени файла. Все главные заголовочные файлы включают также файл errno.h (строка 0200). Здесь содержатся коды ошибок, возвращаемые пользователю в глобальной переменной errno после системных вызовов. Эта переменная также индицирует некоторые внутренние ошибки, например попытку переслать сообщение несуществующему процессу. Причем внутри системы коды ошибок отрицательны, чтобы можно было понять, что это ошибки, а возвращаемое пользователю значение должно быть положительным. Этот эффект достигается с помощью следующего приема: каждый код ошибки определяется специальной строкой вида (строка 0236):

```
#define EPERM (_SIGN 1)
```

При компиляции системных файлов в главных заголовочных файлах определяется макрос _SYSTEM, в результате чего _SIGN транслируется в «-», а при компиляции пользовательских программ _SYSTEM никогда не определяется, и _SIGN просто игнорируется.

Следующая рассматриваемая группа файлов не включается в главные заголовочные файлы, но тем не менее эти файлы повсеместно используются в коде системы. Главный из них — unistd.h (строка 0400). В нем перечислены константы, требуемые стандартом POSIX. Кроме того, в нем описаны прототипы многих функций, в том числе всех функций для доступа к системным вызовам MINIX. Второй файл — string.h, он содержит прототипы большого количества функций для манипуляции со строками. Файл signal.h (строка 0700) задает стандартные имена сигналов. Кроме того, в нем определены прототипы некоторых функций для работы с сигналами. Как мы позднее увидим, работа с сигналами присуща всем составляющим MINIX.

В fcntl.h указываются различные параметры, имеющие значение при управлении файлами. Например, благодаря задаваемым здесь константам для открытия файла в режиме чтения можно указывать макрос O_RDONLY вместо того, чтобы

напрямую передавать значение 0. Этот файл нужен в основном файловой системе, но он же используется в нескольких местах внутри ядра и менеджера памяти.

Оставшиеся в каталоге include/ файлы требуются не так широко, как ранее упомянутые. Файл stdlib.h содержит описания типов, макросов и прототипов функций, нужные практически всем программам, за исключением самых простых. При создании пользовательских программ это один из наиболее часто используемых файлов, хотя в кодах MINIX он применяется только несколько раз в ядре.

Как мы увидим при рассмотрении уровня задач MINIX в главе 3, консольный и терминальный интерфейсы операционной системы сложны, по причине того что большое количество различного оборудования должно взаимодействовать с пользователем стандартным образом. Для управления устройствами ввода/вывода терминального типа используются константы, макросы и функции, прототипы которых приведены в файле termios.h. Самая главная структура здесь — структура termios. В нее входят различные флаги, управляющие режимами работы, переменные для задания скоростей ввода и вывода данных, а также массив специальных символов, таких как INTR и KILL. Эта структура, как и многие макросы и функции файла termios.h, регламентирована стандартом POSIX.

Тем не менее, каким бы всеобъемлющим ни был стандарт POSIX, он не может включить в себя все, что может понадобиться. Поэтому вторая часть файла, начиная со строки 1241, относится к расширениям POSIX. Некоторые из них вполне очевидны, например определение стандартных скоростей передачи данных (57 600 бод и выше). Подобные расширения не запрещены POSIX, а сам стандарт не может объять необъятное. Но при разработке в MINIX программ, которые рассчитаны и на перенос в другое окружение, следует избегать специфичных для MINIX определений. Сделать это несложно. Если в файле имеются специфичные для MINIX расширения, то их использование контролируется макрокомандой

```
#ifdef _MINIX
```

Если макрос _MINIX не определен, расширения игнорируются.

Последний файл в каталоге include/, который мы рассмотрим, — a.out.h. Он определяет формат, в котором исполняемые файлы хранятся на диске, включая заголовок файла, нужный для его запуска, и таблицу символов, создаваемую компилятором. Этот заголовочный файл интересен только файловой системе.

Теперь переместимся в подкаталог include/sys/. Как видно из листинга 2.12, все главные заголовочные файлы основных составных частей MINIX включают в себя файл sys/types.h, сразу после ansi.h. В types.h определяются различные типы данных, встречающиеся в MINIX. Благодаря ему можно избежать различных ошибок, связанных с неправильным использованием типов. Размеры некоторых типов данных (в битах) для 16- и 32-разрядных систем указаны в табл. 2.2. Кроме того, обратите внимание, что имена всех типов данных заканчиваются символами «_t». Это — больше чем договоренность. Это требование стандарта POSIX. Согласно последнему, окончание «_t» является *зарезервированным суффиксом*

и не должно использоваться ни в каких других идентификаторах, не являющихся именами типов.

Таблица 2.2. Размер некоторых типов на 16- и 32-разрядных системах (размер указан в битах)

Тип	16-разрядная MINIX	32-разрядная MINIX
gid_t	8	8
dev_t	16	16
pid_t	16	32
ino_t	16	32

Множество различных макросов для действий по управлению устройствами имеется в файле sys/ioctl.h, хотя он используется не настолько часто, чтобы включать его в главный заголовочный файл для каждой секции. Кроме того, здесь же описывается прототип системного вызова ioctl. Правда, сейчас, когда функции POSIX, прототипы которых находятся в файле include/termios.h, во многом заменили старую библиотечную функцию ioctl, этот вызов не так часто делается программистами напрямую. Но тем не менее он все еще необходим. Фактически функции POSIX для управления терминалами транслируются библиотекой в соответствующие вызовы ioctl. Вдобавок, все возрастает количество прочих интерфейсных устройств, подлежащих контролю. Так, в конце этого файла определено несколько кодов операций, начинающихся с DSPIO, предназначенных для управления числовыми сигнальными процессорами. Конечно, в этой книге мы описываем систему только с основными периферийными устройствами. Но ничто не мешает добавить и многие другие типы устройств: сетевые адаптеры, приводы компакт-дисков, звуковые карты. Управляющие коды для всех этих устройств представлены в файле sys/ioctl.h в виде макросов.

Широко используются в системе еще несколько файлов из рассматриваемого каталога. В sys/sigcontext.h задаются различные структуры, служащие для сохранения и последующего восстановления состояния системы перед вызовом обработчика прерывания. Этот файл необходим как для ядра, так и для менеджера памяти. В MINIX предусмотрена поддержка отладки программ и просмотра памяти при помощи системного вызова ptrace, все возможные действия с которым описаны в файле sys/ptrace.h. Файл sys/stat.h определяет структуры, которые упоминались в листинге 1.1, для системных вызовов stat и fstat. Также в этом файле находятся прототипы функций для работы со свойствами файлов. К нему по мере необходимости обращаются некоторые части файловой системы и менеджера памяти.

Последние два файла, которые мы рассмотрим в текущем разделе, востребованы не так широко, как описанные выше. В файле sys/dir.h описан формат каталога в MINIX. Значение этого файла, помимо прочего, в том, что он задает максимальную длину имени файла. Напрямую он используется только один раз, но опосредовано — достаточно часто, через подключение в другой заголовочный файл. Наконец, в файле sys/wait.h представлены макросы, нужные для работы с системными вызовами wait и waitpid, реализуемыми менеджером памяти.

2.6.3. Заголовочные файлы MINIX

Файлы, специфичные только для MINIX, расположены в каталогах include/minix/ и include/ibm/. Первый из каталогов содержит файлы, общие для реализаций MINIX на всех платформах (хотя в некоторых из файлов есть альтернативные определения, зависящие от платформы), а второй — специфичные для платформы IBM.

Мы начнем изучение с каталога minix/. Ранее был упомянут файл config.h, включаемый во все главные заголовочные файлы основных компонентов системы, и фактически являющийся первым файлом, обрабатываемым компилятором. Во многих случаях, когда в системе меняется оборудование или же нужно изменить способ работы MINIX, все, что нужно сделать, — отредактировать этот файл и пересобрать систему. Начальная часть файла представляет собой список настраиваемых пользователем параметров, первый из которых — MACHINE, может принимать такие значения, как IBM_PC, SUN_4, MACINTOSH или другие, в зависимости от того, для какой машины компилируется MINIX. Это не оказывает влияния на большую часть кода, но у такой программы, как операционная система, всегда есть платформно-зависимый код. В тех немногих частях нашей книги, где мы будем обсуждать подобный код, мы будем ориентироваться на IBM-совместимые компьютеры в 32-разрядном режиме (то есть 80386, 80486, Pentium, Pentium Pro и т. д.); такие процессоры будут называться 32-разрядными Intel-совместимыми. Кроме того, MINIX можно скомпилировать и для более старых процессоров Intel, если использовать 16-разрядные слова. Для таких машин некоторые части системы должны кодироваться иначе. На PC компилятор самостоятельно определяет тип аппаратного обеспечения, для которого компилируется система.

Стандартным инструментом для MINIX на PC является компилятор из пакета разработчика ACK (Amsterdam Compiler Kit). Он идентифицируется предустановленными макросами __STDC__ и __ACK__. Также он самостоятельно определяет макрос, хранящий величину машинного слова целевой машины (в байтах), его имя _EM_WSIZE. В исходных кодах системы создается новый макрос, _WORD_SIZE, которому присваивается значение _EM_WSIZE. Затем этот макрос можно использовать в дальнейшем коде, чтобы определить тип системы. Например, директива препроцессора

```
#if (MACHINE == IBM_PS && _WORD_SIZE == 4)
```

позволяет выделить 32-разрядные PC, чтобы задать для них, например, специфический размер буфера.

Другое назначение config.h — дополнительная настройка конкретной системы. Например, имеется секция настроек, в которой можно указать различные типы драйверов устройств, включаемых в ядро при компиляции. Эта секция начинается строками:

```
#define ENABLE_NETWORKING 0
#define ENABLE_AT_WINI 1
#define ENABLE_BIOS_WINI 0
```

Изменив значение 0 в первой строке на 1, можно включить поддержку сети в ядре MINIX. Если же определить ENABLE_AT_WINI как 0, а ENABLE_BIOS_WINI как 1, можно отключить код драйвера IDE-дисков и тем самым обращаться к жесткому диску через BIOS.

Следующий файл, const.h, демонстрирует другое типичное применение заголовочных файлов. В нем перечислены различные константы, которые, скорее всего, не должны меняться при компиляции ядра, но которые используются в различных частях системы. Поместив подобные определения в отдельный файл, вы оградите себя от ошибок, вызванных несоответствием значений в различных местах программы. В дереве исходных кодов MINIX имеется еще несколько файлов с именем const.h, но у них более ограниченное применение. Например, константы, востребуемые только кодом ядра, помещены в src/kernel/const.h. Константы для файловой системы хранятся в src/fs/const.h. У менеджера памяти тоже собственный файл для своих констант — src/mm/const.h. В основной файл, include/minix/const.h, помещены только те определения, которые используются в различных частях системы.

Имеет смысл особо упомянуть некоторые макроопределения из const.h. Макрос EXTERN транслируется в ключевое слово extern. При помощи этого макроса определяются глобальные переменные, продекларированные в заголовочном файле и включаемые в два или более файлов с кодом. Например:

```
EXTERN int who;
```

Если же просто объявить переменную как

```
int who;
```

и включить это объявление в два или более файла, то при сборке программы редактор связей сообщит о том, что одна и та же переменная объявлена несколько раз. Более того, справочное руководство C (Керниган и Ричи, 1988) строго воспрещает подобную конструкцию.

Чтобы избежать подобных проблем, переменные следует декларировать так:

```
extern int who;
```

причем это нужно делать во всех местах, кроме одного. Здесь и помогает макроопределение EXTERN, разворачивающееся в extern, если включен файл const.h, или в пустую строку, если явно переопределить EXTERN. В каждой из частей MINIX это делается путем помещения глобальных переменных в отдельный файл, называемый glo.h, косвенно включаемый в каждый процесс компиляции. Например, для ядра таким будет файл src/kernel/glo.h. В каждом из glo.h имеются подобные строки:

```
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif
```

В файле table.c для каждой из составных частей системы перед секцией #include имеется такая директива:

```
#define _TABLE
```

Таким образом, когда заголовочный файл включается при компиляции table.c, ключевое слово extern перед объявлениями переменных не ставится (так как в файле table.c макрос EXTERN определен как пустая строка), поэтому все глобальные переменные размещаются в одном месте, в объектном файле table.o.

Если вы новичок в программировании на C и не совсем понимаете, к чему приводят описанные выше действия, не пугайтесь — эти детали в действительности не так важны. Суть в том, что у некоторых компоновщиков многократное включение одного файла может вызвать проблему, так как приводит к многократному объявлению одних и тех же переменных. EXTERN используется для того, чтобы сделать систему более переносимой между различными платформами с потенциально проблематичными компоновщиками.

Макрос PRIVATE синонимичен ключевому слову static. Этот макрос всегда применяется для объявления переменных и функций, на которые не будут ссылаться другие компоненты системы, чтобы имена этих объектов не были видимы за пределами того файла, где они объявлены. Как правило, переменные и процедуры необходимо по возможности размещать в локальной области видимости. Макрос PUBLIC определен как пустая строка. Так, например, декларация:

```
PUBLIC void free_zone(Dev_t dev, zone_t numb)
```

преобразуется препроцессором в следующий код:

```
void free_zone(Dev_t dev, zone_t numb)
```

который, в соответствии с синтаксическими правилами языка C, означает, что функция с именем free_zone экспортируется и может быть использована в других файлах. Использовать макросы PRIVATE и PUBLIC необязательно, это только попытка исправить проблемы, вносимые соглашениями C (где по умолчанию имена размещаются в глобальной области видимости, а должно быть наоборот).

В оставшейся части файла const.h определяются константы, повсеместно используемые в системе. Так, например, везде и всюду в коде фигурирует величина базового блока памяти, зависящая от архитектуры системы. Кроме того, в этом файле определяются удобные макросы MAX и MIN, позволяющие для вычисления большего из двух значений применять следующую запись:

```
z = MAX(x, y);
```

Еще один файл, косвенно включаемый при каждой компиляции в главных заголовочных файлах, — это type.h. В нем содержится ряд описаний ключевых типов и связанных с ними числовых констант. Самый важный тип здесь — message (сообщение). Его можно было бы задать как массив определенного количества байтов, но ради хорошего стиля программирования он описан как структура, содержащая объединение различных возможных типов сообщений. Всего имеется шесть форматов сообщений, с именами от mess_1 до mess_6. В самой структуре message есть поле m_source с информацией об отправителе сообщения, поле m_type, говорящее о том, каков формат сообщения (например, для задачи таймера это может быть GET_TIME), и поля с данными сообщения. Структуры шести типов сообщений показаны на рис. 2.15. На этом рисунке первая и вторая структура кажутся одинаковыми, равно как и вторая и четвертая, что действительно

так для 32-разрядных реализаций MINIX на платформе Intel. Но для других машин, где типы int, long и указатели могут иметь другой размер, это не факт. Поэтому для упрощения компиляции и определено шесть различных структур.

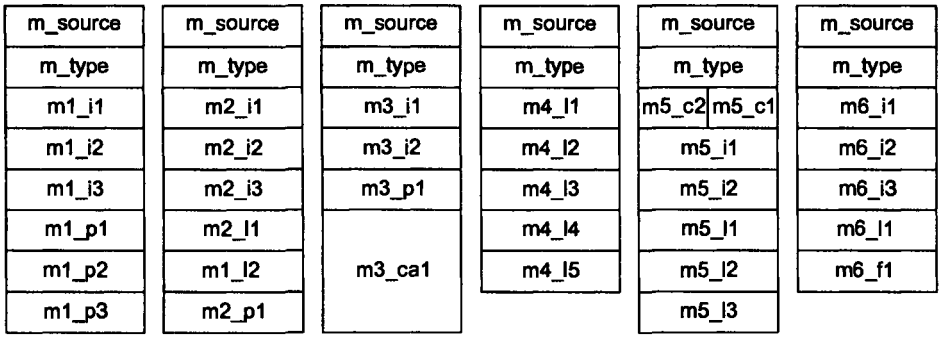


Рис. 2.15. Структуры шести типов сообщений, используемых в MINIX. Размеры элементов могут меняться, в зависимости от архитектуры целевой машины. Диаграмма соответствует компьютеру с 32-разрядными указателями, например Pentium (Pro)

Когда необходимо передать сообщение, содержащее, к примеру, три целочисленных значения и три указателя, задействуется первая структура. Подобным же образом используются и другие форматы. Как же можно присвоить нужное значение первому целочисленному полю в первой структуре? Предположим, что сообщение имеет имя x. Тогда объединение (union), содержащее параметры сообщения, будет иметь имя x.m_u. Чтобы обратиться к первой структуре этого объединения, подставляется имя x.m_u.m_m1. Наконец, чтобы обратиться к первому целочисленному полю в этой структуре, следует применять запись: x.m_u.m_m1.m1i1. Она довольно многословна, поэтому после описания самого типа message описываются несколько макроопределений, слегка укорачивающих запись. Так, x.m_u.m_m1.m1i1 становится заменяемой x.m1_i1. Укороченные имена имеют следующий формат: они начинаются с буквы «m», затем следует номер структуры, знак подчеркивания, один или два символа, обозначающие тип значения (целое число, указатель, длинное целое, символ, массив символов, функция), после чего записывается число, позволяющее локализовать нужное поле в структуре.

Отступая в сторону, при обсуждении форматов сообщений имеет смысл обратить внимание на то, что операционная система и компилятор зачастую «понимают» такие вещи, как размещение структур, что может упростить жизнь программиста. В MINIX поля с типом int зачастую используются для хранения *беззнаковых* целых значений. В некоторых случаях это опасно ошибками переполнения, но система написана в расчете на то, что компилятор без потерь может присваивать значения типа unsigned переменным типа int и наоборот без изменения данных или возникновения переполнения. При более строгом подходе нужно было бы заменить каждую такую целочисленную переменную объединением, состоящим из поля int и поля unsigned. То же самое относится и к полям типа

long, которые иногда используются для передачи данных типа unsigned long. Кто-то может сказать, что мы поступаем неправильно, но если вы собираетесь перенести MINIX на новую платформу, то вам почти наверняка придется сколько-то поработать над точным форматом сообщений, теперь же вы предупреждены, что поведению компилятора также нужно уделить немало внимания.

Есть в каталоге include/minix еще один файл, сквозной по всей системе благодаря тому, что включен в основные заголовочные файлы. Это файл syslib.h, содержащий прототипы библиотечных функций языка C, которые в системе используются для доступа к прочим службам ОС. Библиотеки C не обсуждаются в книге, но большая часть из них стандартна и поставляется с любым компилятором. Тем не менее сами функции, прототипы которых содержатся в файле syslib.h, специфичны для операционной системы, и при переносе MINIX на новую платформу с другим компилятором должны быть переписаны. К счастью, последнее несложно, так как назначение этих функций в том, чтобы извлечь нужные параметры, заполнить ими структуру сообщения, после чего отправить сообщение и извлечь ожидаемые значения из ответа. Как правило, в каждом таком случае требуется не больше дюжины строк кода.

В MINIX, когда процессу нужно сделать системный вызов, он посылает сообщение файловой системе (или, для краткости, FS) или менеджеру памяти (MM). Каждое такое сообщение должно содержать номер требуемого системного вызова. Соответствующие номера перечислены в файле callnr.h.

В файле com.h находятся описания, в основном актуальные при взаимодействии менеджера памяти или файловой системы с задачами ввода/вывода. Кроме того, здесь же указаны номера, соответствующие разным задачам. Задачам присвоены отрицательные номера, чтобы можно было отличать их от процессов. В этом же заголовочном файле описаны типы сообщений (коды функций), которые могут быть посланы каждой из задач. Например, задача таймера воспринимает коды SET_ALARM (установка таймера), CLOCK_TICK (возникновение прерывания таймера), GET_TIME (запрос времени) и SET_TIME (установка времени). Ответ на сообщение GET_TIME имеет код REAL_TIME.

Наконец, в каталоге include/minix содержатся несколько более специфических заголовочных файлов. Среди них — boot.h, который необходим ядру и файловой системе для детектирования устройств и доступа к данным, переданным программе boot. Еще один файл — keymap.h, задает структуры, необходимые для реализации различных национальных раскладок клавиатур. Этот файл нужен программам, генерирующим и загружающим таблицы символов. Другие файлы каталога, такие как partition.h, представляют интерес исключительно для ядра и не требуются файловой системе или менеджеру памяти. Кроме того, если в вашей реализации системы имеется поддержка дополнительных устройств ввода/вывода, то в этом каталоге будут находиться еще несколько подобных файлов, поддерживающих взаимодействие с I/O-устройствами. Структура каталога include/minix требует дополнительного пояснения. Дело в том, что в идеале пользовательские программы должны взаимодействовать с устройствами ввода/вывода исключительно через операционную систему, тогда подобные файлы должны были быть размещены в src/kernel. Тем не менее реалии работы с системой требуют, чтобы

существовали пользовательские команды, предоставляющие доступ к некоторым системным структурам, например команды для создания разделов на жестком диске. Заголовочные файлы, поддерживающие работоспособность таких утилит, вынесены в один из подкаталогов include/.

Последний каталог со специализированными заголовочными файлами, который мы затронем, — это include/ibm. В нем располагается два файла с определениями, специфическими для компьютеров семейства IBM PC. Один из них, diskparam.h, необходим задаче гибкого диска. Исходный код этой задачи, хотя она и входит в стандартную поставку MINIX, не рассматривается в нашей книге, так как он по природе подобен коду задачи жесткого диска. В другом файле, partition.h, описываются таблицы разделов жестких дисков и соответствующие константы, специфичные для IBM-совместимых компьютеров. Эти файлы помещены в отдельный каталог для упрощения переноса ОС на другие аппаратные платформы. На другой платформе файл include/ibm/partition.h будет заменен другим файлом, скорее всего, это должен быть partition.h в соответствующим образом названном каталоге. Но в то же время внутренние структуры MINIX, определяемые файлом include/minix/partition.h, должны остаться неизменными.

2.6.4. Структуры данных процессов и заголовочные файлы

Итак, теперь мы погрузимся глубже в MINIX и посмотрим, на что похож код в каталоге src/kernel. В двух предыдущих разделах дискуссия была построена вокруг выжимки из типичного основного заголовочного файла. Сейчас мы сперва познакомимся с настоящим системным заголовочным файлом, kernel.h. Он начинается с объявления трех макросов. Первый из них — _POSIX_SOURCE, макрос *проверки поддерживаемых функций*, определяемый самим стандартом POSIX. Все подобные макросы начинаются с символа подчеркивания, «_». Этот макрос включается для того, чтобы были видимы все символы, как требуемые стандартом, так и разрешенные, но не обязательные, а неофициальные расширения были бы скрыты. Следующие два макроопределения уже были упомянуты, это макрос _MINIX, переопределяющий эффект макроса _POSIX_SOURCE для расширений MINIX, и макрос _SYSTEM, наличие которого проверяется там, где при компиляции необходимо учитывать разницу между системным и пользовательским кодом (например, может меняться знак кода возврата). Затем в kernel.h включаются другие заголовочные файлы из include/ и подкаталога include/sys/ и include/minix/, в том числе те, что перечислены в листинге 2.12. Все эти файлы уже были нами рассмотрены в двух предыдущих разделах. Затем присоединяются еще четыре заголовочных файла из src/kernel/.

Здесь для новичков в программировании на C нужно упомянуть о том, что обозначают различные виды скобок в директиве #include. У каждого компилятора C имеется каталог, в котором по умолчанию ищутся включаемые файлы. Обычно это /usr/include, как принято в MINIX. Когда имя включаемого файла записано в угловых скобках (<...>), компилятор ищет его в каталоге по умолчанию или в одном из подкаталогов, если таковой указан. Если же имя файла за-

писано в кавычках, компилятор сначала просматривает текущий каталог (или подкаталог, если он указан), и если нужный файл не найден, он ищется в каталоге по умолчанию.

Файл kernel.h позволяет легко включить большое количество всех необходимых определений при помощи одной команды, имеющейся во всех файлах с кодами ядра:

```
#include "kernel h"
```

Иногда имеет значение то, в каком порядке включаются различные заголовочные файлы. А именно использование kernel.h позволяет раз и навсегда гарантировать соблюдение требуемой последовательности. Это поднимает концепцию «сделал и забыл», реализуемую заголовочными файлами, на более высокий уровень. В каталогах с кодами файловой системы и менеджера памяти имеются аналогичные главные заголовочные файлы.

Теперь давайте перейдем к четырем локальным файлам, включаемым в kernel.h. Точно так же, как в include/minix/ имелись файлы const.h и type.h, в каталоге с исходными кодами ядра, src/kernel/, присутствуют свои const.h и type.h. Те файлы, что размещены в include/minix/, помещены там потому, что они необходимы для компиляции различных частей системы, включая программы, работающие под ее управлением. Файлы в src/kernel/ содержат определения, используемые только для компиляции ядра. Аналогичные файлы есть и в каталогах с исходными кодами файловой системы и менеджера памяти, эти файлы определяют типы и константы, принципиальные только для компиляции данных частей системы.

Следующие два файла, включаемые в kernel.h, proto.h и glo.h, не имеют аналогов в общем каталоге include/, но, как будет видно позже, в файловой системе и менеджере памяти аналогичные файлы имеют место.

В const.h определен ряд машинно-зависимых значений, то есть констант, специфических для процессоров Intel. При компиляции для других процессоров эти значения, скорее всего, будут другими. Они помещены между директивами условной компиляции:

```
#if (CHIP == INTEL)
```

и

```
#endif
```

Когда система компилируется для процессора Intel, в файле config.h определяются макросы CHIP и INTEL, в результате чего машинно-зависимый код будет сгенерирован. При переносе MINIX на систему, в основе которой будет процессор Motorola 68000, в этот код будет добавлена еще одна секция, помещенная между следующими директивами:

```
#if (CHIP == M6800)
...
#endif
```

А в файле include/minix/config.h макрос CHIP будет переопределен так:

```
#define CHIP M6800
```

Описанная методика позволяет MINIX использовать константы и код, зависящие от аппаратной платформы. Но это плохо сказывается на читаемости кода, поэтому подобными конструкциями злоупотреблять не следует.

Особого внимания заслуживают еще несколько определений из const.h. Часть из них машинно-зависимые, например основные векторы прерываний и значения полей, требуемые для сброса контроллера прерываний после возбуждения каждого прерывания. У каждой задачи в ядре имеется собственный стек, но при обработке прерываний используется особый стек, размер которого определяется константой K_STACK_BYTES. Эта константа находится в машинно-зависимой секции, так как для другой архитектуры может потребоваться стек другого размера, большего или меньшего.

Прочие определения платформно-независимы, но встречаются во многих местах кода ядра. Так, у планировщика процессов MINIX имеются три очереди NQ с разными приоритетами, которые соответственно называются TASK_Q (высший приоритет), SERVER_Q (средний уровень приоритета) и USER_Q (наименьший приоритет). Эти имена введены для улучшения читаемости кода, а при компиляции подменяются числовыми значениями. Наконец, последней строкой const.h определяется макрос printf, который будет разворачиваться как printk. Это позволяет ядру выводить на консоль сообщения при помощи имеющихся в ядре процедур, что необходимо для обхода обычного механизма вывода сообщений, который требует передачи сообщения сначала файловой системе, а затем из файловой системы — задаче принтера. Дело в том, что при сбое системы такой механизм может не работать. Пример вызова printf смотрите в процедуре с именем panic, которая, как несложно понять по ее имени, вызывается в случае фатальных сбоев.

В файле type.h определены несколько прототипов и структур, нужных для любой реализации MINIX. Структура tasktab описывает отдельную запись массива tasktab, а поля структуры memory содержат два значения, однозначно определяющих область памяти. Здесь нужно упомянуть некоторые концепции, имеющие отношение к организации памяти. Минимальный блок памяти называется кликом (click). В MINIX для процессоров Intel эта величина равна 256 байт. Память измеряется в phys_clicks, удобных ядру для обращения к любой ячейке памяти по всей системе, или в vir_clicks, если к памяти обращается какой-либо другой процесс помимо ядра. Ссылка на ячейку памяти, хранящаяся в vir_clicks, указывает ячейку относительно базового сегмента памяти, присвоенного конкретному процессу. Это неудобство окупается тем, что каждый процесс имеет собственную область памяти, на которую он вправе ссылаться через vir_clicks. Последнее позволяет при обращении к памяти проверять, что процесс не выходит за границы области памяти, выделенной специально для него. Такая защита памяти — одна из основных особенностей *защищенного режима* современных процессоров Intel. Отсутствие защищенного режима в ранних процессорах, таких как 8086 и 8088, вызывало немало головных болей у разработчиков первых версий MINIX.

В type.h находится описание нескольких типов, также зависящих от аппаратной реализации. Это типы port_t, segm_t и reg_t. В версии для процессоров Intel

они используются соответственно для обращения к портам ввода/вывода, сегментам памяти и регистрам процессора.

Структуры также могут зависеть от целевой платформы. Например, существует структура stackframe_t, определяющая то, как машинные регистры сохраняются в стеке, для процессоров Intel. Это чрезвычайно важная структура — она играет свою роль при сохранении и восстановлении состояния процессора, когда процесс приостанавливается или возвращается в состояние выполнения. То, насколько эффективно ассемблерный код будет читать или записывать поля структуры, во многом обусловливает время, необходимое на переключение между контекстами. Segdesc_s — еще одна структура, относящаяся к процессорам Intel. Она связана с механизмом защиты, обеспечивающим разграничение доступа процессов к памяти.

Для процессоров Intel, которые могут быть как с 16-, так и 32-разрядными регистрами, тип reg_t определен как unsigned. Для процессоров Motorola этот тип описан как u32_t. Таким процессорам также нужна структура stackframe_s, но расположение полей в ней иное, с целью ускорения выполнения ассемблерного кода. Архитектура процессоров Motorola такова, что для них не требуются ни типы seg_t и port_t, ни структура segdesc_s. С другой стороны, для архитектуры Motorola характерны несколько определений, у которых нет аналогов в коде для Intel.

Следующий файл, который мы рассмотрим, proto.h — один из самых длинных заголовочных файлов из всех представляющих для нас сейчас интерес. В нем находятся прототипы всех функций, которые должны быть видны за пределами тех файлов, где они объявлены. Все прототипы описаны при помощи макроса _PROTOTYPE, о котором говорилось в предыдущем разделе, благодаря нему ядро MINIX может быть скомпилировано как классическим компилятором C (Кернига и Ричи), так и более современным, соответствующим стандарту ANSI C, такой компилятор входит в комплектацию MINIX версии 2. Отдельные прототипы зависят от системы, к ним относятся обработчики прерываний и исключений, а также функции на языке ассемблера.

Последний из заголовочных файлов, включаемых в kernel.h, — файл glo.h. В нем мы найдем глобальные переменные ядра. Макрос EXTERN, используемый для описания переменных, уже обсуждался при рассмотрении файла include/minix/const.h. Обычно он разворачивается в единственное ключевое слово extern. Но когда этот заголовочный файл включается в table.c, где установлен макрос _TABLE, макрос EXTERN сбрасывается, поэтому он раскрывается в пустую строку. Смысл файла glo.h в том, чтобы позволить использовать в других файлах глобальные переменные из table.c. А именно: held_head и held_tail, которые являются указателями на начало и конец очереди ожидающих прерываний. Переменная proc_ptr ссылается на запись в таблице процессов, соответствующую текущему процессу. Зная эту запись, мы знаем, куда сохранять значения регистров и состояние процессора при возникновении прерывания. Переменная Sig_procs хранит количество процессов, которым поступили сигналы, еще не переданные менеджеру памяти на обработку.

Ряд записей в glo.h объявлены с ключевым словом extern, а не с макросом EXTERN. К ним относятся: sizes, массив, заполняемый монитором загрузки, таблица задач — tasktab, и стек — t_stack. Это связано с тем, что невозможно совмещение макроса EXTERN с инициализацией переменных в стиле C, поскольку каждая переменная должна инициализироваться всего единожды.

У каждой задачи имеется свой собственный стек внутри t_stack. При обработке прерывания ядро работает с отдельным стеком, который здесь не объявлен, так как к нему обращается только код в языке ассемблера, и глобального доступа к этому стеку не требуется.

Есть еще два широко используемых заголовочных файла, относящихся к ядру, хотя они нужны и не так часто, как те, что включены в kernel.h. Первый — proc.h, описывает структуру proc, являющуюся элементом таблицы процессов. В этом же файле далее определяется сама таблица процессов, как массив таких структур, proc[NR_TASKS + NR_PROCS]. Макроопределение NR_TASKS находится в файле include/minix/const.h, а NR_TASKS — в include/minix/config.h. Вместе эти два макроса задают размер таблицы процессов. Значение макроса NR_PROCS можно менять, если надо построить систему, способную работать с большим количеством пользователей. Так как обращения к таблице процессов происходят часто, а вычисление адреса в массиве требует умножения чисел — довольно медленной операции, то для ускорения доступа к записям организуется массив указателей, pproc_addr.

Каждый элемент таблицы процессов хранит регистры процесса, указатель стека, состояние процесса, карту памяти, предельный размер стека, идентификатор процесса, информацию о времени срабатывания таймера, сообщениях и прочие сведения о процессе. Первая часть каждого элемента таблицы — это структура stackframe_s. Когда процесс переходит в состояние выполнения, его указатель стека восстанавливается по адресу из записи в таблице процессов, и все регистры процессора восстанавливаются из этой структуры. В тех случаях, когда процесс не может выполнить вызов send из-за того, что получатель сообщения не ожидает его, процесс помещается в очередь, на которую указывает поле p_callerq у процесса-получателя. Таким образом, когда получатель делает вызов receive, можно легко найти все процессы, которые послали ему сообщения. Для того чтобы связать элементы очереди в единый список, используется поле p_sendlink.

Когда процесс делает вызов receive, а очередь сообщений пуста, процесс блокируется и идентификатор процесса-инициатора сообщения сохраняется в поле p_getfrom. При этом адрес буфера, в который должно быть помещено сообщение, заносится в поле p_messbuf. Последние три поля у каждой записи в таблице процессов: p_nextready, p_pending и p_pendcount. Первое из них позволяет связать вместе процессы в очереди планировщика, второе представляет собой битовую карту для отслеживания сигналов, еще не переданных менеджеру памяти (если менеджер памяти не находится в ожидании сообщения). Третье поле содержит счетчик этих сигналов.

Флаговые биты в p_flags описывают состояние каждой записи в таблице процессов. Если хотя бы один из них установлен, процесс не может перейти в состояние выполнения. Если элемент таблицы не используется, значит, установ-

лен флаг P_SLOT_FREE. После выполнения вызова fork устанавливается флаг NO_MAP, чтобы дочерний процесс не был запущен до того, как сформирована его карта памяти. Флаги SENDING и RECEIVING индицируют блокировку процесса, соответственно, отправляющего сообщение или ожидающего его поступления. Флаги PENDING и SIG_PENDING фиксируют факт получения сигналов, и, наконец, флаг P_STOP необходим для трассировки процесса при отладке.

Потребность в макросе proc_addr проистекает из того, что в C нельзя использовать отрицательные индексы в массивах, как для указателей. Теоретически индексы в массиве proc должны принимать значения из диапазона [-NR_TASKS ... +NR_PROCS]. К сожалению, в C индексы должны начинаться с 0, поэтому proc[0] соответствует наименьшему отрицательному индексу и т. д. Чтобы было проще отслеживать, какая запись соответствует какому процессу, можно использовать макрос и просто писать:

```
rp = proc_addr(n).
```

при этом в переменную rp будет помещен указатель на процесс с положительным или отрицательным индексом *n*.

Указатель bill_ptr ссылается на процесс, ответственный за процессор. Когда пользовательский процесс вызывает файловую систему и она выполняется, то proc_ptr указывает на процесс файловой системы. Вместе с тем bill_ptr будет указывать на пользовательский процесс, и время, которое работала файловая система, учитывается во времени работы пользовательского процесса.

Два массива, rdy_head и rdy_tail, необходимы для поддержания очередей планировщика. Так, например, на первый процесс в очереди задач указывает rdy_head[TASK_Q].

Теперь рассмотрим другой заголовочный файл, который также включается в большое число различных файлов с исходными кодами, protect.h. Почти все, что есть в этом файле, относится к архитектуре процессоров Intel с поддержкой защищенного режима (это процессоры старше 80286). Детальное рассмотрение Intel-процессоров не входит в задачи книги. Достаточно только сказать, что у них есть ряд внутренних регистров, указывающих на *таблицу дескрипторов* в памяти. Информация из таблицы дескрипторов позволяет узнать, как используются ресурсы системы, предотвращать доступ процесса к памяти, принадлежащей другому процессу и др. В дополнение, архитектура таких процессоров предлагает четыре *уровня привилегий*, из которых в MINIX используются три. Центральная часть ядра, то есть код, имеющий дело с прерываниями и переключением контекста, работает с полномочиями уровня INTR_PRIVILEGE. Процесс с таким уровнем доступа вправе обращаться к любым регистрам и любым ячейкам памяти. Задачи работают с уровнем привилегий TASK_PRIVILEGE, который позволяет им обращаться к устройствам ввода/вывода, но не дает изменять значения некоторых специальных регистров, например указатели на таблицы дескрипторов. Серверы и пользовательские процессы выполняются на уровне USER_PRIVILEGE. Этот уровень запрещает процессам выполнять некоторые инструкции, например инструкции ввода/вывода, управления распределением памяти и смены привилегий. Для тех, кто изучал современные процессоры, концепция нескольких уровней защиты по привилегиям наверняка понятна, но те, кто знакомился

с архитектурой компьютеров по маломощным микропроцессорам, могли не сталкиваться с подобными ограничениями.

В каталоге с кодами ядра есть еще несколько заголовочных файлов, но мы коснемся только двух из них. Прежде всего, это sconst.h, содержащий константы, требуемые ассемблерным кодом. Все они представляют собой величины смещений для доступа к полям структуры stackframe_s, записанные в форме, пригодной для встраивания в ассемблерный код. Поскольку ассемблерный код не обрабатывается компилятором ANSI C, подобные определения проще вынести в отдельный файл. Кроме того, все такие определения зависят от аппаратной платформы, поэтому их изолирование упрощает процесс переноса MINIX на другие системы. Обратите внимание на то, что многие константы в этом файле представлены как предыдущая константа плюс *W*, где *W* — величина машинного слова. Это позволяет использовать один и тот же файл как для 16-разрядных, так и для 32-разрядных версий системы.

Здесь есть потенциальная проблема. Назначение заголовочных файлов в том, чтобы один раз записать правильные определения, а затем во многих местах не задумываться о деталях. Очевидно, повторяющиеся определения, как в файле sconst.h, нарушают это правило. Это, конечно, особый случай, но особым случаям и особое внимание. Когда вносятся изменения в этот файл или в procs.h, нужно следить за тем, чтобы все файлы всегда соответствовали друг другу.

Еще для нас представляет интерес файл assert.h. По стандарту POSIX должна быть доступна функция assert, предназначенная для тестирования значений в процессе выполнения и вывода сообщений об ошибке. Фактически POSIX требует наличия файла assert.h в каталоге include/, и там он действительно есть. Зачем же еще один? Причина в том, что, когда что-то не так с пользовательским процессом, операционная система может выполнить такие действия, как вывод сообщения на консоль. Но если неполадки в ядре, на обычные системные ресурсы рассчитывать нельзя. Поэтому ядро предоставляет собственные процедуры для выполнения assert и вывода сообщений, не зависящие от обычных системных библиотек.

В каталоге kernel/ есть несколько файлов, которые мы еще не рассматривали. Они необходимы для поддержки задач ввода/вывода и будут обсуждаться в следующей главе, которая им и посвящена. Тем не менее, прежде чем перейти к рассмотрению исполняемого кода, взглянем на файл table.c, который компилируется в объектный файл, содержащий глобальные переменные ядра. Определения большинства этих структур мы уже видели в файлах glo.h и proc.h. В начале файла, сразу после директив #include, устанавливается макрос _TABLE. Как уже было сказано ранее, задание этого макроса приводит к тому, что макрос EXTERN разворачивается в пустую строку, и для всех данных, объявленных с директивой EXTERN, выделяется область памяти. Кроме переменных, декларированных в файлах glo.h и proc.h, здесь выделяется место для еще нескольких глобальных переменных, объявленных в tty.h. Последние необходимы задаче терминала.

В дополнение к переменным, объявленным в заголовочных файлах, есть еще два места, где выделяются глобальные ресурсы. Некоторые из определений делаются непосредственно в table.c, как, например, область стека для каждой из

задач. Вычислить полный объем стека помогают макросы ENABLE_XXX (объявленные в include/minix/config.h), соответствующие каждой из необязательных задач. Эти макросы отвечают за то, какие задачи будут представлены в массиве tasktab, состоящем из структур tasktab (src/kernel/type.h). В нем выделяются элементы для каждого из процессов, стартующих в процессе инициализации системы, будь то задачи, серверы или пользовательские процессы (то есть init). Индекс в этом массиве однозначно связывает номер задачи и ассоциированные процедуры запуска. В tasktab также указываются необходимый для каждого из процессов объем стека и строка-идентификатор. Массив tasktab помещен сюда, а не в заголовочный файл, потому что трюк с EXTERN, позволяющий избегать повторных деклараций переменных, не работает для переменных с инициализатором. Другими словами, нельзя где угодно использовать подобный код:

```
extern int x = 3.
```

То же относится и к стеку.

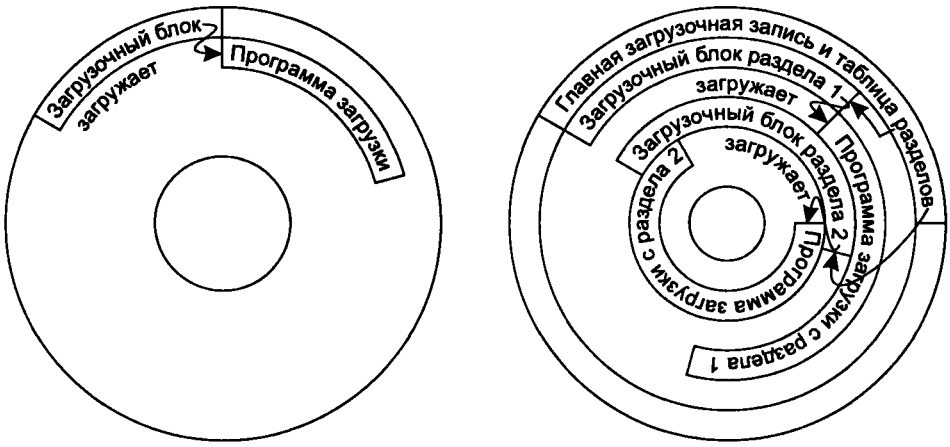
Несмотря на все попытки изолировать друг от друга различные настройки в include/minix/config.h, существует возможность допустить ошибку, приводящую к несоответствию размера tasktab и величины NR_TASKS. Чтобы убедиться, не допущена ли ошибка, в конце файла table.c делается проверка, в основе которой лежит небольшая хитрость. Объявляется массив dummy_tasktab нереализуемого размера, который приводит к ошибке компилятора. Но если этот массив объявляется как extern, место для него здесь не выделяется и ошибки не происходит. Так как других ссылок на массив dummy_tasktab больше нигде нет, он ничем не беспокоит компилятор.

Ассемблерный файл mpx386.s — еще одно место, где распределяется память под глобальные переменные. В этом файле (после метки _sizes) в самое начало сегмента данных ядра помещается сигнатура (магическое число), необходимая для идентификации работоспособного ядра MINIX. Дополнительное место здесь выделяется при помощи псевдоинструкции .space. Такая практика позволяет физически поместить массив _sizes в самом начале сегмента данных ядра, благодаря чему программе boot проще правильно позиционировать свои данные. Монитор загрузки находит сигнатуру и записывает на ее место (то есть в массив _sizes) размеры различных частей ОС MINIX. Затем ядро использует эти данные для инициализации. С точки зрения ядра, этот массив уже инициализирован. Но данные, которые ядро в нем находит, недоступны в момент компиляции. Поэтому монитор загрузки помещает в массив _sizes корректные значения перед тем, как передать управление ядру. Методика, когда одним программам необходимы знания внутренней структуры других программ, несколько необычна. Но момент времени между появлением питания и загрузкой операционной системы необычен сам по себе и требует необычных решений.

2.6.5. Начальная загрузка MINIX

Итак, сейчас почти что настало время перейти к рассмотрению исполняемого кода. Но перед тем, как мы займемся этим, потратим некоторое время на то, чтобы

понять, как MINIX загружается в память. Загрузка, конечно же, производится с диска. На рис. 2.16 показано, как устроены дискеты и жесткие диски, разбитые на разделы.



а *б*

Рис. 2.16. Дисковые структуры, используемые при начальной загрузке: *а* — диск без разбиения на разделы. Первый сектор является загрузочным блоком; *б* — диск, разбитый на разделы. В первом секторе находится главная загрузочная запись

При старте системы аппаратное обеспечение (а в действительности программа из ПЗУ) считывает первый сектор загрузочного диска и исполняет считанный код. На дискете, не разбитой на разделы, первый сектор содержит загрузочный блок, который, в свою очередь, загружает программу `boot`, как показано на рис. 2.16, *а*. В отличие от дискет жесткие диски разбиты на разделы, и в первом секторе находится программа, которая считывает таблицу разделов (из того же первого сектора), а затем загружает и исполняет программу из первого сектора активного раздела, как это показано на рис. 2.16, *б* (один и только один сектор должен быть помечен как активный). Раздел, из которого загружается MINIX, имеет такую же структуру, как и загрузочная дискета MINIX, и в первом секторе находится загрузочный блок.

Реальная ситуация может быть сложнее, чем показано на рисунке, так как отдельные разделы могут быть разбиты на подразделы. В этом случае в первом секторе раздела будет находиться еще одна загрузочная запись, со своей таблицей подразделов. Как бы то ни было, в конце концов управление получит программа из загрузочного сектора на устройстве, которое разделено на разделы. На дискетах, например, загрузочным сектором всегда является первый. MINIX допускает разбиение дискеты на разделы, но в этом случае для загрузки можно использовать только первый раздел, остальные будут недоступны. Такое поведение упрощает монтирование дискет, так как и обычные дискеты, и поделенные на разделы монтируются одинаково. Разбиение на разделы применяется, например,

на инсталляционных дискетах, чтобы разделить образ, копируемый в ОЗУ, и монтируемый раздел, который при необходимости может быть размонтирован, с целью освобождения привода для продолжения процесса установки.

При записи загрузочного сектора на диск в него вписываются номера секторов, необходимые ему, чтобы найти программу boot. Причина в том, что до загрузки операционной системы невозможно найти местоположение нужного файла по его имени. За формирование загрузочного сектора и вписывание в него правильных номеров секторов ответственна программа installboot. Итак, после загрузочного сектора управление получает программа boot. Она уже может не только загрузить саму операционную систему, но и, так как сама является *монитором загрузки*, позволяет пользователю устанавливать, менять и сохранять различные параметры загрузки. Их boot ищет во втором секторе загрузочного раздела. Как требуют стандарты UNIX, MINIX резервирует первый килобайт каждого диска под загрузочный блок, но под загрузчик используется только 512 байт, поэтому еще 512 байт остается для сохранения параметров. Эти параметры управляют процессом загрузки, кроме того, они передаются самой операционной системе. Настройка по умолчанию предлагает пользователю только один вариант — загрузка MINIX, но можно сделать и более сложное меню, при помощи которого пользователь сможет загрузить, например, другие операционные системы (передав управление загрузочному блоку на другом разделе) или же загрузить MINIX, но с другими параметрами. Кроме того, можно настроить загрузчик так, чтобы он пропускал меню и немедленно начинал загрузку MINIX.

Программа boot не является частью операционной системы, но для поиска на диске образа операционной системы достаточно естественно опираться на системные структуры данных. По умолчанию boot ищет файл с именем /minix, а если существует каталог /minix/, то ищет в нем самый новый файл. Конфигурационные параметры позволят изменить это поведение и искать файл с любым наперед заданным именем. Подобная степень свободы имеется не всегда, и для большинства других операционных систем имя загрузочного образа жестко задано. Причина в том, что MINIX — не обычная операционная система, а поощряющая пользователя к ее модификации и созданию новых экспериментальных версий. Осторожность требует того, чтобы оставалась возможность вернуться к прежней версии системы, если новая окажется неработоспособной.

Образ MINIX, который boot загружает в память, представляет собой не что иное, как слитые вместе скомпилированные части ОС: ядро, менеджер памяти, файловую систему и программу init. Каждая из этих частей содержит в своем начале заголовок, формат которого определяется в файле include/a.out.h. Этот заголовок используется boot для того, чтобы определить, какой объем памяти нужен каждой из составляющих системы под неинициализированные данные. Копия этой информации помещается в упомянутый в предыдущем разделе массив _sizes, с той целью, чтобы ядро также знало о положении различных его частей. Области памяти, доступные для загрузочного сектора, самой программы boot и MINIX, зависят от аппаратного обеспечения системы. Кроме того, для некоторых архитектур может потребоваться коррекция кода, чтобы он мог работать, будучи размещенным по тому адресу, куда был загружен. Так как детали процесса загрузки

во многом зависят от платформы, а программа boot частью операционной системы не является, мы не будем далее углубляться в эти вопросы. Важно лишь то, что операционная система тем или иным образом загружается в память, после чего управление передается ядру.

Отступая в сторону, мы должны заметить, что операционные системы не всегда загружаются с диска. *Бездисковые рабочие станции* могут загружаться с удаленной системы через сетевое соединение. Конечно, это требует наличия в ПЗУ программного обеспечения для работы с сетью. В таком случае процесс загрузки будет аналогичен описанному выше, хотя детали могут быть другими. Программа в ПЗУ должна уметь получить через сеть исполняемый код, который, в свою очередь, загрузил бы всю систему. При таком способе загрузки MINIX в процесс инициализации, который начинается сразу после загрузки системы в память, необходимо внести ряд небольших изменений. И конечно же, потребуются сетевой сервер и модифицированная файловая система, способная обращаться к файлам по сети.

2.6.6. Инициализация системы

ОС MINIX для компьютеров семейства IBM PC может быть скомпилирована в 16-разрядном режиме (для совместимости со старыми процессорами) или в 32-разрядном, который позволяет достичь большей производительности на процессорах 80386 и старше. В компиляции участвует один и тот же код на C, а то, какая из версий системы будет получена на выходе, зависит от того, является ли сам компилятор 16- или 32-разрядным. Компилятор собственноручно определяет макрос _WORD_SIZE из файла include/minix/config.h. Первая часть исполняемого кода MINIX написана на языке ассемблера, поэтому для 16- и 32-разрядных версий должен использоваться разный код. В частности, 32-разрядная версия начального кода системы находится в файле mpx386.s. Альтернативная версия для 16-разрядных систем — в mpx88.s. Обе эти версии также включают в себя поддержку различных низкоуровневых операций для ядра. Выбор того, какой из файлов будет использоваться, производится автоматически в файле mpx.s. Этот файл состоит всего из нескольких строк кода и целиком приведен в листинге 2.13.

Листинг 2.13. Как выбирается одна из альтернативных версий ассемблерного кода

```
#include <minix/config.h>
#if _WORD_SIZE == 2
#include "mpx386.s"
#else
#include "mpx88.s"
#endif
```

Здесь демонстрируется необычное использование директивы препроцессора #include. Обычно эта директива применяется для подсоединения заголовочных файлов, но, как показано на листинге, может использоваться и для выбора одной

из альтернативных секций исходного кода. Оперируя лишь директивами #if, весь код, как для 32-разрядной, так и для 16-разрядной версий, пришлось бы поместить в один файл. Это не только громоздко, но и приводит к расточению дискового пространства, так как в конкретных установочных пакетах может быть нужна только одна из версий, в то время как вторая могла бы быть помещена в архив или удалена. В последующем обсуждении мы будем рассматривать в качестве примера только 32-битную версию (mpx386.s).

Так как это первый раз, когда мы приступаем к рассмотрению исполняемого кода, то начнем мы его с того, что скажем несколько слов о порядке такового рассмотрения во всей остальной книге. Большие программы на С состоят из множества файлов, которые трудно выстроить в какой-то последовательности. Как правило, мы будем рассматривать файлы по одному, последовательно их перебирая. Когда нам будет встречаться вызов вспомогательной функции, мы скажем о ней несколько слов, но подробное разбирательство отложим до того времени, когда начнем рассмотрение соответствующего файла. Важнейшие вспомогательные функции обычно объявляются в том же файле, в котором они используются, но небольшие или многоцелевые функции чаще сгруппированы в отдельных файлах. Кроме того, в отдельные файлы вынесены машинно-зависимые функции, ради того чтобы упростить перенос системы на другие платформы. Немало усилий было брошено на то, чтобы организовать код, и фактически многие файлы были переделаны в процессе написания этой книги, только лишь для того, чтобы лучше представить их читателю. Но в больших программах всегда имеется множество ответвлений, и зачастую, для того чтобы понять работу основной программы, требуется рассмотреть те функции, которые она вызывает. Поэтому и вам может быть полезно сделать несколько закладок и отклониться от нашего порядка повествования.

Изложив общий порядок обсуждения кода, обратим внимание на важное исключение. Процесс запуска MINIX включает в себя несколько передач управления между функциями на С и ассемблерными процедурами из mpx386.s, поэтому рассмотрение будет вестись в той последовательности, в какой вызываются функции.

После того как процесс начальной загрузки поместил в память код операционной системы, управление передается по метке MINIX (файл mpx386.s). Первая инструкция выполняет переход вперед на несколько байтов, чтобы обойти область, занимаемую флагами, которые монитор загрузки использует для получения информации о ядре. Самый важный из флагов сообщает, является ли ядро 16- или 32-битным. Монитор загрузки всегда работает в 16-разрядном режиме, но при необходимости может переключить процессор в 32-разрядный режим перед запуском системы. Кроме того, монитор загрузки настраивает стек. Значительная часть работы должна делаться ассемблерным кодом: настройка кадра стека, чтобы создать необходимую среду для кода на С, копирование таблиц, описывающих сегменты памяти, и установка различных регистров процессора. Когда эта работа завершена, управление передается функции cstart, написанной на С. Обратите внимание на то, что в ассемблерном коде ссылка на нее выглядит как _cstart. Это происходит потому, что компилятор каждое имя функции в таблице

символов предваряет знаком подчеркивания, и компоновщику при связывании различных файлов требуются именно такие имена. Так как при использовании ассемблера никаких дополнительных знаков к именам не приписывается, здесь к именам всех функций на C необходимо явно добавлять знак подчеркивания, иначе компоновщик не сможет правильно связать объектные файлы. В свою очередь, cstart вызывает функцию для инициализации *таблицы глобальных дескрипторов* (Global Descriptor Table), центральной структуры данных, используемой 32-разрядными процессорами Intel для защиты памяти, и *таблицы дескрипторов прерываний* (Interrupt Descriptor Table), с помощью которой выбирается обработчик для каждого конкретного прерывания. После возврата из cstart сформированные таблицы активизируются при помощи команд lgdt и lidt. Следующая инструкция:

```
jmpbf CS_SELECTOR:csinit
```

на первый взгляд выглядит как пустая операция, так как она передает управление точно в ту же точку, куда оно бы попало после серии инструкций nop. Но в действительности это важный элемент процесса инициализации, поскольку переход принуждает к использованию только что инициализированных структур данных. Далее следует ряд манипуляций с регистрами процессора, после чего выполняется переход к точке входа ядра main в файле main.c (именно переход, а не вызов). К этому моменту код инициализации в mpx386.s завершил свою работу. Остальная часть ассемблерного кода ответственна за запуск или перезапуск процессов и задач, обработку прерываний и прочие действия, которые для эффективности должны быть написаны на языке ассемблера. Мы вернемся к этим процедурам в следующем разделе.

Теперь перейдем к рассмотрению высокоуровневых функций инициализации на языке C. Общая стратегия такова, чтобы как можно больше использовать высокоуровневый код на C. Как мы уже видели, имеются две версии ассемблерного кода, и все, что может быть перенесено на C, уменьшает ассемблерные включения.

Практически первое, что делается в cstart, — это вызов процедуры prot_init, инициализирующей механизмы защиты памяти и таблицы прерываний. Затем на очереди такие действия, как перенос параметров загрузки в область памяти ядра и преобразование их в числовые значения. Кроме того, определяются тип дисплея, объем памяти, тип компьютера, режим работы процессора (реальный или защищенный), а также то, возможен ли возврат в монитор начальной загрузки. Вся эта информация помещается в соответствующие глобальные переменные, чтобы ее можно было при необходимости использовать в других частях ядра.

Функция main (main.c) завершает инициализацию и переводит систему в режим нормальной работы. Для этого сначала настраиваются средства управления прерываниями с помощью функции intr_init. Данный вызов помещен здесь потому, что он зависит от аппаратного обеспечения и не может быть сделан до того, как определен тип машины. Значение первого аргумента, передаваемое intr_init, указывает подпрограмме инициализировать прерывания для MINIX. Если пере-

дать 0, настройки будут возвращены в исходное состояние. Кроме того, вызов intr_init происходит в два шага, чтобы все произошедшие ранее прерывания не вызвали никаких побочных эффектов. Сначала в каждый контроллер прерываний записывается байт, блокирующий любую реакцию контроллера на внешние данные. Затем обработчиком всех прерываний устанавливается функция, которая просто печатает на экран сообщение о том, что запрещенное прерывание все же произошло. Позже записи в таблице прерываний будут одна за другой заменены указателями на обработчики прерываний по мере того, как запускаются задачи ввода/вывода. Установив обработчик, каждая из задач передает соответствующему контроллеру байт, разрешающий обработку прерываний.

Далее вызывается функция mem_init. Она инициализирует массив, содержащий информацию обо всех блоках памяти, доступных системе. Как и в случае с прерываниями, детали этого процесса зависят от аппаратуры, поэтому функция вынесена в отдельный файл, что освобождает main от непереносимого кода.

Большая часть кода main занимается созданием таблицы процессов, деятельностью, направленной на то, чтобы в момент начала работы планировщика регистры и карты памяти первых задач были бы установлены правильно. Все ячейки таблицы процессов помечаются как свободные, а цикл заполняет массив pproc_addr, введенный для ускорения доступа к процессам. Следующий код:

```
(pproc_addr + NR_TASKS)[t] = rp;
```

можно было бы записать просто как

```
pproc_addr[NR_TASKS + t] = rp;
```

поскольку в C запись a[i] в точности эквивалентна записи *(a+i), и поэтому нет большой разницы, прибавлять ли смещение к адресу массива или к индексу. Но некоторые компиляторы генерируют немного лучший код, если постоянное смещение будет прибавляться к адресу массива.

Самая большая часть кода main, в границах длинного цикла, заполняет таблицу процессов информацией, необходимой для запуска задач, серверов и процесса init. Все эти процессы обязаны существовать к моменту запуска системы, и ни один из них при нормальной работе не должен завершаться. В начале цикла в rp помещается адрес элемента таблицы процессов. Это указатель на структуру, поэтому к полям самой структуры можно обратиться с помощью записи вида rp->имя_поля. Подобная запись широко используется в коде MINIX.

Код всех задач находится в том же файле, что и ядро, а информация о требуемом для них размере стека имеется в массиве tasktab, описанном в файле table.c. Так как сами задачи скомпонованы в один файл с ядром и могут обращаться к коду и данным ядра, действительный размер задач не имеет большого смысла, поэтому поле размера задачи заполняется размером самого ядра. В массиве sizes хранятся размеры (в кликах) текста и данных ядра, менеджера памяти, файловой системы и init. Эта информация вписывается в поля данных ядра программой boot еще до того, как ядро получает управление, и ядро обращается к ним, как если бы эти значения были константами, заданными при компиляции. Первые два элемента массива содержат размеры текста и данных ядра, следующие

два относятся к менеджеру памяти и т. д. Если какая-либо из программ не различает пространства текста и данных, то размер текста устанавливается равным 0, а сегменты склеиваются вместе как данные. В поле sizeindex таблицы процессов сначала записывается значение 0, а в следующих строках в это поле помещается индекс соответствующей записи в таблице sizes.

Компьютеры IBM PC разработаны так, что область ПЗУ расположена в верхней части имеющегося адресного пространства, объем которого для процессоров 8088 составляет 1 Мбайт. У более современных компьютеров памяти почти всегда больше, чем у оригинальных IBM PC, но для совместимости область ПЗУ размещается у них в том же месте — между 640 Кбайт и 1 Мбайт. Таким образом, доступная для записи область памяти не является непрерывной. Монитор загрузки пытается по возможности загрузить серверы и init в верхнюю область памяти, чтобы оставался свободным большой блок доступных адресов. Код в директивах условной компиляции обеспечивает то, что использование верхней области памяти отражается в таблице процессов.

Две ячейки в таблице процессов заняты процессами, которые не должны планироваться обычным путем. Это процессы IDLE и HARDWARE. Первый — IDLE, это просто пустой цикл, который выполняется тогда, когда нет других процессов, готовых к выполнению. Процесс HARDWARE нужен для подсчета процессорного времени — время, которое процессор обслуживал прерывания, присваивается этому процессу. Все остальные процессы причисляются к соответствующим очередям планировщика. Вызываемая при этом функция, lock_ready, устанавливает значение переменной ограничения доступа (семафора), switching, перед изменением очереди планировщика. В данном месте использование семафоров необязательно, но это стандартный метод и нет никаких причин писать здесь специальный код.

Последний этап инициализации — вызов функции alloc_segments. Она принадлежит к числу системных задач, но, конечно, никаких задач в этот момент не запущено. Это машинно-зависимая процедура, заполняющая поля положения, размера и ограничения доступа у каждого из сегментов. Для более старых процессоров Intel, не поддерживающих защищенный режим, определяется только положение сегмента. Если систему нужно будет перенести на процессор, выделяющий память иначе, процедура alloc_segments должна быть переписана.

После инициализации в таблице процессов записей для всех задач, серверов и процесса init система почти готова к работе. В переменной bill_ptr хранится ссылка на процесс, которому в текущий момент передается процессор. Эту переменную необходимо инициализировать каким-то значением, для чего подходит ссылка на процесс IDLE. Впоследствии это значение может быть изменено следующей вызванной функцией, lock_pick_proc. Итак, теперь все задачи готовы к запуску, и ссылка на пользовательский процесс, когда он запустится, будет помещена в переменную bill_ptr. Еще одно назначение функции lock_pick_ptr — поддерживать такое состояние переменной proc_ptr, чтобы она всегда указывала на следующий процесс, который будет запущен. В данном случае, после вызова функции, в переменную proc_ptr будет занесена ссылка на задачу консоли, которая всегда первой получает управление.

Наконец, функция main выполнила свое предназначение. Во многих программах на C функция main представляет собой цикл, но в данном случае ее задача ограничивается инициализацией. Вызов функции restart запускает первую задачу. После чего управление уже никогда не возвращается в main.

Функция _restart представляет собой ассемблерную подпрограмму, код которой находится в mpx386.s. Фактически _restart не является самостоятельной функцией. Это промежуточная точка входа в более сложную процедуру. Подробно мы рассмотрим ее в следующем разделе, а пока скажем лишь, что _restart вызывает переключение контекста и управление получает процесс, на который указывает переменная proc_ptr. В первый раз _restart запускает процесс. Затем эта функция вызывается снова и снова, по мере того как пользовательские процессы, задачи и серверы по очереди получают и теряют управление, либо приостанавливаясь в ожидании ввода, либо по истечении кванта времени.

Первой в очереди задач (то есть задачей с индексом 0, которому соответствует наименьший из отрицательных номеров) всегда стоит задача консоли, чтобы другие задачи могли использовать консоль для вывода сообщений. Эта задача работает до тех пор, пока не блокируется в ожидании сообщения. Затем управление получает следующая задача, также до момента, пока не попадет в состояние блокировки и т. д. В конечном счете, настанет момент, когда все задачи будут приостановлены, и управление смогут получить менеджер памяти и файловая система. Проработав некоторое время, обе эти составные части системы инициализируются и также переходят в состояние блокировки. Далее управление может получить уже процесс init, который запустит несколько дочерних процессов getty, по одному на каждый терминал. Эти процессы также будут заблокированы до ввода какой-либо информации на терминал. Вот теперь пользователь может войти в систему.

Мы рассмотрели процесс запуска MINIX, управляемый кодом из трех файлов, два из которых написаны на C, а один — на ассемблере. В ассемблерном файле, mpx386.s, есть дополнительный код, связанный с обработкой прерываний, обсуждение которого мы отложим до следующего раздела. А пока давайте завершим рассмотрение кратким описанием оставшихся функций из файлов на C. В start.c это процедура k_atoi, преобразующая строку в целое число, и k_getenv, необходимая для получения значений переменных окружения ядра, являющихся копией параметров загрузки.

Обе эти функции представляют собой упрощенные версии аналогичных функций из стандартной библиотеки, переписанных с целью упрощения ядра. В файле main.c осталась нерассмотренной процедура panic, вызываемая в том случае, когда возникает ситуация, делающая невозможной дальнейшую работу системы. Типичными примерами таких ситуаций являются сбои при чтении критических секторов диска, обнаружение недопустимого состояния или внутрисистемный вызов с неправильными данными. Вызовы printf в этой процедуре в действительности вызывают процедуру printk, чтобы ядро могло выводить сообщения даже в случае, когда нарушен нормальный механизм взаимодействия процессов.

2.6.7. Обработка прерываний в MINIX

Особенности обработки прерываний во многом зависят от аппаратной платформы, но у любой системы должны быть элементы с подобной функциональностью. В 32-битных процессорах Intel прерывания генерируются аппаратным обеспечением и в виде электрических сигналов передаются сначала на контроллер прерываний. Это интегральная схема, которая умеет различать несколько подобных сигналов и для каждого из них генерировать уникальный идентификатор на шине данных процессора. У компьютеров с 32-битными процессорами Intel обычно имеется два таких контроллера, каждый из которых обслуживает 8 устройств. Но один контроллер подчинен другому (slave), то есть сигналы с его выхода передаются на вход основного контроллера (master). Таким образом, эта комбинация контроллеров способна обслуживать 15 различных устройств, как показано на рис. 2.17.

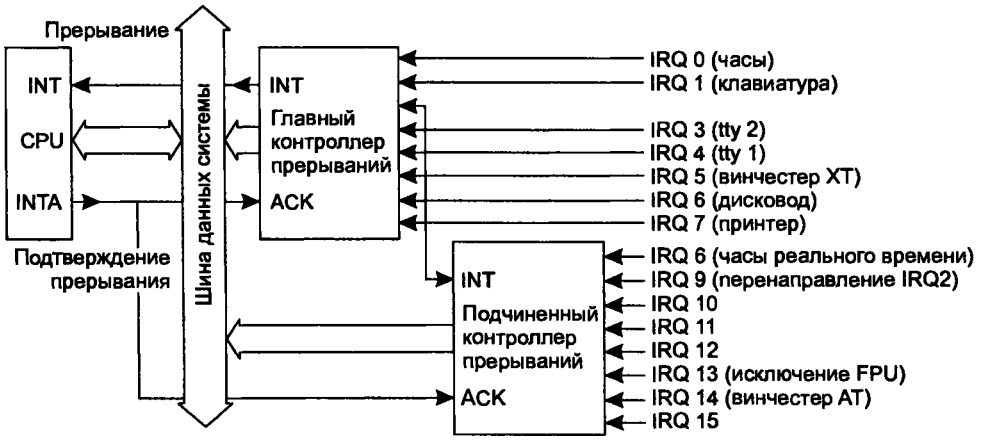


Рис. 2.17. Обработка прерываний в 32-битных процессорах Intel

Здесь прерывания приходят по одной из линий IRQ n, нарисованных в правой части схемы. Процессор получает сигнал о том, что произошло прерывание, по контакту INT. В ответ на это сообщение процессор посылает по контакту INTA (INTerrupt Acknowledge, подтверждение прерывания) сигнал, по которому контроллер помещает в шину данных информацию, говорящую системе о том, какой из обработчиков вызывать. Программируются контроллеры прерываний при инициализации системы в момент, когда функция main вызывает intr_init. При программировании определяется, какие данные контроллер сформирует на шине данных процессора при поступлении сигнала по каждой из входных линий контроллера, а также устанавливается ряд других параметров контроллера. Передаваемые на шину данные представляют собой 8-разрядное число, используемое далее как индекс в массиве максимум из 256 элементов. В MINIX этот массив содержит 56 элементов, из которых реально задействуются 35. Остальные

зарезервированы для будущих версий процессоров Intel или изменений в MINIX. В 32-битной версии системы в записях таблицы содержатся дескрипторы шлюзов прерываний (interrupt gate descriptor). Каждый дескриптор является 8-байтовой структурой с несколькими полями.

Есть несколько режимов обработки прерываний. В том, который принят в MINIX, из нескольких полей дескриптора наибольшее значение для нас будут иметь адрес сегмента памяти, где размещена функция-обработчик, и адрес этой функции внутри сегмента. Получив сигнал прерывания, процессор выполняет код, на который ссылается запись в таблице. Результат полностью эквивалентен ассемблерному вызову

```
int <nnn>
```

Разница только в том, что при аппаратном прерывании адрес процедуры берется не из памяти, а из регистра контроллера.

Механизм переключения задач в 32-битных процессорах Intel, как ответ на прерывание, довольно сложен, и изменение значения счетчика команд — только часть этого процесса. Когда процессор, уже обрабатывающий некоторый процесс, получает прерывание, он сначала выделяет новый стек, который будет использоваться для выполнения обработчика. Положение стека определяется значением записи в *сегменте состояния задачи* (Task State Segment, TSS). Эта структура едина для всей системы и инициализируется при вызове prot_init. Положение стека обновляется при запуске каждого нового процесса так, чтобы новый стек всегда начинался от конца структуры stackframe_s в записи в таблице процессов того процесса, который был прерван. Затем процессор автоматически помещает в новый стек значения нескольких регистров, включая нужные для восстановления стека и счетчика команд прерванного процесса. Обработчик прерывания заносит в стек значения дополнительных регистров, заполняя кадр стека, после чего переключается на другой стек, предоставляемый ядром. Этот стек и используется для всех действий по обработке прерывания.

Завершив свою работу, обработчик прерывания переключается обратно на кадр стека в таблице процессов (не обязательно на тот, который использовался для предыдущего прерывания), затем самостоятельно извлекает из стека значения дополнительных регистров и выполняет инструкцию iretd (возврат из прерывания). Эта инструкция восстанавливает значения регистров, помещенные в стек процессором, и переключается на исходный стек (до прерывания). Таким образом, генерация прерывания останавливает текущий процесс, а по завершении обработчика запускает процесс, причем не обязательно тот, что был приостановлен. Данная схема отличается от более простых алгоритмов, часто встречающихся во многих ассемблерных программах, тем, что в стеке прерванного процесса не сохраняется никаких данных. Более того, так как стек пересоздается в заранее заданном месте (которое определяется записью в TSS), упрощается управление несколькими параллельно работающими процессами. Все, что нужно для запуска нового процесса, — поместить в указатель стека адрес нового кадра стека, извлечь из стека значения регистров, помещаемые туда программно, и выполнить инструкцию iretd.

Получив прерывание, процессор блокирует все остальные прерывания. Это гарантирует, что с кадром стека во время выполнения обработчика не произойдет ничего, что может вызвать переполнение жестко ограниченного кадра стека. Блокировка происходит автоматически, хотя существуют и ассемблерные инструкции, которые могут запрещать или разрешать обработку прерываний. Переключившись на стек, предоставленный ядром, обработчик прерываний снова разрешает прерывания. Конечно, перед тем как обработчик для завершения вновь переключится на фрейм стека в таблице процессов, прерывания опять должны быть запрещены, но в процессе выполнения обработчика прерывания могут обрабатываться. Процессор отслеживает вложенные вызовы обработчиков прерываний и для переключения на новый обработчик и возврата из одного обработчика в другой применяет более простую процедуру. Когда во время обработки прерывания возникает новое прерывание, новый стек не создается. На сей раз существенно важные регистры сохраняются в существующий стек, и осуществляется переход на новый обработчик. Когда он вызывает инструкцию iretd, также используется более простой механизм возврата. Процессор определяет, какой из вариантов будет выполняться, по значению сегмента кода, который при выполнении iretd извлекается из стека.

Различные варианты реакции процессора на прерывания при выполнении кода ядра определяются уровнями привилегий, упомянутыми выше. Когда уровень привилегий прерванного кода совпадает с уровнем обработчика, действует упрощенный механизм. Механизм, ориентированный на TSS и создание нового стека, применяется только в том случае, когда прерванный код менее привилегирован, чем обработчик. Уровень привилегий запоминается в селекторе сегмента кода, который сохраняется в стеке, что и позволяет при возврате из прерывания определять, какой механизм будет применяться. При создании нового стека аппаратно решается еще одна задача. Проверяется, достаточно ли новый стек велик для того, чтобы вместить необходимый минимум информации. Это предотвращает случайный (или намеренный) крах ядра, когда пользователь делает системный вызов при несоответствующем размере стека. Такая функциональность встроена в процессор специально для того, чтобы реализовывать мультипрограммные операционные системы.

Описанное выше поведение может показаться непонятным непосвященным в архитектуру 32-битных процессоров Intel. Обычно мы будем стараться избегать подобных деталей, но для понимания того, как осуществляются переключения между процессами, существенно важно вникнуть в механизм обработки прерываний и работы инструкции iretd. То, что значительная часть работы выполняется аппаратно, упрощает жизнь программиста и, по видимости, делает систему более эффективной. С другой стороны, такая помощь со стороны аппаратного обеспечения усложняет осознание того, что происходит, для того, кто читает код.

Аппаратные прерывания различает только крохотная часть MINIX. Ответственный за это код находится в файле mpx386.s. Для каждого прерывания в этом файле имеется точка входа. Так, обработчики от _hwint00 до _hwint07 вызывают hwint_master, а обработчики от _hwint08 до _hwint15 вызывают hwint_slave. Каж-

дая точка входа передает вызову аргумент, указывающий, какое из устройств требует внимания. Первое, что делает hwint_master, это вызов save. Эта подпрограмма помещает в стек значения вех регистров, необходимых для восстановления прерванного процесса. Ее можно было бы переписать как часть макроса, чтобы увеличить скорость, но это увеличило бы размер макроса более чем в два раза, и, кроме того, подпрограмма save необходима для вызовов, осуществляемых другими функциями. Как мы увидим, save занимается тем, что хитро манипулирует со стеком. После возврата из hwint_master используется стек, выделенный ядром, а не кадр стека в таблице процессов. Следующие действия блокируют контроллер прерываний, чтобы до завершения обработчика не было принято прерывание от того же источника. Это делается путем блокирования способности контроллера отвечать на сигнал по одному из входов; команды процессора для запрета обработки всех прерываний здесь пока не применяются.

Следующие две строки кода сбрасывают контроллер прерываний и подают процессору команду, разрешающую обрабатывать прерывания из других источников. Далее, в следующих двух строках номер обслуживаемого прерывания используется как индекс в таблице адресов обработчиков для косвенного вызова специфичных подпрограмм обработки для каждого из устройств. Мы называем эти подпрограммы низкоуровневыми, но в действительности они написаны на языке C и обычно выполняют такие задачи, как передача данных устройству и копирование блока данных в буфер, который сможет прочитать соответствующая задача, когда получит процессор. До возврата и этого вызова может быть проведено немалое количество вычислений.

Примеры низкоуровневого кода драйверов мы увидим в следующей главе. Тем не менее для понимания того, что происходит в hwint_master, мы упомянем, что низкоуровневый код может вызывать функцию interrupt (ее код находится в proc.c, мы рассмотрим этот файл в следующей главе). Последняя преобразует прерывание в сообщение для задачи, обслуживающей вызвавшее прерывание устройство. Кроме того, interrupt затрагивает планировщик и может сделать так, чтобы нужная задача следующей получила управление. После возврата из специфичного для устройства кода процессору при помощи инструкции cli вновь запрещается обрабатывать прерывания. Контроллер прерываний при этом опять подготавливается к тому, чтобы, как и раньше, отвечать на сигналы от вызвавшего текущее прерывание устройства. Затем hwint_master завершается инструкцией ret. В данный момент происходит нечто не совсем очевидное. Если процесс был прерван, то в этой точке используется стек ядра, а не стек в таблице процессов. Это приводит к тому, что задача, сервер или пользовательский процесс снова получает процессор. Получивший управление процесс может и не быть тем процессом, который был прерван. Более того, это, скорее всего, будет другой процесс. Низкоуровневый код вправе вмешаться в очереди планировщика. Таким образом, мы видим самое сердце механизма, обеспечивающего иллюзию нескольких одновременно работающих процессов.

Для полноты упомянем, что в том случае, когда прерывание происходит в тот момент, когда выполняется код ядра, уже используется стек ядра. Поэтому функция save помещает в стек адрес restart1. Тогда после завершения hwint_master

командой ret работа ядра восстанавливается. Это позволяет делать вложенные прерывания, но когда выполнение низкоуровневых процедур завершается, вызывается _restart, и шанс получить процессор переходит к другому процессу.

Процедура hwint_slave отличается от hwint_master только тем, что в ней необходимо восстанавливать два контроллера, и подчиненный и основной, так как они оба блокируются при приеме подчиненным контроллером сигнала. Здесь вы можете увидеть несколько тонких моментов программирования на ассемблере. Прежде всего обратите внимание на инструкцию

```
jmp .+2
```

Эта инструкция означает переход на следующую команду. Она помещена в код исключительно для того, чтобы добавить небольшую задержку. Разработчики BIOS IBM PC решили, что между последовательными инструкциями ввода/вывода необходима задержка, и мы следуем их рекомендациям, хотя для современных компьютеров это может быть ненужным. Подобные тонкости — одна из причин, по которой некоторые считают работу с аппаратным обеспечением чем-то сродни колдовства. Далее вы можете увидеть инструкцию с числовой меткой:

```
0:  ret
```

По этой метке четырьмя строками позже осуществляется условный переход:

```
jz 0f
```

Запись 0f означает не шестнадцатеричное число, а так называемую *локальную метку*. Здесь это переход вверх по коду к ближайшей инструкции с меткой 0. Обычные метки с чисел начинаться не могут. В этом же файле есть еще один интересный и способный запутать момент, шестью десятками строк выше в hwint_master. Ситуация тут еще более запутана, чем может показаться на первый взгляд, так как метки находятся внутри макросов, которые разворачиваются перед тем, как код передается ассемблеру. Таким образом, ассемблер видит в означенном коде 16 меток 0:. Каждый раз, когда ассемблеру будет встречаться ссылка на подобную метку, будет использоваться ближайшая из них, поэтому различные метки не будут конфликтовать. Возможность применять метки в макросах, несомненно, и явилась причиной существования локальных меток.

Теперь двинемся дальше и рассмотрим процедуру save, уже несколько раз упомянутую ранее. Ее имя описывает ее функции, заключающиеся в том, чтобы сохранять контекст прерванного процесса в выделенный процессором стек (в кадре стека внутри таблицы процессов). Кроме того, для поддержки вложенных прерываний save использует переменную _k_reenter, чтобы подсчитывать число вложений и определять их наличие. Если процесс был прерван, инструкция

```
mov  esp, k_stktop
```

переключается на стек в ядре, а следующая за ней инструкция помещает в стек адрес подпрограммы _restart. В противном случае, если уже используется стек ядра, в него заносится адрес restart1. Независимо от того, какая ветвь алгоритма была выполнена, для возврата из процедуры не годится обычная инструкция ret,

так как положение стека могло быть изменено, а адрес возврата «похоронен» под помещенными в стек регистрами. Поэтому для возврата применяется инструкция

```
jmp RETADR-P_STACKBASE(eax)
```

которую вы можете увидеть в двух точках выхода процедуры. Эта инструкция восстанавливает тот адрес, который был помещен в стек перед вызовом save, и осуществляет переход по нему.

Далее в файле mpx386.s следует процедура _s_call. Прежде чем вдаваться в детали, посмотрим на то, как она заканчивается. Здесь вы не увидите в конце инструкции ret или jmp. После того как командой cli запрещается обработка прерываний, код продолжает выполняться и переходит в _restart. Функция _s_call является двойником механизма обработки прерываний для системных вызовов. Эта подпрограмма получает управление при возникновении программного прерывания, то есть в результате срабатывания инструкции int <nnn>. Программные прерывания обрабатываются так же, как и аппаратные, за исключением того, что индекс записи в таблице дескрипторов прерываний берется из инструкции, а не передается контроллером. Таким образом, когда _s_call получает управление, процессор уже переключен на стек в таблице процессов и в него помещены значения нескольких регистров. Так как у подпрограммы _s_call нет в конце команды выхода, выполнение переходит в _restart, после чего подпрограмма окончательно завершается инструкцией iretd. В результате, как и в случае с аппаратным прерыванием, запускается процесс, на который в данный момент ссылается указатель proc_ptr. На рис. 2.18 сравниваются механизмы обработки аппаратного прерывания и системного вызова, использующего программные прерывания.

Рис. 2.18. *а* — обработка аппаратного прерывания; *б* — так происходит системный вызов

Рассмотрим _s_call более подробно. Еще одна метка, _p_s_call, — специфика 16-битной версии MINIX, в которой имеются раздельные процедуры для защищенного и реального режимов работы. В 32-битной версии вызов по любой из меток приводит к одному результату. Когда программист на С программирует системный вызов, он пишет код, выглядящий как обычный вызов функции, локальной или библиотечной. Поддерживающая системные вызовы библиотечная подпрограмма подготавливает соответствующее сообщение, помещает идентификатор процесса и адрес сообщения в регистры процессора и вызывает инструкцию int SYS386_VECTOR. Как было сказано выше, такая инструкция инициирует программное прерывание и управление передается подпрограмме _s_call, перед вызовом которой в стек (в таблице процессов) заталкивается ряд регистров.

Первая часть процедуры _s_call напоминает код save с раскрытыми макросами. Как и в коде save, процессор переключается на стек в ядре инструкцией

```
mov   esp, k_stktop
```

и разрешаются прерывания. (Сходство программных и аппаратных прерываний проявляется и в том, что в обоих случаях прерывания перед входом в обработчик запрещаются.) Далее следует вызов процедуры _sys_call, которую мы рассмотрим разделом позже. Сейчас мы скажем только то, что она приводит к доставке сообщения и, следовательно, запускает планировщик. Таким образом, когда _sys_call выполняет возврат, указатель proc_ptr может не указывать на тот процесс, который сделал системный вызов. Далее, перед тем как передать управление в restart, инструкция cli запрещает прерывания, во избежание переполнения кадра стека запускаемого процесса.

Мы видим, что вызов _restart происходит в трех случаях:

1. В функции main при запуске системы.
2. При выполнении перехода в функциях hwint_master или hwint_slave после аппаратного прерывания.
3. После выполнения системного вызова, за счет того, что _s_call не содержит завершающей инструкции.

В любом из трех случаев прерывания перед вызовом _restart запрещаются. Если _restart обнаруживает, что во время обработки прерывания были задержаны и остались необслуженными другие прерывания, вызывается процедура unhold. Это позволяет преобразовать прерывания в сообщения перед тем, как какой-либо из процессов вновь будет запущен. Процедура unhold временно разрешает прерывания, но перед возвратом из нее прерывания вновь запрещаются. Таблица процессов разработана так, чтобы начинаться с кадра стека, поэтому инструкция

```
mov   esp, (_proc_ptr)
```

заносит в регистр указателя стека ссылку на кадр стека в таблице процессов. Далее инструкция

```
lldt   P_LDTSEL(esp)
```

загружает значение регистра таблицы локальных дескрипторов из кадра стека. Такая инструкция заставляет процессор использовать сегменты памяти, принад-

лежащие тому процессу, который будет запущен. Следующая инструкция загружает из записи очередного процесса адрес кадра стека, который будет использоваться при возникновении следующего прерывания, и помещает этот адрес в сегмент состояния задачи (TSS). Первая часть подпрограммы _restart не нужна в том случае, если прерывание произошло в момент работы кода ядра, поскольку стек ядра уже используется, и после завершения обработчика выполнение кода должно продолжиться с прерванного места. Метка restart1 отмечает инструкцию, с которой в данном случае необходимо продолжить выполнение. В этом месте декрементируется значение переменной k_reenter, обозначая тем самым новый уровень вложения прерываний, а последующие команды восстанавливают состояние процессора. Завершающие инструкции модифицируют стек так, чтобы игнорировался адрес возврата, помещенный в стек при вызове save. Если прерывание произошло во время выполнения пользовательского процесса, завершающая инструкция iretd передает управление следующему процессу в очереди планировщика. Но если управление было передано через restart1, то есть задействован не кадр стека, а стек ядра, это означает, что после завершения совсем не нужно переходить на новый процесс, а требуется завершить выполнение прерванного кода ядра. Процессор отслеживает подобную ситуацию, когда извлекает дескриптор сегмента кода из стека при выполнении iretd, и, обнаружив ее, оставляет в использовании стек ядра.

В файле mpx386.s есть еще несколько достойных обсуждения моментов. В дополнение к программным и аппаратным прерываниям, различные ошибки выполнения могут вызвать возникновение *исключения*. Исключения — это не всегда плохо. Они полезны, чтобы побудить систему предоставить некоторые дополнительные услуги, например выделить программе дополнительную память или загрузить в оперативную память страницу, перемещенную в область подкачки (хотя в MINIX подобные сервисы не предусмотрены). Но, как бы то ни было, когда исключение возникает, его нельзя игнорировать. Обрабатывают исключения так же, как и прерывания, то есть через дескрипторы в таблице дескрипторов прерываний. В этой таблице имеется шестнадцать записей, содержащих указатели на точки входа обработчиков исключений, начиная с _divide_error и заканчивая _copr_err, которую можно увидеть в конце файла mpx386.s. Каждая из этих точек входа передает управление процедуре exception или errexception, в зависимости от того, помещается ли при исключении в стек код ошибки или нет. Обработка во многом сходна с уже рассмотренным нами кодом: значения регистров сохраняются в стеке и вызывается функция _exception на языке C (обратите внимание на знак подчеркивания перед именем). Некоторые исключения игнорируются, некоторые вызывают сообщение о сбое ядра (kernel panic), другие посылают сигналы процессам. Самой функцией _exception мы займемся в следующем разделе.

Существует еще одна точка входа, которая обрабатывается как прерывание, это _level0_call. Ее мы отложим до следующего раздела, когда будем обсуждать код, на который она переходит: _level0_func. Эта точка входа находится в файле mpx386.s вместе с прерываниями и исключениями потому, что она также вызывается при помощи инструкции int. Как и обработчики исключений, она выполняет вызов save, а завершается инструкцией ret, ведущей к _restart. Последняя

функция файла mpx386.s — _idle_task. Это пустой цикл, отрабатывающий тогда, когда нет других готовых к выполнению процессов.

Наконец, в конце ассемблерного файла выделяется место для хранения некоторых данных. Определяются два различных сегмента данных. Декларация

```
.sect.rom
```

гарантирует, что эта область памяти находится в самом начале сегмента данных ядра. Сюда компилятор помещает сигнатуру (магическое число), чтобы программа boot могла убедиться, что загружает действительно ядро. После загрузки программа boot перезаписывает на это место данные из массива _sizes, как это уже было показано нами ранее, при обсуждении структур данных ядра. Поэтому выделяемая память должна вмещать массив _sizes, который может содержать до шестнадцати элементов (в случае, если включены дополнительные серверы). Еще одна область данных задается директивой

```
.sect.bss
```

резервирующей неинициализированную область для стека ядра и для переменных, используемых обработчиками прерываний. Размер стека для пользовательских процессов и серверов задается при их компоновке, и при их выполнении ядро устанавливает нужное значение дескриптора сегмента стека. О себе ядро должно заботиться само.

2.6.8. Взаимодействие между процессами в MINIX

В MINIX процессы взаимодействуют друг с другом при помощи сообщений, через механизм, называемый *рандеву*. Когда процесс делает системный вызов send (отправка сообщения), нижний уровень ядра проверяет, ожидает ли адресат сообщений от отправителя (или от *любого другого* процесса). Если это так, сообщение копируется из буфера отправителя в буфер получателя и оба процесса помечаются как готовые к выполнению. Если получатель не ждет сообщений, отправитель блокируется и помещается в очередь процессов, ожидающих отправки сообщения.

Когда процесс делает системный вызов receive, ядро проверяет, есть ли в очереди ожидающих отправки процессов пытающиеся отправить сообщение текущему. Если таковые есть, сообщение передается из буфера отправителя в буфер адресата, и оба процесса выходят из состояния блокировки. Если ожидающих отправителей нет, процесс-получатель приостанавливается до прибытия сообщения.

Высокоуровневый код для обеспечения обмена информацией между процессами находится в proc.h. Задача ядра состоит в том, чтобы преобразовать аппаратное или программное прерывание в сообщение. Первые генерируются аппаратным обеспечением компьютера, а вторые служат для передачи ядру запросов, в данном случае — системных вызовов. Оба варианта достаточно похожи и для их обработки можно было бы использовать одну функцию, но более эффективно создать две специализированные.

Сначала рассмотрим функцию interrupt. Она вызывается низкоуровневой подпрограммой, обслуживающей аппаратные прерывания. Ее назначение в том, чтобы преобразовать прерывание в сообщение обслуживающей устройство задаче. Производимая перед вызовом обработка информации весьма невелика. Например, весь низкоуровневый код обработчика прерываний для драйвера жесткого диска можно уместить в три строки:

```
w_status = in_byte(w_wn->base + REG_STATUS); /* подтверждение прерывания */
interrupt(WINCHESTER);
return 1;
```

Если бы не требовалось считывать данные с порта ввода/вывода дискового контроллера, чтобы получить состояние, то вызов interrupt можно было бы поместить прямо в mpx386.s, а не в at_wini.c. Первое, что делает interrupt, — проверяет, обслуживались ли другие прерывания в тот момент, когда произошло текущее, по значению переменной k_reenter. Если такие прерывания имеются, текущее прерывание ставится в очередь и выполняется возврат из функции. Поставленное в очередь прерывание будет обслужено позже, когда будет сделан вызов unhold. Далее делается проверка, ожидает ли задача прерывания. Если задача не готова, выставляется флаг p_int_blocked и сообщение не отправляется. Как мы увидим позже, установка этого флага позволяет восстановить необслуженные прерывания. Отправить сообщение от аппаратуры задаче несложно, так как задачи и ядро скомпонованы в один файл и могут обращаться к структурам данных друг друга. Отправляемое сообщение копируется в приемный буфер ожидающей задачи, у которой сбрасывается флаг RECEIVING, с целью ее разблокировать. После того как сообщение отправлено, получившая его задача ставится на первое место в очереди планировщика. Более подробно мы обсудим работу планировщика в следующем разделе, а сейчас вы можете рассматривать дальнейший код в interrupt как предварительный пример, поскольку он соответствует процедуре ready, ставящей процесс в очередь. Этот код проще, так как здесь сообщения могут отправляться только задачам, и нет необходимости определять, какую из трех очередей модифицировать.

Далее в файле proc.c следует функция sys_call. Ее назначение аналогично назначению interrupt, поскольку эта функция преобразует программное прерывание (инструкция int SYS386_VECTOR служит для инициации системного вызова) в сообщение. Так как в данном случае диапазон возможных адресатов сообщений более широк, а также может потребоваться либо отправка, либо прием сообщения, либо и то и другое, то у sys_call работы больше. Как это часто бывает в подобных ситуациях, это означает, что код sys_call проще и понятней, все-таки большую часть своих задач эта функция выполняет, вызывая другие процедуры. Первый такой вызов — isoksrc_dest, макрос, определенный в proc.h. Этот макрос использует другой макрос, isokprocn, который также задан в файле proc.h и проверяет, корректно ли указан адресат или отправитель сообщения. Далее следует макрос isuserp, нужный, чтобы убедиться, что пользовательский процесс собирается отправить сообщение и ждать ответа — единственное, что разрешено пользовательским процессам. Нарушения этих условий маловероятны, но проверку сделать просто, поэтому проверяющий код сводится к сравнению небольших

целых чисел. Рассматриваемый код относится к базовому уровню системы, на котором желательно делать все возможные операции контроля. Этот код будет выполняться много раз каждую секунду, пока система запущена.

Наконец, если вызов требует отправки сообщения, вызывается функция mini_send, а если требуется принять сообщение, вызывается mini_rec. Эти две функции — сердце механизма обработки сообщений и заслуживают тщательного изучения.

У mini_send три входных параметра: отправитель, адресат и указатель на буфер, в котором расположено сообщение. Функция проводит ряд проверок входных данных. Прежде всего, удостоверяется, что пользовательский процесс пытается отправить сообщение либо файловой системе, либо менеджеру памяти. Значение аргумента caller_ptr тестируется макросом isuserp, который убеждается, что отправитель является пользовательским процессом. Аргумент dest тестируется при помощи аналогичного макроса, issysentn, проверяющего, является ли процесс файловой системой или менеджером памяти. Если обнаружено несоответствие, mini_send завершается с ошибкой.

Далее проверяется, что адресат — активный процесс, а не пустая ячейка в таблице процессов. Затем выполняется контроль, помещается ли сообщение целиком в сегмент данных приемника, сегмент кода или в промежуток между ними. Если нет, возвращается код ошибки.

Следующий тест оценивает, не попадут ли процессы в состояние взаимной блокировки. Для этого проверяется, не пытается ли получатель сам отправить сообщение отправителю.

Следующая проверка является ключевой для mini_send. А именно, проверяется, находится ли адресат в состоянии блокировки после вызова receive. Для этого анализируется значение бита RECEIVING в поле p_flags дескриптора процесса. Если получатель ждет, выясняется, от кого он ждет сообщение. Если объектом интереса является отправитель или ANY, срабатывает макрос CopyMess, что приводит к копированию данных в буфер адресата и разблокированию последнего. Этот макрос вызывает ассемблерную процедуру cp_mess из файла klib386.s.

Если же адресат не блокирован или блокирован, но ждет сообщение от другого процесса, выполняется код, переводящий в состояние блокировки уже отправителя. Все процессы, ожидающие отправки сообщения одному и тому же адресату, связываются вместе в список, на голову которого указывает поле p_callerq адресата. Пример на рис. 2.19, *а* показывает, что произойдет, если процесс 3 не сможет отправить сообщение процессу 0. Если при этом процесс 4 также не может отправить сообщение процессу 0, возникает ситуация, продемонстрированная на рис. 2.20, *б*.

Перейдем к рассмотрению функции mini_rec. Она вызывается, когда sys_call вызывается с параметром RECEIVE или BOTH. Сначала цикл ищет среди процессов, ожидающих отправки, нужного отправителя. Если один из них удовлетворяет условиям, сообщение копируется из буфера отправителя в буфер приемника, затем отправитель выводится из блокировки и исключается из очереди процессов, ждущих отправки сообщения.

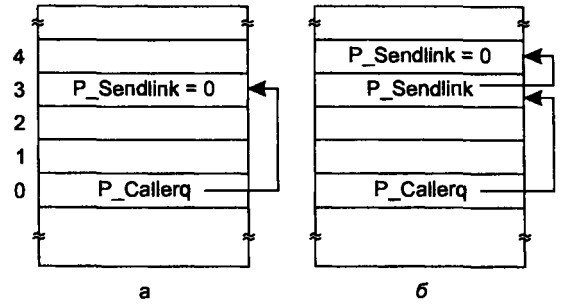


Рис. 2.19. Очереди процессов, ожидающих отправления сообщения процессу 0

Если ни одного подходящего отправителя не найдено, делается проверка флага p_int_blocked, который означает, что ранее было заблокировано прерывание для данного получателя. Если это так, создается новое сообщение (так как сообщения от HARDWARE не имеют содержимого и всегда относятся к типу HARD_INT, копировать буфер в этом случае не нужно).

Если заблокированного прерывания не обнаружено, адрес приемного буфера сохраняется в таблице процессов, а сам процесс переводится в состояние ожидания путем установки бита RECEIVING. Вызов unready исключает данный процесс из очереди готовых к запуску процессов. Перед вызовом unready делается проверка, не установлены ли у него другие флаги в p_flags. У процесса могут иметься текущие сигналы, и если это так, ему нужна возможность обработать их.

Предпоследнее выражение в mini_rec связано с тем, как обрабатываются генерируемые ядром сигналы SIGINT, SIGQUIT и SIGALARM. Когда генерируется один из этих сигналов, менеджеру памяти посылается сообщение, если он ждет уведомления от ANY. Если менеджер памяти такого сообщения не ждет, сигнал запоминается в ядре до тех пор, пока MM не попытается получить сообщение от ANY. В коде делается проверка этого условия и при необходимости вызывается inform.

2.6.9. Планирование процессов в MINIX

В MINIX применяется многоуровневый алгоритм планирования, структура которого близка к показанной на рис. 2.20. На этом рисунке мы видим задачи ввода/вывода на уровне 2, сервера на уровне 3 и пользовательские процессы на уровне 4. У планировщика есть три очереди готовых к работе процессов, по одной на каждый слой, как показано на рисунке. Массив rdy_head содержит по одному элементу на каждую из очередей, в котором хранится указатель на начало соответствующей очереди. Аналогично, массив rdy_tail хранит указатели на конец каждой из очередей. Оба этих массива описаны в proc.h при помощи макроса EXTERN.

Каждый раз, когда приостановленный процесс вновь приходит в состояние готовности, он помещается в хвост соответствующей очереди. Эффективное добавление элемента в хвост списка обеспечивает массив rdy_tail. А когда выпол-

няемый процесс блокируется или завершается по сигналу, он удаляется из очереди, так как в ней находятся только готовые к запуску процессы.

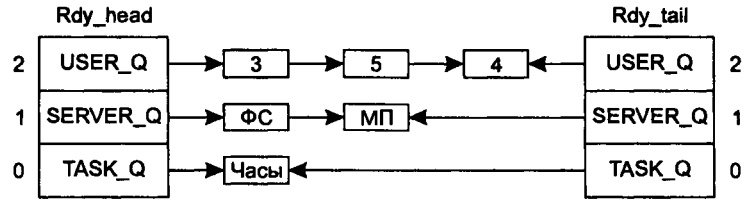


Рис. 2.20. Три очереди планировщика, по одной на каждый уровень приоритета

Алгоритм диспетчеризации с описанной выше структурой очередей прост. Берется непустая очередь с наивысшим приоритетом, в которой выбирается первый процесс. Если же все очереди пусты, выбирается подпрограмма бездействия. На рис. 2.20 наибольший приоритет у очереди TASK_Q. Код планировщика находится в файле proc.c. Очередь выбирается в функции pick_proc, основная задача которой — установить значение proc_ptr. Если над очередями производятся какие-либо действия, которые могут повлиять на выбор следующего готового процесса, то pick_proc необходимо вызвать снова. Эта функция вызывается каждый раз, когда блокируется один процесс, с целью поставить на выполнение следующий.

Сама функция pick_proc по простоте сродни алгоритму. Сначала проверяется каждая из очередей: в «первую очередь» TASK_Q, и при наличии в ней процесса устанавливается значение proc_ptr, и функция немедленно завершается. Далее проверяется очередь SERVER_Q, и, опять же, если она не пуста, устанавливается proc_ptr и функция завершается. Затем, если в очереди USER_Q имеются готовые процессы, устанавливается значение переменной bill_ptr, чтобы обозначить, что за процессорное время, которое будет отдано процессу, отвечает именно он. Если ни в одной из очередей нет готовых к работе процессов, следующая строка перекладывает учет времени на процесс IDLE, который и выбирается на выполнение планировщиком. Выбранный процесс не исключается из очереди.

Для управления очередями служат процедуры ready и unready, первая из них предназначена для постановки процесса в его очередь, а вторая убирает процесс из очереди готовых к запуску. Как мы видели, ready вызывается из mini_send и из mini_rec. Кроме того, ready можно было бы использовать в функции interrupt, но по соображениям производительности код ready встроен в pick_proc. Функция ready модифицирует одну из трех очередей, добавляя новый процесс в хвост очереди.

Функция unready также управляет очередями. В обычной ситуации она удаляет процесс из хвоста очереди, так как, чтобы попасть в состояние блокировки, процесс должен выполняться. В этом случае unready перед тем, как завершиться, вызывает pick_proc. Процесс может попасть в состояние блокировки и в результате сигнала. Тогда, если процесс не найден в хвосте очереди, он ищется во всей цепочке USER_Q.

Хотя большая часть действий по планированию процессов происходит, когда процесс блокируется или приходит в готовность, планирование необходимо и тогда, когда задача таймера сообщает, что квант времени текущего процесса истек. В данном случае задача таймера вызывает функцию sched, которая перемещает процесс из начала очереди USER_Q в конец. Благодаря такому алгоритму пользовательские процессы выполняются по кругу (алгоритм циклического планирования). Файловая система, менеджер памяти и задачи ввода/вывода никогда не перемещаются в конец очереди. Им оказывается доверие, и блокируются они тогда, когда завершают свою работу.

В proc.c есть еще несколько процедур, необходимых для планирования. Пять из них: lock_mini_send, lock_pick_proc, lock_ready, lock_unready и lock_sched устанавливают значение переменной-семафора switching перед вызовом соответствующей функции, а затем освобождают семафор после завершения функции. Последняя функция в файле, unhold, упоминалась при обсуждении _restart в mpx386.s. Она просматривает очередь задержанных прерываний, вызывая для каждого из них interrupt, чтобы все прерывания были преобразованы в сообщения, до того как какой-либо из процессов запустится.

Итак, алгоритм планирования использует три очереди с разными приоритетами, для задач ввода/вывода, серверов и пользовательских процессов. Следующим всегда запускается тот процесс, который ждет первым в очереди. Задачи и серверы всегда работают до тех пор, пока не попадут в состояние блокировки, а для пользовательских процессов выделяются ограниченные кванты времени. Если процесс исчерпывает свой квант, он перемещается в конец очереди, тем самым обеспечивается простой круговой механизм планирования пользовательских процессов.

2.6.10. Аппаратно-зависимая поддержка в ядре

В системе есть несколько функций на C, которые очень существенно зависят от аппаратной платформы. Чтобы способствовать переносу MINIX на разные платформы, такие функции выделены в отдельные файлы: exception.c, i8259.c и protect.c, а не помещены в том же файле, в котором они используются.

Файл exception.c содержит обработчик исключений, функцию exception, которая вызывается из ассемблерной программы обработки исключений в mpx386.s (вызывается как _exception). Когда исключение исходит от пользователя, оно преобразуется в сигнал, так как ошибки в пользовательских программах ожидаемы. Но когда исключение исходит от операционной системы, это признак того, что произошло нечто действительно серьезное. Такое исключение приводит к сообщению о сбое ядра (kernel panic). Сообщение, которое при этом будет выведено на экран, или сигнал, который будет послан приложению, задаются массивом ex_data. Третье поле задает минимальный номер модели процессора, поддерживающего данное прерывание, поскольку круг прерываний в ранних моделях процессоров Intel более узок. Этот массив является интересным индикатором развития семейства процессоров Intel, на которых реализована MINIX.

Три функции из файл i8259.c вызываются при запуске системы для инициализации контроллеров прерываний на чипах Intel 8259. Функция intr_init инициализирует контроллеры, записывая данные в несколько портов. В нескольких строках кода проверяется значение переменной, зависящей от параметров загрузки ядра, с целью учесть различные модели компьютеров. Параметр mine проверяется для того, чтобы записать в порт значение, соответствующее либо MINIX, либо ПЗУ BIOS. Когда MINIX завершает свою работу, функция intr_init может восстановить векторы BIOS, тем самым корректно передавая управление монитору загрузки. Параметр mine определяет, какой режим будет использоваться. Полное понимание того, что происходит в функциях этого файла, требует изучения документации чипа 8259, поэтому мы не будем вдаваться в детали. Отметим лишь, что вызов out_byte, помеченный комментарием «IRQ 0–7 mask», приводит к тому, что основной контроллер перестает откликаться на сигналы подчиненного, а аналогичная инструкция шестью строками далее блокирует реакцию подчиненного контроллера на все его входы. Кроме того, последняя инструкция этой функции загружает в каждую ячейку таблицы irq_table адрес функции spurious_irq. Это гарантирует, что прерывания, которые произойдут до того, как будут назначены реальные обработчики, не причинят вреда.

Последняя функция в файле i8259.c — put_irq_handler. При инициализации каждая из задач ввода/вывода, отвечающая на определенные прерывания, переписывает с ее помощью указатель на функцию-обработчик.

В файле protect.c находятся несколько функций, необходимых для работы защищенного режима процессоров Intel. В памяти выделяются глобальная таблица дескрипторов (Global Descriptor Table, GDT), локальные таблицы дескрипторов (Local Descriptor Table, LDT) и таблица дескрипторов прерываний (Interrupt Descriptor Table, IDT), необходимые для предоставление защищенного доступа к системным ресурсам. Адреса GDT и LDT хранятся в специальных регистрах процессора, а записи в GDT содержат ссылки на отдельные локальные таблицы дескрипторов. GDT должна быть доступна для всех процессов и содержит дескрипторы сегментов для областей памяти, используемые операционной системой. Дескрипторы представляют собой структуры объемом 8 байт, состоящие из нескольких компонентов. Важнейшие из полей хранят адрес и размер области памяти. Таблица дескрипторов прерываний также состоит из 8-байтовых дескрипторов, в которых важнейшее поле хранит адрес кода, который будет выполнен при возникновении прерывания.

Функция prot_init вызывается из start.c, чтобы установить GDT. BIOS IBM PC требует того, чтобы эта таблица была упорядочена определенным образом, для чего в файле protect.h описаны необходимые значения индексов. Область памяти для LDT выделяется в таблице процессов. Каждая такая таблица содержит два дескриптора: один для сегмента кода и один для сегмента данных. Заметьте: сегменты, которые мы обсуждаем, это сегменты, как их воспринимает аппаратное обеспечение системы. Это не то же самое, что сегменты, с которыми работает операционная система. Так, заданный аппаратно сегмент данных будет далее разбит на сегменты данных и стека. Для каждой из LDT создаются дескрипторы, помещаемые в GDT. За это ответственны функции init_dataseg и init_codeseg. За-

писи в самой LDT инициализируются в тот момент, когда меняется карта памяти процесса (то есть когда делается системный вызов exec).

Еще одна системная структура данных, которую необходимо инициализировать, это сегмент состояния задачи (Task State Segment, TSS). Его структура определяется в начале файла и включает в себя области для хранения регистров процессора, а также другой информации, которую необходимо запоминать при переключении задач. В MINIX используются только те поля, которые определяют, где будет создан новый стек в момент возникновения прерывания. Вызов init_dataseg гарантирует, что этот стек может быть найден через GDT.

Чтобы понять, как MINIX работает на нижнем уровне, важнее всего разобраться, как прерывания, исключения и инструкции int <nnn> приводят к исполнению различных процедур, написанных для их обслуживания. Для этого необходима таблица дескрипторов шлюзов прерываний. Массив gate_table заполняется компилятором адресами процедур, которые обрабатывают исключения и аппаратные прерывания. Затем значительная часть этого массива в цикле инициируется при помощи вызовов функции int_gate. Незаполненные векторы, SYS_VECTOR, SYS386_VECTOR и LEVEL0_VECTOR, требуют другого уровня привилегий и заполняются после цикла.

Существует несколько причин, в силу которых данные определенным образом структурированы в виде дескрипторов. Основная мотивация — особенности аппаратного обеспечения и необходимость поддерживать совместимость между современными и старыми 16-битными процессорами. К счастью, детали мы можем оставить разработчикам процессоров Intel. Язык программирования C позволяет по большей части избегать таких мелочей. Тем не менее при разработке операционной системы так или иначе придется с ними столкнуться. Внутренняя структура одного дескриптора сегмента приведена на рис. 2.21. Обратите внимание на то, что базовый адрес, к которому программы на C обращаются как к простому 32-разрядному целому числу, разбит на три части, две из которых разделены 1-, 2- и 4-битными блоками. Размер области является 20-битным числом, которое хранится в виде пары блоков из 16 и 4 бит. Этот размер может интерпретироваться либо как количество байтов, либо как количество страниц объемом 4096 байт, в зависимости от значения бита G. У других дескрипторов другая структура, не менее сложная. Более подробно мы обсудим эти структуры в главе 4.

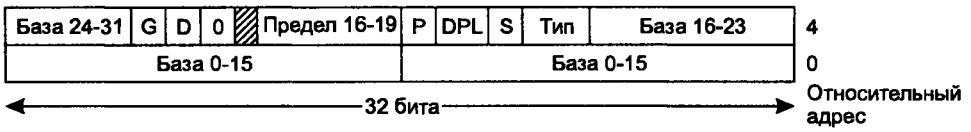


Рис. 2.21. Формат дескриптора сегмента у Intel

Большинство остальных функций из protect.c предназначены для преобразования данных между переменными в программах на C и тем сложным представлением, которое принято в дескрипторах. Функции init_codeseg и init_dataseg работают сходным образом. Их назначение в том, чтобы преобразовать переданные

им данные в дескрипторы сегментов. Обе они, чтобы завершить свою работу, в свою очередь вызывают другую функцию, sdesc. Именно она имеет дело с запутанной структурой, которая показана на рис. 2.21. Функции init_codeseg и init_dataseg вызываются не только при инициализации системы. Помимо этого, они вызываются системой каждый раз, когда запускается новый процесс, чтобы выделить новому процессу необходимые сегменты памяти. Вызов функции seg2phys выполняется только из файла start.c, ее действие обратно действию sdesc — извлечение базового адреса сегмента из дескриптора. Функция int_gate, аналогичная init_codeseg и init_dataseg, заполняет ячейки таблицы дескрипторов прерываний.

Последняя функция в protect.c, enable_iop, необходима для выполнения одного «грязного» трюка. Мы указывали на то, что одно из назначений операционной системы в том, чтобы ограничивать доступ к системным ресурсам, и один из способов, которым это реализуется в MINIX, являются уровни привилегий. Тем не менее MINIX может использоваться и в малых системах, рассчитанных всего на одного-нескольких пользователей, которым оказывается доверие. На таких системах пользователю может потребоваться написать программу, не брезгующую инструкциями ввода/ вывода, например, чтобы накапливать данные эксперимента. У файловой системы есть один меленький секрет, который встроен в ее код. Когда открывается один из файлов /dev/mem или /dev/kmem, задача памяти вызывает функцию enable_iop, которая изменяет уровень привилегий для операций ввода/вывода, позволяя текущему процессу выполнять инструкции, считывающие данные из порта ввода/вывода и записывающие их. Описание того, что делает эта функция, сложнее, чем ее код, который сводится к установке двух битов в одном из слов кадра стека вызвавшего функцию процесса. Это слово будет загружено в регистр состояния процессора при следующем запуске процесса.

2.6.11. Утилиты и библиотека ядра

У ядра есть библиотека вспомогательных функций, написанных на языке ассемблера. Эти функции подключаются при компиляции klib.s. Также имеется несколько вспомогательных программ на языке С, которые находятся в файле misc.c. Сначала мы рассмотрим ассемблерные файлы. Файл klib.s представляет собой короткий ассемблерный файл, сходный с mpx.s. В нем в зависимости от значения WORD_SIZE выбирается одна из версий кода. Код, который мы будем обсуждать, расположен в файле klib386.s. В этом файле содержится около двух дюжин вспомогательных функций, которые написаны на ассемблере либо по соображениям эффективности, либо потому, что они вообще не могут быть реализованы на С.

Функция _monitor позволяет вернуться из системы в монитор загрузки. С его точки зрения, вся операционная система — не более, чем подпрограмма, и, когда MINIX запускается, адрес монитора остается в стеке. Функции _monitor необходимо только восстановить значения селекторов сегментов и указателя стека, после чего выполнить возврат из подпрограммы.

Следующая функция, _check_mem, используется во время запуска, для того чтобы определить величину блока памяти. Она просто побитово тестирует каждый 16-й байт при помощи контрольных двоичных значений.

Хотя функция _cp_mess (см. ниже) могла бы использоваться для копирования сообщений, есть более быстрая специализированная функция, cp_mess. Она вызывается следующим образом:

```
cp_mess(source, src_clicks, src_dest, dest_clicks, dest_offset);
```

Здесь source — это номер процесса отправителя, который копируется в поле m_source буфера адресата. Адреса буферов приемника и отправителя представлены в виде комбинации количества кликов, обычно это базовый адрес сегмента, содержащего буфер, и смещения. Это более эффективная форма представления адресов, чем 32-разрядные целые числа, как в _phys_copy.

Функции _exit, __exit и ___exit необходимы потому, что некоторые из библиотечных процедур, которые могут использоваться при компиляции MINIX, способны делать вызовы стандартной функции языка C exit. Она вызывается, когда необходимо выйти из программы, но выход из ядра не имеет смысла, здесь просто некуда выходить. Эта проблема решена так: функция разрешает прерывания и входит в бесконечный цикл. В определенный момент возникает прерывание таймера или от операции ввода/вывода, и система возвращается к нормальной работе. Точка входа с именем ___main — еще один пример борьбы с поведением компилятора, которое имеет смысл при компиляции пользовательской программы, но бесполезно для ядра. Эта точка входа указывает на ассемблерную инструкцию ret.

Функции _in_byte, _in_word, _out_byte и _out_word предоставляют доступ к портам ввода/вывода, которые в архитектуре процессоров Intel занимают отдельное адресное пространство, поэтому для работы с ними требуются инструкции, отличные от обычных инструкций работы с памятью. Передачей блоков данных между портами и памятью заведуют функции _port_read, _port_read_byte, _port_write и _port_write_byte. Они используются в основном при взаимодействии с диском, которое должно происходить быстрее, чем это позволяют обычные функции ввода/вывода. Байтовые версии этих функций читают по 8 бит, а не по 16, чтобы соответствовать старым 8-битным периферийным устройствам.

Изредка задачам необходима функция, которая временно блокировала бы все прерывания процессора. Это делается путем вызова _lock. Когда прерывания можно вновь разрешить, вызывается _unlock. Каждый из обоих вызовов сводится к одной машинной инструкции. В противоположность этому, функции _enable_irq и _disable_irq более сложно устроены. Они работают на уровне контроллера прерываний и разрешают или запрещают отдельные аппаратные прерывания.

Функция _phys_copy вызывается из программы на C следующим образом:

```
_phys_copy(source_address, destination_address, bytes);
```

Она копирует блок данных из одной области физической памяти в любое другое место. Оба передаваемых этой функции адреса абсолютные, то есть 0 означает

первый байт адресного пространства. Все три аргумента функции являются по типу беззнаковыми длинными целыми.

Две следующие короткие функции очень специфичны для процессоров Intel. Функция _mem_rdw «возвращает» 16-битное слово из произвольного места памяти. Результат дополняется нулями и помещается в регистр eax. Функция _reset сбрасывает процессор. Для этого в дескриптор прерываний процессора загружается нулевой указатель, после чего вызывается программное прерывание. Результирующий эффект равносилен аппаратному сбросу.

Далее идут две функции, поддерживающие работу видеодисплея и вызываемые задачей консоли. Подпрограмма _mem_vid_copy копирует строку, содержащую различные символы и управляющие коды, из области ядра в видеопамять. Функция _vid_vid_copy копирует блок в пределах видеопамяти. Эта функция несколько сложнее, так как возможно перекрывание блоков, поэтому важно учитывать направление копирования.

Последняя функция в файле klib386.s — _level0. Она позволяет задачам при необходимости получать более высокий уровень привилегий — нулевой. На данном уровне разрешены такие действия, как сброс процессора или обращение к подпрограммам BIOS.

Вспомогательные функции на языке C в файле misc.c специализированы. Функция mem_init вызывается только из main при запуске MINIX. У компьютера IBM PC область памяти может быть не непрерывной, а состоять из нескольких блоков. Размеры нижнего диапазона, который известен пользователям PC как «обычная» память, и верхнего, который расположен после области ПЗУ, передаются ядру монитором загрузки через параметры, которые затем интерпретируются функцией cstart и помещаются в переменные low_memsize и ext_memsize. Третья область памяти — это «теневая» память, в которую может копироваться содержимое BIOS из ПЗУ для увеличения производительности, так как ПЗУ обычно медленнее, чем обычная память произвольного доступа. Поскольку MINIX при своей обычной работе не пользуется BIOS, функция mem_init старается обнаружить такую память и сделать ее доступной. Для этого вызывается функция check_mem, проверяющую «пул» памяти, где ожидаемо присутствие высокопроизводительных участков.

Следующая подпрограмма, env_parse, также используется в процессе запуска системы. Монитор загрузки передает ядру отдельные параметры через строки подобного вида: «DPETH0=300:10». Функция пытается найти строку, начало которой совпадает с первым переданным ей аргументом, и извлекает значение параметра. Использование этой функции поясняют комментарии в коде. Прежде всего она предназначена для помощи тем, кто хочет разработать новый драйвер, которому могут потребоваться параметры. Показанный пример с параметром DPETH характерен для передачи параметров адаптеру Ethernet, если в ядре включена поддержка сети.

Последние две подпрограммы, bad_assertion и bad_compare, компилируются только в том случае, если макрос DEBUG определен как TRUE. Эти функции поддерживают макросы в assert.h. Хотя на них нет ни одной ссылки в тексте книги, они могут быть полезны при отладке, если читатель захочет создать собственную версию MINIX.

Резюме

Чтобы скрыть эффект прерываний, операционная система предоставляет концептуальную модель, в которой параллельно выполняются логически упорядоченные процессы. Процессы могут взаимодействовать друг с другом при помощи примитивов межпроцессного взаимодействия, таких как семафоры, мониторы и сообщения. Назначение примитивов в том, чтобы гарантировать, что никакие два процесса не окажутся в критической секции единовременно. Процессы могут находиться в состоянии выполнения, готовности и в приостановленном состоянии и могут переходить из одного состояния в другое, когда тот или иной процесс исполняет один из примитивов взаимодействия между процессами.

Примитивы необходимы для решения таких задач, как потребитель-поставщик, задача «обедающих философов», задача одновременного чтения и записи и «спящего брадобрея». Но даже при использовании примитивов необходимо быть осторожным, во избежание ошибок и взаимных блокировок. Известно много различных алгоритмов планирования, таких как круговой (round-robin), планирование с приоритетами, планирование с многоуровневыми очередями и управление политиками планирования.

ОС MINIX поддерживает концепцию процесса и предоставляет примитивы для реализации взаимодействия между процессами. Сообщения не буферизуются, поэтому вызов send завершается успехом только после того, как адресат получит сообщение. Аналогично, вызов receive завершается лишь тогда, когда сообщение уже отправлено. В противном случае сделавший вызов процесс переходит в состояние ожидания.

Когда возникает прерывание, на нижнем уровне ядра создается и отправляется сообщение для задачи, ассоциированной с устройством, вызвавшим прерывание. Пусть, например, задача диска делает вызов receive и, передав контроллеру диска команду на чтение блока данных, блокируется. Когда контролер, наконец, считывает запрошенные данные, он вызывает аппаратное прерывание. Код нижнего уровня системы обрабатывает это прерывание и посылает сообщение задаче диска, переводя ее в состояние готовности. Кроме того, обработчик прерывания может напрямую выполнять некоторые действия, например, обработчик прерывания таймера вправе обновлять системное время.

За прерыванием может следовать переключение задачи. Когда возникает прерывание, создается новый стек внутри ячейки таблицы процессов, принадлежащей текущему процессу. В этот стек помещается вся информация, необходимая для восстановления его работы. Для того чтобы вновь запустить любой приостановленный процесс, необходимо установить указатель стека на кадр стека в таблице процессов и запустить последовательность инструкций, восстанавливающих регистры процессора. В завершение нужно выполнить возврат по инструкции iretd. Какой из процессов запускать, решает планировщик.

Прерывания могут возникать и тогда, когда выполняется код ядра. Эта ситуация обнаруживается процессором, и вместо стека в таблице процессов используется стек самого ядра. Такое вложенное прерывание будет обрабатываться после

того, как завершится текущий обработчик. Когда все произошедшие вложенные прерывания будут обслужены, управление передается пользовательским процессам.

Алгоритм планирования в MINIX использует три очереди с разными приоритетами: для задач (высший приоритет), для файловой системы, менеджера памяти и прочих серверов (средний приоритет) и для пользовательских процессов (низший уровень приоритета). Пользовательские процессы планируются по круговому алгоритму с квантованием времени. Остальные процессы работают до тех пор, пока сами не перейдут в состояние блокировки.

Вопросы

1. Представьте, что вы разрабатываете архитектуру компьютера высокого уровня, который переключает процессы аппаратно, а не с помощью прерываний. Какая информация потребуется процессору? Опишите возможную реализацию переключения процессов на базе аппаратного обеспечения.
2. На всех существующих компьютерах как минимум часть обработчиков прерываний написана на ассемблере. Почему?
3. В тексте утверждалось, что модель, представленная на рис. 2.4, *а*, не подходит для файлового сервера с кэшем в памяти. Почему? Может ли каждый процесс иметь собственный кэш?
4. В случае разветвления многопоточного процесса возникнет проблема, если дочернему процессу достанутся копии всех нитей. Пусть одна из исходных нитей ожидала ввода с клавиатуры. После ветвления обе нити будут ожидать ввода с клавиатуры, по одному в каждом процессе. Может ли такая проблема возникнуть в случае ветвления однопоточного процесса?
5. Что такое состояние состязания?
6. Напишите сценарий оболочки, которая создает файл, содержащий последовательные числа, путем считывания последнего числа, прибавления к нему единицы и записывания результата в конец файла. Запустите одну копию сценария в качестве фонового процесса и одну — в качестве приоритетного процесса. Сколько времени пройдет, прежде чем образуется состояние соревнования? Что в данной модели является критической областью? Измените сценарий, чтобы избежать состояния состязания. Подсказка: воспользуйтесь следующим выражением, чтобы заблокировать файл данных

   ```
   In file file.lock
   ```

7. Является ли выражение

   ```
   In file file.lock
   ```

   эффективным механизмом разделения доступа для таких пользовательских программ, как сценарии в предыдущей задаче? Почему?

8. Будет ли работать решение активного ожидания с использованием переменной turn (рис. 2.6) в случае двух процессоров, совместно использующих общую память?
9. Рассмотрим компьютер, в котором нет инструкции tsl, но есть инструкция для обмена содержимого регистра и слова памяти за одну неделимую операцию. Можно ли применить эту инструкцию для написания программы enter_region, аналогичной листингу 2.3?
10. Опишите коротко, как реализовать семафоры в операционной системе, умеющей блокировать прерывания.
11. Покажите, как можно реализовать «считающие» семафоры (то есть способные хранить произвольные значения) при помощи только бинарных семафоров и обычных машинных команд.
12. В разделе «Примитивы межпроцессного взаимодействия» была описана ситуация с высокоприоритетным процессом H и низкоприоритетным процессом L, которая приводила к вечному зацикливанию процесса H. Может ли возникнуть подобная проблема, если вместо планирования по приоритетам использовать карусельное? Поясните.
13. Синхронизация в мониторах происходит с использованием переменных состояния и двух специальных операций, wait и signal. Более общая форма синхронизации предполагает один примитив waituntil с произвольным булевым предикатом в качестве параметра. Например,

    ```
    waituntil x<0 or y+z<n
    ```

    В данном случае примитив signal больше не нужен. Эта схема существенно более общая, чем схемы Хоара и Хансена, но тем не менее она не используется. Почему? Подсказка: подумайте о реализации.
14. В ресторане быстрого питания есть четыре категории обслуживающего персонала: 1) работники, принимающие заказы; 2) повара, готовящие пищу; 3) специалисты по упаковке блюд и 4) кассиры, принимающие у клиентов деньги и выдающие еду. Каждому из видов персонала можно сопоставить последовательный процесс взаимодействия. Какой формой межпроцессного взаимодействия они пользуются? Свяжите эту модель с процессами в UNIX.
15. Рассмотрим систему передачи сообщений через почтовые ящики. При попытке послать сообщение в полный ящик или получить сообщение из пустого ящика процесс не блокируется, а получает код ошибки. Затем процесс повторяет попытку, пока она не окажется успешной. Приведет ли подобная схема к состоянию состязания?
16. Почему в процедуре take_forks решения задачи обедающих философов (см. листинг 2.9) переменной состояния присваивается значение HUNGRY?
17. Рассмотрим процедуру put_forks в листинге 2.9. Пусть переменной состояния присваивается значение THINKING после двух вызовов процедуры test, а не до. Как это повлияет на решение?

18. Проблему читателей и писателей можно формулировать по-разному, в зависимости от того, какие процессы и в какое время могут быть запущены. Тщательно опишите три варианта задачи, в каждом из которых предоставляется (или не предоставляется) преимущество одной из категорий. В каждом варианте укажите, что происходит, когда читающий или пишущий процесс готов обратиться к базе данных, и что происходит, когда процесс заканчивает работу с базой.
19. Компьютеры CDC 6600 в состоянии обрабатывать до 10 процессов ввода/вывода одновременно благодаря интересной форме циклического планирования, называемой *разделением процессора*. Переключение между процессами происходит после каждой команды, поэтому команда 1 поступает от первого процесса, команда 2 — от второго и т. д. Переключение процессов производится аппаратными средствами, и издержки равны нулю. Если в отсутствие других процессов процессу для выполнения работы нужно T секунд, сколько ему потребуется времени в случае n процессов?
20. Обычно планировщики с циклическим алгоритмом поддерживают список процессов, готовых к работе, причем каждый процесс находится в списке в единственном экземпляре. Что произойдет, если процесс окажется в списке дважды? Существует ли причина, по которой подобное изменение будет полезным?
21. Измерения показали, что время выполнения среднестатистического процесса до блокировки ввода/вывода равно T. На переключение между процессами уходит время S, которое теряется впустую. Напишите формулу расчета эффективности для циклического планирования с квантом Q, принимающим следующие значения:
    1) $Q = \infty$,
    2) $Q > T$,
    3) $S < Q < T$,
    4) $Q = S$,
    5) Q около 0.
22. Запуска ожидают пять задач. Предполагаемое время выполнения задач составляет 9, 6, 3, 5 и X. В каком порядке их следует запустить, чтобы минимизировать среднее время отклика? (Ответ должен зависеть от X.)
23. Пять пакетных задач, A, B, C, D, E, поступают в компьютерный центр практически одновременно. Ожидается, что время их выполнения составит 10, 6, 2, 4 и 8 мин. Их установленные приоритеты равны 3, 5, 2, 1 и 4, причем 5 — высший приоритет. Определите среднее оборотное время для каждого из следующих алгоритмов планирования, пренебрегая потерями при переключении между процессами:
    - циклическое планирование;
    - планирование согласно приоритетам;

- «первым пришел — первым обслужен» (в порядке 10, 6, 2, 4, 8);
- «кратчайшая задача — первая».

В случае 1 предполагается, что система многозадачная и каждой задаче достается справедливая доля процессорного времени. В случаях 2–4 считается, что в каждый момент времени запущена одна задача, работающая вплоть до завершения. Все задачи ограничены исключительно возможностями процессора.

24. Процессу, запущенному в системе CTSS, для завершения необходимо 30 квантов. Сколько раз он будет перегружен на диск, учитывая самый первый раз (прежде, чем он был запущен)?
25. Для предсказания времени выполнения используется алгоритм старения с $a = 1/2$. Предыдущие четыре значения времени составляли 40, 20, 40 и 15 мс (первое значение — самое давнее). Оцените следующее время выполнения.
26. В гибкую систему реального времени поступают четыре периодических сигнала с периодами 50, 100, 200 и 250 мс. На обработку каждого сигнала требуется 35, 20, 10 и x мс времени процессора. Укажите максимальное значение x, при котором система остается поддающейся планированию.
27. Объясните причины широкого распространения двухуровневого планирования.
28. В процессе работы MINIX использует переменную proc_ptr, содержащую указатель на ячейку текущего процесса в таблице процессов. Зачем?
29. В MINIX сообщения не буферизуются. Поясните, почему это приводит к проблемам с прерываниями таймера и клавиатуры.
30. Когда в MINIX приостановленному процессу отправляется сообщение, вызывается подпрограмма ready, которая помещает процесс в одну из очередей планировщика. Эта подпрограмма начинается с блокирования прерываний. Почему?
31. В MINIX функция mini_rec содержит цикл. Зачем?
32. Схема диспетчеризации в MINIX в целом соответствует показанной на рис. 2.10, с различными приоритетами для разных классов. Нижний класс (пользовательские процессы) планируется по циклической схеме, но задачи и серверы всегда выполняются, пока не попадут в блокировку. Возможно ли, что процессы нижнего уровня будут зависать? Почему (или почему нет)?
33. Походит ли MINIX для решения задач реального времени (таких как сбор данных)? Если нет, то что можно сделать, чтобы исправить это?
34. Предположим, у вас имеется операционная система, предоставляющая семафоры. Реализуйте систему обмена сообщениями. Напишите подпрограммы для отправки и приема сообщений.

35. Студент, изучающий антропологию и посещающий занятия по информатике, занялся исследовательским проектом, цель которого — выяснить, можно ли научить африканских бабуинов тому, что такое взаимная блокировка. Он нашел глубокий каньон и протянул через него веревку, чтобы бабуины могли перебраться на другую сторону на руках. За один раз могут перебраться несколько бабуинов, если все они движутся в одном направлении. Если на веревке одновременно оказываются бабуины, двигающиеся на восток и на запад, возникнет взаимная блокировка (бабуины зависнут посреди веревки), так как один бабуин не может перебраться через другого. Поэтому если бабуин хочет перебраться, он сначала должен проверить, что на веревке нет тех, кто движется в обратном направлении. Напишите программу с использованием семафоров, которая не допускала бы блокировки. Не беспокойтесь о том, что серия бабуинов, движущихся на восток, может бесконечно удерживать тех, кто хочет двигаться на запад.
36. Решите предыдущую задачу, но теперь постарайтесь избежать зависаний. Для этого, когда бабуин хочет перебраться на восток, и при этом на веревке уже висят несколько обезьян, движущихся на запад, бабуин ждет, когда веревка освободится. Другое условие: нужно запретить для остальных движение на запад, пока он не перейдет на восток.
37. Решите задачу обедающих философов в стиле «каждому философу по монитору».
38. Добавьте в ядро MINIX код, который бы подсчитывал, сколько раз процесс (или задача) *i* обратился к процессу (или задаче) *j*. Сделайте так, чтобы эта матрица выводилась по нажатию клавиши F4.
39. Измените планировщик MINIX так, чтобы он отслеживал, сколько времени каждый процесс занимал процессор последний раз. Когда все задачи и серверы освободят процессор для пользовательских процессов, выбирайте тот из них, который меньше всех тревожил процессор.
40. Измените MINIX так, чтобы у каждого процесса был приоритет, который хранится в одном из полей таблицы процессов, с целью давать пользовательским процессам больший или меньший приоритет.
41. Перепишите макросы hwint_master и hwint_slave так, чтобы действия, выполняемые функцией save, были бы встроены в код. Насколько увеличился объем кода? Можете ли вы измерить прирост производительности?

Глава 3
Ввод/вывод

Одна из важнейших функций операционной системы состоит в управлении всеми устройствами ввода/вывода компьютера. Операционная система должна давать этим устройствам команды, перехватывать прерывания и обрабатывать ошибки. Она должна также обеспечить простой и удобный интерфейс между устройствами и остальной частью системы. Интерфейс, насколько это возможно, должен быть одинаковым для всех устройств (для достижения независимости от применяемого оборудования). Программное обеспечение ввода/вывода составляет существенную часть операционной системы. Тому, как операционная система управляет устройствами ввода/вывода, и посвящена эта глава.

Глава организована следующим образом. Сначала мы рассмотрим некоторые основы аппаратуры ввода/вывода, затем в общих чертах познакомимся с соответствующим программным обеспечением. Программное обеспечение ввода/вывода может быть структурировано в виде уровней, каждому из которых отведен строго очерченный круг задач. Мы побываем на всех уровнях, чтобы понять, что они делают и как согласуются друг с другом.

Далее следует раздел, посвященный взаимной блокировке. Мы дадим строгое определение этому понятию, покажем, как взаимоблокировки возникают, представим две модели для их анализа и обсудим некоторые алгоритмы их предотвращения.

Затем мы сделаем общий обзор ввода/вывода в MINIX. После этого вступления мы подробно ознакомимся с четырьмя конкретными устройствами ввода/вывода: RAM-диском, жестким диском, часами и терминалом. Для каждого устройства мы рассмотрим его аппаратную часть, программное обеспечение и его реализацию в MINIX. Наконец, завершит главу краткое описание некоторых частей MINIX, расположенных на том же уровне, что и задачи ввода/вывода, но задачами не являющихся. Они предоставляют некоторые дополнительные службы для менеджера памяти и файловой системы, например выполняют передачу блоков данных от пользовательских процессов.

3.1. Принципы аппаратуры ввода/вывода

Разные специалисты рассматривают аппаратуру ввода/вывода с различных точек зрения. Инженеры-электронщики видят в ней микросхемы, проводники, источники питания, двигатели и прочие физические компоненты. Программисты, в первую очередь, обращают внимание на интерфейс, предоставляемый программному обеспечению, — команды, принимаемые аппаратурой, выполняемые ею функции, ошибки, о которых аппаратура может сообщить. В этой книге нас интересует именно программирование устройств ввода/вывода, а не их проекти-

рование, построение или поддержка. Поэтому сфера наших интересов будет ограничена тем, как программировать аппаратуру, а физические принципы работы аппаратуры останутся вне изложения. В то же время программирование многих устройств ввода/вывода часто оказывается тесно связанным с их внутренним функционированием. В следующих трех разделах будут кратко затронуты общие основы знаний из области аппаратуры ввода/вывода, касающиеся программирования. Этот материал можно рассматривать в качестве обзорного и как продолжение темы главы 1.

3.1.1. Устройства ввода/вывода

Устройства ввода/вывода можно грубо разделить на две категории: *блочные* и *символьные*. Блочными называются устройства, хранящие информацию в виде адресуемых блоков фиксированного размера. Обычно размеры блоков варьируются от 512 байт до 32 768 байт. Важное свойство блочного устройства состоит в том, что каждый блок может быть прочитан независимо от остальных блоков. Наиболее распространенными блочными устройствами являются диски.

Если приглядеться внимательнее, то окажется, что граница между блочно адресуемыми устройствами и устройствами, к отдельным составляющим которых нельзя адресоваться напрямую, не определена строго. Все согласны с тем, что диск является блочно адресуемым устройством, так как вне зависимости от текущего положения головки дисковода всегда можно переместить ее на определенный цилиндр и затем считать или записать отдельный блок с нужной дорожки. Рассмотрим теперь накопитель на магнитной ленте (магнитофон), применяемый для хранения резервных копий диска. На ленте хранится последовательность блоков. Если магнитофону дать команду прочитать некоторый блок, ему потребуется перемотать ленту и начать читать данные, пока процесс не дойдет до запрашиваемого блока. Эта операция подобна поиску блока на диске с той лишь разницей, что она занимает значительно больше времени. Кроме того, в зависимости от накопителя и формата хранящихся на нем данных не гарантирована запись отдельного произвольного блока в середине ленты. Попытка использовать магнитные ленты в качестве блочных устройств произвольного доступа явилась бы в какой-то степени натяжкой: никто их не использует таким образом.

Другой тип устройств ввода/вывода — символьные устройства. Символьное устройство принимает или предоставляет поток символов без какой-либо блочной структуры. Оно не является адресуемым и не выполняет операцию поиска. Принтеры, сетевые интерфейсные адаптеры, мыши (для указания точки на экране), крысы (для лабораторных экспериментов по психологии) и большинство других устройств, не похожих на диски, можно рассматривать как символьные устройства.

Такая схема классификации не совершенна. Некоторые устройства просто не попадают ни в одну из категорий. Например, часы не являются блочно адресуемыми. Они также не формируют и не принимают символьных потоков. Вся их деятельность сводится к инициированию прерываний в строго определенные моменты времени. Видеопамять также не укладывается в рамки этой модели.

И все же классификация на блочные и символьные устройства является настолько общей, что неплохо подходит в качестве основы для достижения независимости от устройств некоторого программного обеспечения операционных систем, имеющего дело с вводом/выводом. Так, файловая система общается с абстрактными блочными устройствами, а зависимую от устройств часть оставляет программному обеспечению низкого уровня, *драйверам устройств.*

3.1.2. Контроллеры устройств

Устройства ввода/вывода, как правило, состоят из механической части и электронной части. В большинстве случаев эти части можно особо выделить для придания модели более модульного и общего характера. Электронный компонент называется *контроллером устройства.* В персональных компьютерах он обычно принимает форму печатной платы, вставляемой в слот расширения объединительной платы (ранее некорректно называемой материнской). Механический компонент находится в самом устройстве.

Плата контроллера обычно снабжается разъемом, к которому может быть подключен кабель, ведущий к самому устройству. Многие контроллеры способны управлять двумя, четырьмя или даже восемью идентичными устройствами. Если интерфейс между контроллером и устройством является стандартным, то есть официальным стандартом ANSI, IEEE или ISO, либо фактическим стандартом, это смягчает ограничения выпуска по отдельности контроллеров и устройств, удовлетворяющих данному интерфейсу. Так, многие компании производят жесткие диски, соответствующие интерфейсу IDE или SCSI.

Мы упоминаем о различии между контроллером и устройством потому, что операционная система практически всегда имеет дело с контроллером, а не с самим устройством. У большинства небольших компьютеров взаимодействие с устройствами организуется по модели единой шины, показанной на рис. 3.1. У больших машин, мэйнфреймов, применяется другая модель с несколькими шинами, которыми заведуют специализированные компьютеры ввода/вывода, называемые *каналами ввода/вывода.* Такая организация позволяет уменьшить нагрузку на основной процессор.

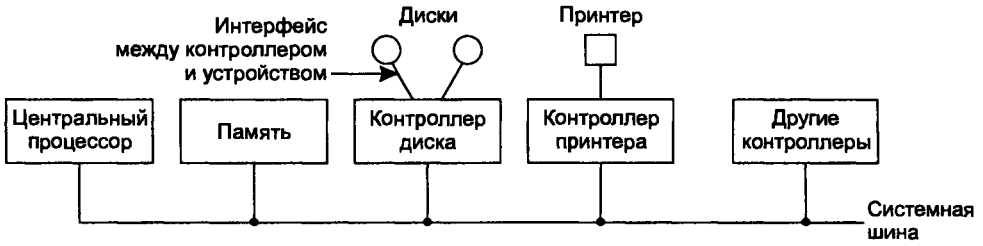


Рис. 3.1. Соединение процессора, памяти, контроллеров и устройств ввода/вывода

Интерфейс между устройством и контроллером часто является интерфейсом очень низкого уровня. Например, какой-нибудь жесткий диск может быть от-

форматирован по 256 секторов на дорожку, с размером секторов по 512 байт. В действительности с диска в контроллер поступает последовательный поток битов, начинающийся с *заголовка сектора* (преамбулы), за которым следует 4096 бит в секторе и, наконец, *контрольная сумма*, также называемая *кодом исправления ошибок* (ECC, Error-Correcting Code). Заголовок сектора записывается на диск во время форматирования. Он содержит номера цилиндра и сектора, размер сектора, информацию синхронизации и т. п.

Работа контроллера заключается в преобразовании последовательного потока битов в блок байтов и выполнении коррекции ошибок, если это необходимо. Обычно байтовый блок собирается бит за битом в буфере контроллера. Затем проверяется контрольная сумма блока, и если она совпадает с указанной в заголовке сектора, блок объявляется считанным без ошибок, после чего он копируется в оперативную память.

Контроллер монитора (видеоконтроллер) также работает как бит-последовательное устройство, на таком же низком уровне. Он считывает из памяти байты, содержащие символы, которые следует отобразить, и формирует сигналы, используемые для модуляции луча электронной трубки, заставляющие ее выводить изображение на экран. Видеоконтроллер также формирует сигналы, управляющие горизонтальным и вертикальным перемещениями электронного луча. Если бы не контроллер, программисту пришлось бы делать это самому. В действительности же операционная система всего-навсего инициализирует контроллер, задавая небольшое число параметров, таких как количество символов или пикселов в строке и число строк на экране, а тяжелую работу по управлению разверткой берет на себя контроллер.

У каждого контроллера есть несколько регистров, с помощью которых с ним может общаться центральный процессор. У некоторых компьютеров такие регистры являются частью единого адресного пространства системы. Примером такой системы может служить 680x0. Назначение адреса каждому из устройств производится посредством схем логики декодирования шины и контроллеров устройств. У других систем для устройств ввода/вывода отводится специальное адресное пространство, в котором выделяются адреса для каждого из устройств. Таковы, например, IBM PC-совместимые машины.

В дополнение к портам ввода/вывода, часто используются прерывания, при помощи которых контроллер может сообщить процессору, что его регистры готовы для записи или для чтения. Прерывание, прежде всего, является электрическим сигналом. Линия запроса аппаратного прерывания (IRQ, Interrupt ReQuest line) является одним из физических входов чипа контроллера прерываний. Количество этих входов ограничено. Например, у персональных компьютеров Pentium только 15 IRQ доступны для устройств ввода/вывода. Некоторые из контроллеров встроены в материнскую плату, как, например, контроллер клавиатуры на IBM PC. У тех контроллеров, что вставляются в разъем на объединительной плате, установить соответствие между сигналом IRQ и устройством иногда можно при помощи перемычек или переключателей. Это бывает необходимо для исключения конфликтов, поскольку на современных материнских платах, поддерживающих технологию Plug'n'Play, IRQ назначаются программно. Каждая из

линий IRQ связывается с вектором прерываний, который указывает на программу обработки прерывания. В качестве примера в табл. 3.1 приведены адреса ввода/вывода, IRQ и значения векторов прерываний для нескольких контроллеров IBM PC. В MINIX действуют те же самые значения IRQ, но векторы прерываний назначаются другие.

Операционная система обменивается с устройством информацией, записывая команды в регистры контроллера. Контроллер гибкого диска IBM PC воспринимает 15 различных команд, в том числе READ, WRITE, SEEK, FORMAT и CALIBRATE. У многих команд есть параметры, которые также передаются через регистры контроллера. Передав команду контроллеру, процессор может продолжить свою работу. Затем, когда устройство выполнит команду, контроллер инициирует прерывание, чтобы операционная система вновь получила управление и могла бы проверить результаты операции, которые процессор получает, считывая один или несколько байтов из регистров контроллера.

Таблица 3.1. Некоторые контроллеры, их адреса ввода/вывода, IRQ и векторы прерываний для IBM PC, работающей под управлением MS-DOS

Контроллер ввода/вывода	Адрес ввода/вывода	IRQ	Вектор прерывания
Таймер	040–043	0	8
Клавиатура	060–063	1	9
Жесткий диск	1F0–1F7	14	118
Дополнительный RS232	2F8–2FF	3	11
Принтер	378–37F	7	15
Дисковод	3F0–3F7	6	14
Основной RS232	3F8–3FF	4	12

3.1.3. Прямой доступ к памяти (DMA)

Многие контроллеры, особенно контроллеры блок-ориентированных устройств, поддерживают *прямой доступ к памяти* — DMA (Direct Memory Access). Чтобы понять, как работает DMA, познакомимся сначала с тем, как происходит чтение с диска в отсутствие DMA. Сначала контроллер считывает с диска блок (один или несколько секторов) последовательно, бит за битом, пока весь блок не окажется во внутреннем буфере контроллера. Затем контроллер проверяет контрольную сумму, чтобы убедиться, что при чтении не произошло ошибки. После этого контроллер инициирует прерывание. Когда операционная система начинает работу, она может прочитать блок диска побайтно или пословно, в цикле сохраняя считанное слово или байт в оперативной памяти.

Естественно, цикл, байт за байтом считывающий данные с контроллера, расходует процессорное время. Чтобы освободить процессор от низкоуровневой работы, и был изобретен прямой доступ к памяти. При использовании DMA процедура совершенно другая. Сначала центральный процессор программирует DMA-контроллер, устанавливая его регистры и указывая таким образом, какие данные и куда следует переместить (рис. 3.2). Затем процессор дает команду

дисковому контроллеру прочитать данные во внутренний буфер и проверить контрольную сумму. Когда данные получены и проверены контроллером диска, устройство DMA может начинать работу.

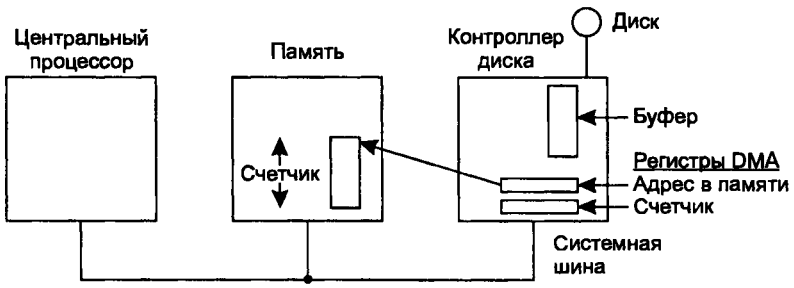


Рис. 3.2. Работа DMA-контроллера

DMA-контроллер начинает перенос данных, посылая дисковому контроллеру по шине запрос на чтение. Этот запрос выглядит как обычный запрос чтения, потому контроллер диска даже не знает, поступил ли он от центрального процессора или от контроллера DMA. Обычно адрес памяти уже находится на адресной шине, соответственно, контроллер диска всегда в курсе, куда нужно переслать следующее слово из своего внутреннего буфера. Запись в память является еще одним стандартным циклом шины. Когда запись закончена, контроллер диска также по шине посылает сигнал подтверждения контроллеру DMA. Затем контроллер DMA увеличивает используемый адрес памяти и уменьшает значение счетчика байтов. После этого шаги со 2-го по 4-й повторяются, пока значение счетчика не станет равно нулю. По завершении цикла копирования контроллер DMA инициирует прерывание процессора, сообщая ему таким образом, что перенос данных завершен. Операционной системе не нужно копировать блок диска в память. Он уже находится там.

Как мы уже упоминали, до начала операции прямого доступа к памяти диск предварительно считывает данные в свой внутренний буфер. Возможно, вы задаетесь вопросом, почему контроллер не помещает данные прямо в оперативную память, по мере получения их с диска. Другими словами, зачем ему нужен внутренний буфер? Тому две причины. Во-первых, при помощи внутренней буферизации контроллер диска может проверить контрольную сумму до начала переноса данных в память. Если значения не совпадают, формируется сигнал об ошибке и перенос данных не производится.

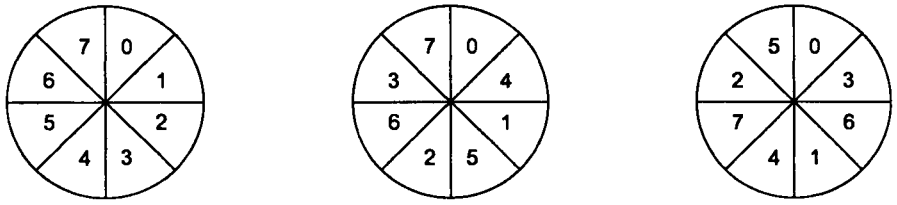
Во-вторых, дело в том, что как только началась операция чтения с диска, биты начинают поступать с постоянной скоростью, независимо от того, готов контроллер диска их принимать или нет. Если контроллер диска попытается писать эти данные напрямую в память, ему придется делать это по системной шине. Если при передаче очередного слова шина окажется занятой каким-либо другим устройством (например, использующим ее в пакетном режиме), контроллеру диска придется ждать. Если следующее слово с диска прибудет раньше, чем кон-

троллер успеет сохранить отложенное, контроллер либо потеряет предыдущее слово, либо ему придется запоминать его где-либо еще. Таким образом, необходимость внутренней буферизации становится очевидной. При наличии внутреннего буфера контроллеру диска шина не нужна до тех пор, пока не начнется операция DMA. В результате устройство контроллера диска оказывается проще, так как при операции прямого доступа к памяти параметр времени не является критичным. (Некоторые древние контроллеры действительно напрямую обращались к памяти, обладая внутренним буфером небольшого размера, что часто приводило к ошибкам перегрузки при занятости шины.)

Описанная выше двухэтапная процедура буферизации имеет важные последствия, касающиеся производительности ввода/вывода. Когда данные передаются от контроллера в память самим контроллером или центральным процессором, под дисковой головкой проходит следующий сектор данных, биты которого поступают в контроллер. Простые контроллеры не умели одновременно считывать и передавать данные, и сектор пропускался.

Как следствие, может быть прочитан любой другой сектор, кроме следующего, а для чтения всей дорожки необходимы два полных оборота диска. Если же время, требуемое для передачи сектора данных в память, превышает время считывания сектора, может потребоваться пропуск двух (или более) блоков.

Пропуск блоков с целью дать контроллеру время для передачи данных в память называется *чередованием*. При этом во время форматирования секторы нумеруются с учетом фактора чередования. На рис. 3.3, *а* показан диск с восемью секторами без чередования. На рис. 3.3, *б* вы видите тот же диск с фактором чередования 1. На рис. 3.3, *в* фактор чередования равен 2.



а *б* *в*

Рис. 3.3. *а* — диск без чередования; *б* — фактор чередования 1; *в* — фактор чередования 2

Идея подобной нумерации блоков в том, чтобы позволить системе читать блоки с последовательными номерами и чтобы аппаратное обеспечение работало с максимальной производительностью. Если бы диск был без чередования, а контроллер не справлялся с чтением последовательных блоков, то для полного последовательного считывания всей дорожки потребовалось бы 8 оборотов диска. (Конечно, проблема решаема и программно, но лучше, чтобы об этом заботился контроллер.)

DMA используется не во всех компьютерах. Главный аргумент против прямого доступа к памяти: центральный процессор обычно значительно превосходит DMA-контроллер по скорости и в состоянии выполнить ту же работу

значительно быстрее (если только скорость ограничена не быстродействием устройства ввода/вывода). При отсутствии другой нагрузки у центрального процессора заставлять быстрый центральный процессор ждать, пока медленный контроллер DMA выполнит свою работу, бессмысленно. Кроме того, компьютер без контроллера DMA, с центральным процессором, выполняющим все программно, оказывается дешевле, что крайне важно в производстве компьютеров нижней ценовой категории.

3.2. Принципы программного обеспечения ввода/вывода

Перейдем теперь от рассмотрения аппаратуры ввода/вывода к знакомству с программным обеспечением ввода/вывода. Сначала мы проникнемся целями программного обеспечения ввода/вывода, а затем поглядим на различные способы выполнения операций ввода/вывода с точки зрения операционной системы.

3.2.1. Задачи программного обеспечения ввода/вывода

Ключевая концепция разработки программного обеспечения ввода/вывода известна как *независимость от устройств*. Эта концепция означает возможность написания программ, способных получать доступ к любому устройству ввода/вывода без предварительного указания конкретного устройства. Соответственно, программа, читающая данные из входного файла, должна с одинаковым успехом работать с файлом на дискете, жестком диске или компакт-диске. Причем без каких-либо изменений в программе. Например, должна быть возможность дать команду вроде

```
sort <input >output
```

и эта команда должна работать, невзирая на то, что указано в качестве входного устройства — гибкий диск, IDE-диск, SCSI-диск или клавиатура. В качестве выходного устройства также с равным успехом может быть указан экран, файл на любом диске или принтер. Все проблемы, связанные с отличиями этих устройств, должна решать операционная система.

Тесно связан с идеей независимости от устройств принцип *единообразного именования*. Имя файла или устройства должно быть просто текстовой строкой или целым числом и никоим образом не зависеть от физического устройства. В системе UNIX все диски могут быть произвольным образом интегрированы в иерархию файловой системы, поэтому пользователю не обязательно знать, какое имя какому устройству соответствует. Например, гибкий диск не запрещается *смонтировать* поверх каталога /usr/ast/backup, вследствие чего копирование файла в каталог /usr/ast/backup/monday автоматически приведет к копированию файлов на гибкий диск. Таким образом, все файлы и устройства адресуются одним и тем же способом: по пути к ним.

Другим важным аспектом программного обеспечения ввода/вывода является *обработка ошибок*. Ошибки должны обрабатываться как можно ближе к аппаратуре. Если контроллер обнаружил ошибку чтения, он должен попытаться по возможности исправить эту ошибку сам. Если он не в силах это сделать, тогда ошибку обязан обработать драйвер устройства, допустим, попытавшись прочитать этот блок еще раз. Многие ошибки бывают временными, как, например, ошибки чтения, вызванные пылинками на читающих головках. Такие ошибки часто не воспроизводятся при повторной попытке чтения блока. Только если нижний уровень пасует перед проблемой, о ней следует информировать верхний уровень. Во многих случаях восстановление после ошибок предпочтительно делать на нижнем уровне, прозрачно для верхних уровней, то есть так, что вышестоящие уровни даже не будут подозревать о наличии сбоев.

Еще один ключевой вопрос — способ переноса данных: *синхронный* (блокирующий) против *асинхронного* (управляемого прерываниями). Большинство операций ввода/вывода на физическом уровне являются асинхронными — центральный процессор запускает передачу данных и забывает о ней, пока не сгенерируется прерывание. Пользовательские программы значительно легче написать, используя блокирующие операции ввода/вывода, — после обращения к системному вызову read программа автоматически приостанавливается до тех пор, пока данные не появятся в буфере. Тем, чтобы операции ввода/вывода, в действительности являющиеся асинхронными, выглядели как блокирующие в программах пользователя, занимается операционная система.

Говоря о программном обеспечении ввода/вывода, нельзя обойти вниманием *буферизацию*. Часто данные, поступающие с устройства, не могут быть сохранены сразу там, куда они в конечном итоге направляются. Например, когда пакет приходит по сети, операционная система не знает, куда его поместить, пока не изучит его содержимое, для чего этот пакет нужно где-то временно пристроить. Кроме того, для многих устройств реального времени крайне важными оказываются параметры сроков поступления данных (например, для устройств воспроизведения цифрового звука), поэтому полученные данные должны быть помещены в выходной буфер заранее, чтобы скорость, с которой они извлекаются из буфера проигрывателем, не зависела от скорости заполнения буфера. Таким образом удается избежать неравномерности воспроизведения звука. Буферизация подразумевает копирование данных в значительных количествах, что часто является основным фактором снижения производительности операций ввода/вывода.

И последнее — это понятие *выделенных* устройств и устройств *коллективного использования*. С некоторыми устройствами ввода/вывода, такими как диски, может одновременно работать большое количество пользователей. При этом не должно возникать проблем, когда несколько пользователей одновременно откроют файлы на одном и том же диске. Другие устройства, такие как накопители на магнитной ленте, должны предоставляться в монопольное владение одному пользователю, пока он не завершит свою работу с этим устройством. Если два или более пользователей одновременно станут писать вперемешку блоки на одну ленту, ничего хорошего не получится. Введение понятия выделенных (монопольно используемых) устройств также привносит целый спектр проблем, таких

как взаимоблокировки. Тем не менее операционная система обязана управлять как устройствами общего доступа, так и выделенными устройствами и преодолевать различные потенциальные проблемы самостоятельно.

Эти задачи решаются путем разбиения программного обеспечения ввода/вывода на четыре уровня.

1. Обработчики прерываний (нижний уровень).
2. Драйверы устройств.
3. Независимый от аппаратуры код операционной системы.
4. Пользовательские программы (верхний уровень).

3.2.2. Обработчики прерываний

Хотя программный ввод/вывод иногда бывает полезен, для большинства операций ввода/вывода прерывания являются неприятным, но необходимым фактом. Прерывания должны быть упрятаны как можно глубже во внутренностях операционной системы, чтобы о них знала как можно меньшая ее часть. Лучший способ завуалировать их заключается в блокировке драйвера, начавшего операцию ввода/вывода, вплоть до окончания этой операции и получения прерывания. Драйвер может заблокировать себя сам, выполнив на семафоре процедуру down, процедуру wait на переменной состояния, процедуру receive на сообщении или что-либо подобное.

Когда происходит прерывание, начинает работу обработчик прерываний. По ее окончании он может разблокировать драйвер-инициатор. В некоторых случаях это реализуется через процедуру up на семафоре. В других ситуациях обработчик прерываний вызывает процедуру монитора signal с переменной состояния. Или же он посылает заблокированному драйверу сообщение. В любом случае драйвер разблокируется обработчиком прерываний. Эта схема лучше всего работает в драйверах, являющихся процессами ядра со своим собственным состоянием, стеком и счетчиком команд.

3.2.3. Драйверы устройств

Весь код, зависимый от конкретного устройства, находится в драйверах. Каждый драйвер обслуживает один тип устройств или, как максимум, целый класс сходных устройств. Например, было бы хорошей идеей иметь один драйвер терминала, несмотря на то что система поддерживает несколько различающиеся типы терминалов. С другой стороны, примитивный механический терминал, распечатывающий текст, и интеллектуальный графический терминал с мышью разнятся настолько, что для них предпочтительны персональные драйверы.

Ранее в этой главе мы познакомились с функциями контроллеров устройств ввода/вывода. Как было сказано, у каждого контроллера есть набор регистров, используемых для того, чтобы давать опекаемому устройству команды и читать состояние устройства. Число таких регистров и выдаваемые команды зависят от конкретного устройства. Например, драйвер диска должен знать о секторах, до-

рожках, цилиндрах, головках, их перемещении и времени установки, двигателях и тому подобных вещах, что необходимо для правильной работы диска.

Вообще говоря, назначение драйвера в том, чтобы воспринимать абстрактные запросы от аппаратно-независимых программ верхнего уровня и сообщать им, что запрос выполнен. При этом, если в момент передачи запроса драйвер бездействовал, он сразу начинает работу. Если же драйвер был занят, запрос обычно помещается в очередь и обслуживается по мере возможности.

Поэтому для управления каждым устройством ввода/вывода, подключенным к компьютеру, требуется специальная программа. Эта программа, называемая *драйвером устройства*, часто пишется производителем устройства и распространяется в той же коробке. Поскольку для каждой операционной системы требуются специализированные драйверы, производители обычно поставляют драйверы для нескольких наиболее популярных операционных систем.

При обработке запроса на обмен данными драйвер прежде всего преобразует запрос из абстрактного представления в конкретную форму. Скажем, драйвер диска должен выяснить, где находится запрошенный блок данных, проверить, работает ли привод диска, находится ли головка над нужной дорожкой и т. д. Говоря коротко, драйвер должен сам определить свою последовательность действий.

После того как необходимые команды определены, драйвер начинает передавать их устройству через регистры контроллера. Причем с оглядкой на то, что некоторые контроллеры способны воспринимать только по одной команде за раз, а другие — цепочку команд, выполняемых далее без вмешательства операционной системы.

Когда все команды переданы, ситуация развивается по двум сценариям. Во многих случаях драйвер устройства должен ждать, пока контроллер не выполнит для него определенную работу, поэтому он блокируется до поступления прерывания от устройства. В других вариантах операция завершается без задержек, и драйверу не нужно блокироваться. Например, для прокрутки экрана в символьном режиме требуется записать лишь несколько байтов в регистры контроллера. Вся операция занимает несколько наносекунд.

Таким образом, заблокированный драйвер будет либо активизирован прерыванием, либо он вовсе не блокируется. Как бы то ни было, по завершении операции драйвер обязан убедиться, что операция прошла без ошибок. Если все в порядке, драйверу, возможно, придется передать данные (например, только что прочитанный блок) независимому от устройств программному обеспечению. Наконец, драйвер возвращает некоторую информацию вызывающей программе для уведомления той о статусе завершения операции. Если в очереди находились другие запросы, один из них теперь может быть выбран и запущен, иначе драйвер блокируется в ожидании следующего запроса.

3.2.4. Независимое от устройств программное обеспечение ввода/вывода

Хотя некоторая часть программного обеспечения ввода/вывода предназначена для работы с конкретными устройствами, другая часть является независимой от

устройств. Расположение точной границы между драйверами и независимым от устройств программным обеспечением обусловлено системой (и устройствами), так как некоторые функции, которые могут быть реализованы независимым от устройств образом, часто выполняются прямо в драйверах из различных соображений, в том числе с позиций эффективности. Функции, показанные в табл. 3.2, обычно реализуются в независимом от устройств программном обеспечении. В MINIX большая часть таких программ является составляющей файловой системы, которая находится на уровне 3 (см. рис. 2.13). Файловую систему мы будем изучать в главе 5, а здесь дадим только краткий обзор, чтобы продемонстрировать некоторые перспективы и лучше объяснить, как работают драйверы.

Таблица 3.2. Функции независимого от устройств программного обеспечения

Единообразный интерфейс для драйверов устройств
Именование устройств
Защита устройств
Обеспечение аппаратно-независимого размера блока
Буферизация
Сообщение об ошибках
Выделение места на блочных устройствах
Захват и освобождение выделенных устройств
Обработка ошибок

Основная задача независимого от устройств программного обеспечения заключается в выполнении функций ввода/вывода, общих для всех устройств, и предоставлении единообразного интерфейса для программ уровня пользователя.

Одна из основных задач операционной системы состоит в том, чтобы дать имена таким объектам, как файлы и устройства ввода/вывода. Отображением символических имен устройств на соответствующие драйверы занимаются аппаратно-независимые программы. Например, в UNIX имя устройства /dev/disk0 однозначно указывает *i*-узел специального файла, а подходящий драйвер определяется по *главному номеру устройства*. Этот *i*-узел также содержит *младший номер устройства*, передаваемый в виде параметра драйверу для указания конкретного диска или раздела диска, к которому относится операция чтения или записи. Все устройства в системе UNIX имеют главный и второстепенный номера, по которым они однозначно идентифицируются. Выбор всех драйверов осуществляется по главному номеру устройства.

С именованием устройств тесно связан вопрос защиты. Как операционная система предотвращает доступ пользователей к устройствам, на который у них нет прав? В UNIX и в Windows 2000 устройства представляются в файловой системе в виде именованных объектов, что дает возможность применять обычные правила защиты файлов к устройствам ввода/вывода. Таким образом, системному администратору легко установить нужные разрешения для каждого устройства.

У различных дисков могут быть разные размеры сектора. Независимое от устройств программное обеспечение должно скрывать этот факт от верхних уровней и предоставлять им единообразный размер блока, например, объединяя несколько физических сегментов в одну логическую сущность. При этом более высокие уровни имеют дело только с абстрактными устройствами, с одним и тем же размером логического блока, не зависящим от размера физического сектора. Некоторые символьные устройства предоставляют свои данные побайтово (например, модемы), тогда как другие выдают их большими порциями (сетевые интерфейсы). Эти различия также могут быть скрыты.

Буферизация также является важным вопросом как для блочных, так и для символьных устройств по самым разным причинам. Для блочных устройств аппаратное обеспечение обычно требует того, чтобы чтение или запись производились большими блоками. Но для пользовательских программ такого ограничения нет, и они вправе передавать любые объемы информации. Поэтому, если пользователь передал только половину блока, операционная система обычно не станет сразу записывать эти данные на диск, а дождется того, когда будет передана оставшаяся часть блока. Что касается символьных устройств, то пользователь может передавать данные быстрее, чем устройство в состоянии их воспринять, таким образом, также необходима буферизация. Не исключено также, что данные, например, от клавиатуры, могут опережать свое считывание, и в этом случае также не обойтись без буфера.

Когда файл создается и заполняется данными, необходимо выделять для него новые блоки на диске. Для этого операционной системе нужен список или карта свободных блоков на диске. Алгоритм обнаружении свободных блоков является аппаратно-независимым и перемещаем с уровня драйвера на более высокий уровень.

Некоторые устройства, например привод CD-RW, рассчитаны на монопольное владение в каждый момент времени. Операционная система должна рассмотреть запросы на использование такого устройства и либо принять их, либо отказать в выполнении запроса, в зависимости от доступности запрашиваемого устройства. Простой способ обработки этих запросов заключается в соответствующей реализации системного вызова `open` по отношению к специальным файлам. Если устройство недоступно, вызов `open` завершится неуспешно. Обращение к системному вызову `close` освобождает устройство.

Обработка ошибок, по большей части, производится драйвером. Многие ошибки являются специфичными для конкретного устройства и должны обрабатываться соответствующей программой, так как только она знает, что делать (например, повторить попытку, игнорировать ошибку или инициировать сбой системы). Типичная ошибка, когда блок на диске поврежден или не может быть прочитан. Драйвер диска пытается несколько раз повторить чтение и, если оно не удается, информирует вышестоящую программу. С этого момента обработка ошибки является аппаратно-независимой. Если ошибка имела место при чтении пользовательского файла, достаточно просто передать сообщение программе, сделавшей вызов. Если же невозможно причитать критическую системную структуру, не исключено, что системе придется вывести информацию об ошибке и завершить свою работу.

3.2.5. Программное обеспечение ввода/вывода пространства пользователя

Хотя большая часть программного обеспечения ввода/вывода находится в операционной системе, небольшие его порции состоят из библиотек, присоединенных к программам пользователя, или даже целых программ, работающих вне ядра. Системные вызовы, включая системные вызовы ввода/вывода, обычно собраны из библиотечных процедур. Если программа на C содержит вызов

```
count = write(fd, buffer, nbytes);
```

библиотечная процедура write будет скомпонована с программой и, таким образом, будет содержаться в двоичном коде, загружаемом в память во время выполнения программы. Набор всех этих библиотечных процедур, несомненно, является частью системы ввода/вывода.

Хотя многие такие процедуры мало что делают, помимо обращения к системному вызову с соответствующими аргументами, есть ряд процедур ввода/вывода, производящих определенную работу. В частности, библиотечными процедурами выполняются операции форматного ввода и вывода. Например, процедура printf языка C, принимающая на входе текстовую строку и, возможно, несколько переменных, создает из нее ASCII-строку, после чего обращается к системному вызову write для непосредственного вывода. Примером сходной процедуры ввода служит scanf, читающая текстовую строку и преобразующая ее в значения переменных в соответствии с форматом, сходным с используемым процедурой printf. Стандартная библиотека ввода/вывода содержит большое количество процедур, включающих операции ввода/вывода и работающих как часть программы пользователя.

Не все программное обеспечение ввода/вывода пространства пользователя состоит из библиотечных процедур. Другая важная категория — это система спулинга. *Спулинг* (spooling — подкачка, предварительное накопление данных) представляет собой способ работы с выделенными устройствами в многозадачной системе. Рассмотрим типичное устройство, на котором используется подкачка: принтер. В принципе, можно разрешить каждому пользователю открывать специальный символьный файл принтера, однако представьте себе, что процесс открыл его, а затем не обращался к принтеру в течение нескольких часов. Ни один другой процесс в это время не сможет ничего напечатать.

Вместо этого создается специальный процесс, называемый *демоном*, и специальный каталог, называемый *каталогом спулинга* или *каталогом спулера*. Чтобы распечатать файл, процесс сначала создает специальный файл, предназначенный для печати, который помещает в каталог спулинга. Этот файл печатает демон, единственный процесс, которому разрешается пользоваться специальным файлом принтера. Таким образом, потенциальная пробка, связанная с тем, что какой-либо процесс на слишком долгий срок захватит принтер, решается при помощи защиты специального файла принтера от прямого доступа пользователей.

Спулинг используется не только для принтеров. Например, передачу файлов по сети также часто осуществляет специальный сетевой демон. Чтобы послать

куда-либо файл, пользователь помещает его в каталог сетевого демона. Затем сетевой демон извлекает оттуда файл и посылает по сети. Подобный способ принят в системе сетевых новостей USENET. Эта сеть состоит из миллионов машин по всему миру, общающихся друг с другом по Интернету. Существуют тысячи конференций по самым разным темам. Чтобы послать новое сообщение, пользователь вызывает программу новостей, которая принимает сообщение, а затем помещает его в каталог спулинга для последующей отправки на другие машины. Вся система новостей работает вне операционной системы.

На рис. 3.4 показана структура системы ввода/вывода, со всеми уровнями и основными функциями каждого уровня. В порядке снизу вверх эти уровни представляют собой: аппаратуру, обработчики прерываний, независимое от устройств программное обеспечение и, наконец, процессы пользователя.

Рис. 3.4. Уровни и основные функции системы ввода/вывода

Стрелки на рис. 3.4 изображают поток управления. Например, когда программа пользователя пытается прочитать блок из файла, для обработки вызова запускается операционная система. Независимое от устройств программное обеспечение ищет этот блок в кэше. Если требуемого блока там нет, оно вызывает драйвер устройства, чтобы обратиться к аппаратуре и получить этот блок с диска. Процесс же блокируется до завершения дисковой операции.

Когда диск завершает операцию, аппаратура инициирует прерывание. Обработчик прерываний запускается с целью определить, что случилось, то есть какое устройство требует внимания. Затем он извлекает статус устройства и активизирует «спящий» процесс, чтобы завершить запрос ввода/вывода и предоставить пользовательскому процессу возможность продолжения работы.

3.3. Взаимоблокировка

В компьютерных системах существует большое количество ресурсов, каждый из которых в конкретный момент времени может использоваться только одним про-

цессом. В качестве таких примеров можно привести принтеры, накопители на магнитной ленте и элементы внутренних таблиц системы. Появление двух процессов, одновременно передающих данные на принтер, приведет к печати бессмысленного набора символов. Наличие двух процессов, использующих один и тот же элемент таблицы файловой системы, обязательно станет причиной разрушения файловой структуры. Поэтому все операционные системы обладают способностью предоставлять процессу эксклюзивный доступ (по крайней мере, временный) к определенным ресурсам.

Часто прикладной процесс нуждается в исключительном доступе не к одному, а к нескольким ресурсам. Предположим, например, что каждый из двух процессов хочет записать отсканированный документ на компакт-диск. Процесс A запрашивает разрешение на использование сканера и получает его. Процесс B запрограммирован по-другому, поэтому сначала запрашивает устройство для записи компакт-дисков и также получает его. Затем процесс A обращается к устройству для записи компакт-дисков, но запрос отклоняется до тех пор, пока это устройство занято процессом B. К сожалению, вместо того чтобы освободить устройство для записи компакт-дисков, B запрашивает сканер. В этот момент процессы заблокированы и будут вечно оставаться в подвешенном состоянии. Такая ситуация называется *тупиком*, *клинчем* или *взаимоблокировкой*.

Взаимоблокировки вероятны во множестве других ситуаций помимо запросов выделенных устройств ввода/вывода. В системах баз данных программа может оказаться вынужденной заблокировать несколько записей, чтобы избежать состояния конкуренции. Если процесс A блокирует запись R1, процесс B блокирует запись R2, а затем каждый процесс попытается заблокировать чужую запись, мы также окажемся в тупике. Таким образом, взаимоблокировки появляются при работе как с аппаратными, так и с программными ресурсами.

В этой главе мы рассмотрим тупиковые ситуации более подробно, увидим, как они возникают, и изучим некоторые способы, позволяющие предупредить взаимные блокировки или избежать их. Хотя информация о тупиках представлена в контексте операционной системы, они также могут встретиться в системах баз данных и во множестве других ситуаций, поэтому рассматриваемый материал на самом деле применим к широкому ряду систем, работающих с несколькими процессами. О взаимоблокировках было сложено много книг и статей. Несмотря на то что эти библиографии датируются давними цифрами, так как большая часть работ по взаимоблокировкам была проделана до 1980 года, данные книги полезны до сих пор.

3.3.1. Ресурсы

Система может зайти в тупик, когда процессам предоставляются исключительные права доступа к устройствам, файлам и т. д. Чтобы максимально обобщить рассказ о взаимоблокировках, мы будем называть объекты предоставления доступа *ресурсами*. Ресурсом может быть аппаратное устройство (например, накопитель на магнитной ленте) или часть информации (закрытая запись в базе данных). В компьютере существует масса различных ресурсов, к которым могут

происходить обращения. Кроме того, в системе может оказаться несколько идентичных экземпляров какого-либо ресурса, например три накопителя на магнитных лентах. Если в системе есть несколько экземпляров ресурса, то в ответ на обращение к нему может предоставляться любая из доступных копий. Короче говоря, ресурс — это все то, что вправе использоваться только одним процессом в любой момент времени.

Ресурсы бывают двух типов: выгружаемые и невыгружаемые. *Выгружаемый ресурс* позволяется безболезненно забирать у владеющего им процесса. Образцом такого ресурса является память. Рассмотрим, например, систему с пользовательской памятью размером 512 Кбайт, одним принтером и двумя процессами по 512 Кбайт, каждый из которых хочет что-то напечатать. Процесс A запрашивает и получает принтер, затем начинает вычислять данные для печати. Еще не закончив расчеты, он превышает свой квант времени и выгружается на диск в область подкачки.

Теперь работает процесс B и безуспешно пытается обратиться к принтеру. В данный момент мы получили потенциальную тупиковую ситуацию, поскольку процесс A оккупирует принтер, а процесс B занимает память, и ни один из них не может продолжать работу без ресурса, удерживаемого другим. К счастью, не запрещено выгрузить (забрать) память у процесса B, переместив его на диск в область подкачки и загрузив с диска в память процесс A. Теперь процесс A может закончить вычисления, выполнить печать и затем освободить принтер. Взаимоблокировки не происходит.

Невыгружаемый ресурс, в противоположность выгружаемому, — это такой ресурс, который нельзя забрать от текущего владельца, не уничтожив результаты вычислений. Если в момент записи компакт-диска внезапно отнять у процесса устройство для записи и передать его другому процессу, то в результате мы получим испорченный компакт-диск. Устройство для записи компакт-дисков не является выгружаемым в произвольный момент времени ресурсом.

Вообще говоря, взаимоблокировки касаются невыгружаемых ресурсов. Потенциальные тупиковые ситуации, в которые вовлечен противоположный вид ресурсов, обычно разрешаемы с помощью перераспределения ресурсов от одного процесса к другому. Поэтому мы сконцентрируем свое внимание на невыгружаемых ресурсах.

Последовательность событий, необходимых для использования ресурса, представлена ниже в абстрактной форме.

1. Запрос ресурса.
2. Использование ресурса.
3. Возврат ресурса.

Если ресурс недоступен, запрашивающий его процесс вынужден ждать. В некоторых операционных системах при неудачном обращении к ресурсу процесс автоматически блокируется и возобновляется только после того, как ресурс становится доступным. В других системах запрос ресурса, получивший отказ, возвращает код ошибки, тогда вызывающий процесс может подождать немного и повторить попытку.

3.3.2. Понятие взаимной блокировки

Определение взаимоблокировки формально можно озвучить так:

> *Группа процессов находится в тупиковой ситуации, если каждый процесс из группы ожидает события, которое может вызвать только другой процесс из той же группы.*

Так как все процессы находятся в состоянии ожидания, ни один из них не будет причиной какого-либо события, которое могло бы активизировать любой другой процесс в группе, и все процессы продолжают ждать до бесконечности. В этой модели мы предполагаем, что процессы имеют только один поток управления и что нет прерываний, способных активизировать заблокированный процесс. Условие отсутствия прерываний необходимо, чтобы предотвратить ситуацию, когда тот или иной заблокированный процесс активизируется, скажем, по сигналу тревоги и затем приводит к событию, которое освободит другие процессы в группе.

В большинстве случаев событием, которого ждет каждый процесс, является возврат какого-либо ресурса, в данный момент занятого другим процессом группы. Другими словами, каждый участник в группе процессов, зашедших в тупик, ожидает доступа к ресурсу, принадлежащему заблокированному процессу. Ни один из процессов не может работать, ни один из них не может освободить какой-либо ресурс и ни один из них не может возобновиться. Количество процессов и количество и вид ресурсов, имеющихся и запрашиваемых, здесь не важны. Результат остается тем же самым для любого вида ресурсов, аппаратных и программных.

Условия взаимоблокировки

Коффман (Coffman et al.) [70] и другие исследователи доказали, что для возникновения ситуации взаимоблокировки должны выполняться четыре условия.

1. Условие взаимного исключения. Каждый ресурс в данный момент или отдан ровно одному процессу, или доступен.
2. Условие удержания и ожидания. Процессы, в данный момент удерживающие полученные ранее ресурсы, вправе запрашивать новые ресурсы.
3. Условие отсутствия принудительной выгрузки ресурса. У процесса нельзя принудительным образом забрать ранее полученные ресурсы. Процесс, владеющий ими, должен сам освободить ресурсы.
4. Условие циклического ожидания. Должна существовать круговая последовательность из двух и более процессов, каждый из которых ждет доступа к ресурсу, удерживаемому следующим членом последовательности.

Для того чтобы произошла взаимоблокировка, должны выполниться все эти четыре условия. Если хоть одно из них отсутствует, тупиковая ситуация невозможна.

Моделирование взаимоблокировок

В [44] Холт (Holt) показал, как можно смоделировать четыре условия возникновения тупиков, используя направленные графы. Графы имеют два вида узлов:

процессы, показанные кружочками, и ресурсы, нарисованные квадратиками. Ребро, направленное от узла ресурса (квадрат) к узлу процесса (круг), означает, что ресурс ранее был запрошен процессом, получен и в данный момент используется этим процессом. На рис. 3.5, *а* ресурс R в настоящее время отдан процессу A.

Ребро, направленное от процесса к ресурсу, означает, что процесс в данный момент блокирован и находится в состоянии ожидания доступа к этому ресурсу. На рис. 3.5, *б* процесс B ждет ресурс S. На рис. 3.5, *в* мы видим взаимоблокировку: процесс C ожидает ресурс T, удерживаемый в настоящее время процессом D. Процесс D вовсе не намеревается освобождать ресурс T, потому что он ждет ресурс U, используемый процессом C. Оба процесса будут ждать до бесконечности. Цикл в графе означает наличие взаимоблокировки, циклично включающей процессы и ресурсы (предполагается, что в системе есть по одному ресурсу каждого вида). В этом примере циклом является последовательность C-T-D-U-C.

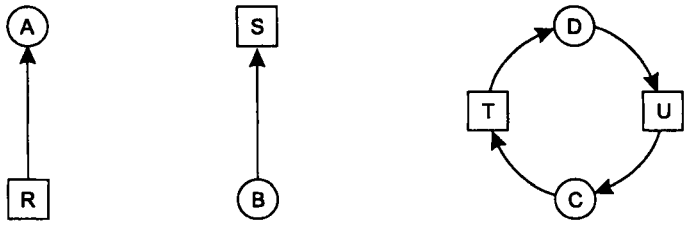


Рис. 3.5. Графы распределения ресурсов: *а* — ресурс занят; *б* — запрос ресурса; *в* — взаимоблокировка

Теперь рассмотрим пример того, как извлечь пользу из графов ресурсов. Представим, что у нас есть три процесса: A, B и C, и три ресурса: R, S и T. Последовательность запросов и возвратов ресурсов для трех процессов показана на рис. 3.6, *а–в*. Операционная система может запустить любой незаблокированный процесс в любой момент времени, значит, таковым может оказаться процесс A. Процесс A будет выполняться до тех пор, пока не закончит всю свою работу, затем будет запущен процесс B до его завершения и, наконец, процесс C.

Такой порядок не приводит к взаимоблокировке (не возникает соперничества за использование ресурсов), но здесь также по сути нет параллельной работы. Кроме запросов и возвратов ресурсов, процессы выполняют вычисления и ввод/вывод данных. Когда процессы работают последовательно, нереальна ситуация, при которой один процесс использует процессор, в то время как другой ждет завершения операции ввода/вывода. Таким образом, строго последовательная работа процессов не бывает оптимальной. С другой стороны, если вообще ни один процесс не выполняет операций ввода/вывода, алгоритм «кратчайшая задача — первая» работает эффективнее, чем циклический, поэтому в некоторой обстановке последовательный запуск всех процессов может быть наилучшим.

Теперь предположим, что процессы выполняют как расчеты, так и ввод/вывод, соответственно циклический алгоритм планирования является рациональ-

ным. Запросы ресурсов могут происходить в порядке, указанном на рис. 3.6, *г*. Если эти шесть запросов будут осуществлены в такой последовательности, в результате мы получим шесть графов, показанных на рис. 3.6, *д*–*к*. После запроса 4 процесс A блокируется в ожидании ресурса S (см. рис. 3.6, *з*). На двух следующих шагах также блокируются процессы B и C, в конечном счете приводя к циклу и взаимоблокировке на рис. 3.6, *к*.

Однако, как мы упоминали ранее, операционная система не обязана запускать процессы в каком-то особом порядке. В частности, если выполнение отдельного запроса приводит в тупик, операционная система вправе просто приостановить процесс без удовлетворения запроса (то есть не выполняя план процесса) до тех пор, пока это безопасно. На рис. 3.6 операционная система могла бы приостановить процесс B вместо того, чтобы отдавать ему ресурс S, если бы она знала о предстоящей взаимоблокировке. Работая только с процессами A и C, мы могли бы получить порядок запросов ресурсов и их возвратов, продемонстрированный на рис. 3.6, *л*, вместо показанного на рис. 3.6, *г*. Такая последовательность действий отражена графами на рис. 3.6, *м*–*с*, и она не приводит к тупику.

После шага *с* процесс B может получить ресурс S, потому что процесс A уже закончил свою работу, а процесс C имеет в своем распоряжении все необходимые ему ресурсы. Даже если затем процесс B, когда он запросит ресурс T, будет заблокирован, система не повиснет. Процесс B всего лишь будет ждать завершения работы процесса C.

Позже в этой главе мы изучим подробный алгоритм для принятия решений о распределении ресурсов, которые не приведут к взаимоблокировке. В данный момент важно понять, что графы ресурсов являются инструментом, позволяющим нам увидеть, станет ли заданная последовательность запросов/возвратов ресурсов причиной тупиковой ситуации. Мы всего лишь шаг за шагом осуществляем запросы и возвраты ресурсов и после каждого шага проверяем граф на содержание циклов. Если они есть, мы зашли в тупик; если нет, значит, взаимоблокировки тоже нет. Хотя мы рассматривали графы ресурсов для случая, когда в системе присутствует по одному ресурсу каждого типа, графы также можно построить для обработки ситуации с несколькими одинаковыми ресурсами [44]. Вообще говоря, при столкновении с взаимоблокировками практикуются четыре стратегии.

1. Пренебрежение проблемой в целом. Если вы проигнорируете проблему, возможно, затем она проигнорирует вас.
2. Обнаружение и восстановление. Позволить взаимоблокировке произойти, обнаружить ее и предпринять какие-либо действия.
3. Динамическое избежание тупиковых ситуаций с помощью аккуратного распределения ресурсов.
4. Предотвращение с помощью структурного опровержения одного из четырех условий, необходимых для взаимоблокировки.

Мы по очереди изучим каждый из этих методов в следующих четырех разделах.

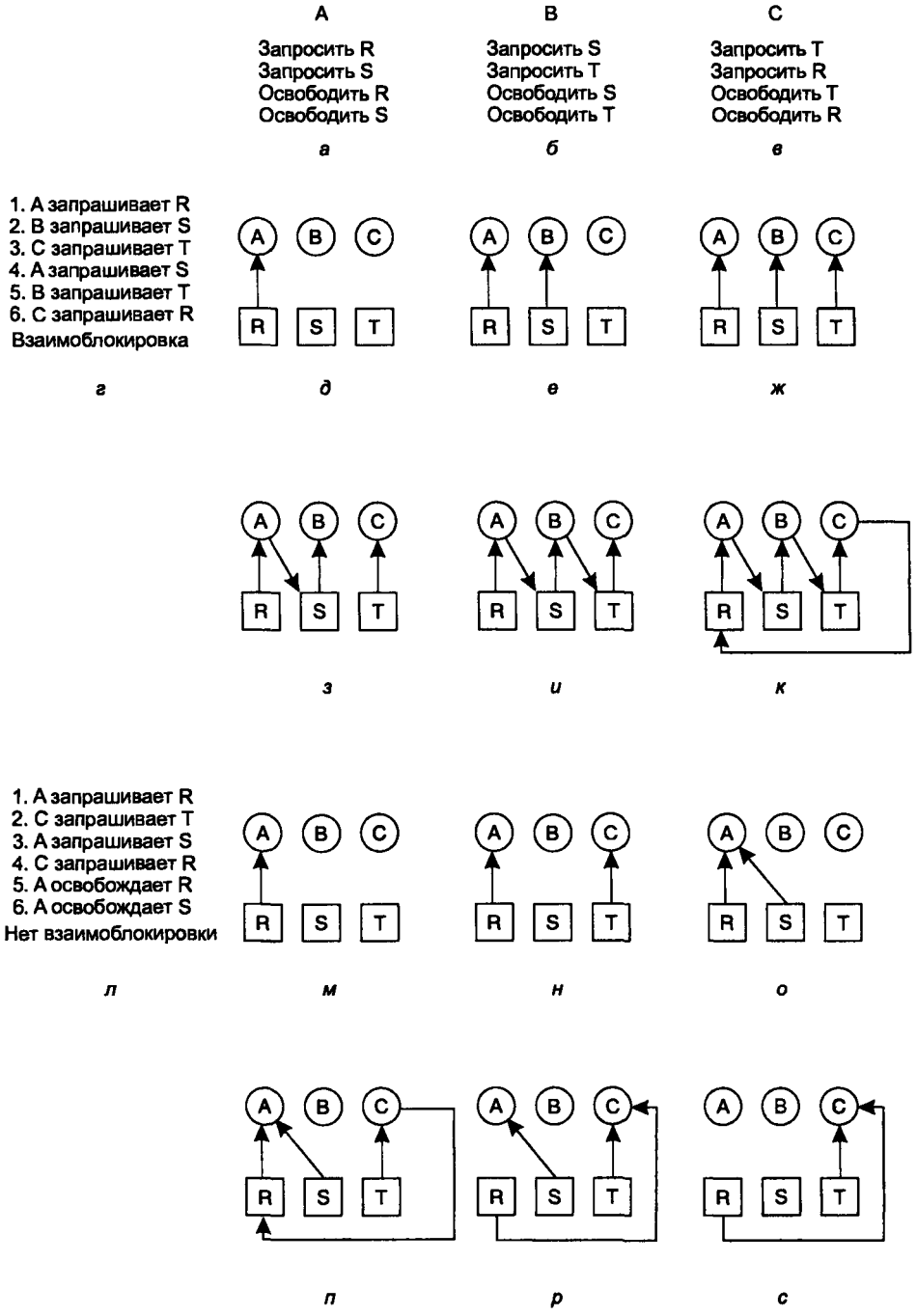


Рис. 3.6. Пример возникновения взаимоблокировки и способы избежать ее

3.3.3. Страусовый алгоритм

Самым простым подходом является «страусовый алгоритм»: воткните голову в песок и притворитесь, что проблемы вообще не существует. Различные люди отзываются об этой стратегии по-разному. Математики считают ее полностью неприемлемой и говорят, что взаимоблокировки нужно предотвращать любой ценой. Инженеры спрашивают, как часто встает подобная проблема, как часто система попадает в аварийные ситуации по другим причинам и насколько серьезны последствия взаимоблокировок. Если взаимоблокировки случаются в среднем один раз в пять лет, а сбои операционной системы, ошибки компилятора и поломки компьютера из-за неисправности аппаратуры происходят раз в неделю, то большинство инженеров не захотят добровольно уступать в производительности и удобстве для того, чтобы ликвидировать возможность взаимоблокировок.

Для усиления контраста между этими подходами добавим что большинство операционных систем потенциально страдают от взаимоблокировок, которые даже не обнаруживаются, не говоря уже об автоматическом выходе из тупика. Суммарное количество процессов в системе определяется количеством записей в таблице процесса. Таким образом, ячейки таблицы процесса являются ограниченным ресурсом. Если системный вызов fork получает отказ, в силу того что таблица целиком заполнена, разумно будет, что программа, вызывающая fork, подождет какое-то время и повторит попытку.

Теперь предположим, что система UNIX имеет 100 ячеек процессов. Работают десять программ, каждой необходимо создать 12 (под)процессов. После образования каждым процессом девяти процессов 10 исходных и 90 новых процессов заполнят таблицу до конца. Теперь каждый из десяти исходных процессов попадает в бесконечный цикл, состоящий из попыток разветвления и отказов, то есть возникает взаимоблокировка. Вероятность того, что произойдет подобное, минимальна, но это *могло бы* случиться. Должны ли мы отказаться от процессов и вызова fork, чтобы устранить данную проблему?

Максимальное количество открытых файлов также ограниченно размером таблицы *i*-узлов, следовательно, когда таблица заполняется целиком, возникает та же самая проблема. Пространство для подкачки файлов на диск является еще одним ограниченным ресурсом. Фактически почти каждая таблица в операционной системе представляет собой ресурс, имеющий пределы. Должны ли мы упразднить их все из-за того, что теоретически ожидаема ситуация, когда в группе из n процессов каждый может потребовать $1/n$ от целого, а затем попытаться получить еще часть?

Большая часть операционных систем, включая UNIX и Windows, игнорируют эту проблему. Они исходят из предположения, что большинство пользователей скорее предпочтут иметь дело со случающимися время от времени взаимоблокировками, чем с правилом, по которому всем пользователям разрешается только один процесс, один открытый файл и т. д. Если бы можно было легко устранить взаимоблокировки, не возникло бы столько разговоров на эту тему. Сложность заключается в том, что цена достаточно высока, и в основном она, как мы вскоре увидим, исчисляется в наложении неудобных ограничений на процессы. Таким образом, мы столкнулись с неприятным выбором между удобством и корректностью и множеством дискуссий о том, что более важно и для кого. При всех этих условиях трудно найти верное решение.

3.3.4. Обнаружение и устранение взаимоблокировок

Вторая техника представляет собой обнаружение и восстановление. Здесь система не пытается предотвратить попадание в тупиковые ситуации. Каждый раз, когда запрашивается или освобождается новый ресурс, то есть когда обновляется граф ресурсов, система проверяет, имеются ли в нем циклы. Если цикл есть, один из входящих в него процессов принудительно завершается. Это повторяется до тех пор, пока взаимная блокировка не будет устранена.

Более грубый метод не анализирует граф ресурсов, а просто проверяет наличие процессов, которые были заблокированы долго, скажем, в течение одного часа. Если такие процессы обнаруживаются, они завершаются.

Стратегия обнаружения и восстановления применяется в больших компьютерных системах, особенно в системах пакетной обработки, где принудительное завершение и повторный запуск обычно приемлемы. Тем не менее необходимо с осторожностью производить восстановление любых модифицированных файлов и устранение любых побочных эффектов, которые могли произойти.

3.3.5. Предотвращение взаимоблокировок

Как мы видели, уклонение от взаимоблокировок, в сущности, невозможно, так как оно требует наличия никому не известной информации о будущих процессах. Тогда возникает справедливый вопрос: как же реальные системы избегают попадания в тупики? Для того чтобы ответить на этот вопрос, вернемся назад к четырем условиям, сформулированным в [11] (см. раздел «Условия взаимоблокировки» данной главы), и посмотрим, дадут ли они нам ключ к разрешению проблемы. Если мы сумеем гарантировать, что хотя бы одно из этих условий никогда не будет выполнено, тогда взаимоблокировки станут конструктивно невозможными [41].

Сначала попробуем атаковать условие взаимного исключения. Если в системе нет ресурсов, отданных в единоличное пользование одному процессу, мы никогда не попадем в тупик. Но в равной степени понятно, что если позволить двум процессам одновременно печатать данные на принтере, воцарится хаос. Используя подкачку выходных данных для печати, несколько процессов могут одновременно генерировать свои выходные данные. В такой модели только один процесс, который фактически запрашивает физический принтер, является демоном принтера. Так как демон не запрашивает никакие другие ресурсы, для принтера тупики исключаются.

К сожалению, не все устройства поддерживают подкачку (свопинг) данных (таблица процессов — дело резидентное). Кроме того, конкуренция за дисковое пространство для подкачки сама по себе чревата тупиком. Что получится, если два процесса заполнили своими выходными данными каждый по половине диска, отведенного под свопинг, и ни один из них не закончил вычисления? Демон может быть запрограммирован так, что начнет печать, не дожидаясь подкачки всех выходных данных, и принтер тогда простоит впустую в том случае, если вычисляющий процесс решил подождать несколько часов после первого пакета вы-

ходных данных. По этой причине обычно демоны программируют так, что они начинают печать только после того, как файл выходных данных целиком станет доступен. В этом же случае мы получаем два процесса, каждый из которых обработал часть выходных данных, но не все и не в состоянии продолжать вычисления дальше. Ни один из двух процессов никогда не завершится, то есть имеет место взаимоблокировка на диске.

Второе из условий, сформулированных Коффманом (Coffman) и другими, кажется, все же подает надежду. Если у нас получится уберечь процессы, занимающие некоторые ресурсы, от ожидания остальных ресурсов, мы устраним тупиковую ситуацию. Один из способов достижения этой цели состоит в требовании, следуя которому любой процесс должен запрашивать все необходимые ресурсы до начала работы. Если все ресурсы доступны, процесс получит все, что ему нужно, и сможет работать до успешного завершения. Если один или несколько ресурсов заняты, процессу ничего не предоставляется, и он непременно попадает в состояние ожидания.

Первая проблема, вносимая этим подходом, заключается в том, что многие процессы не знают, сколько ресурсов им понадобится, до тех пор пока не начнут работу. На самом деле, если бы они обладали подобными сведениями, мог бы использоваться и алгоритм банкира. Другая проблема состоит в том, что ресурсы не будут расходоваться оптимально. Возьмем, например, процесс, который читает данные с входной ленты, анализирует их в течение часа и затем пишет выходную ленту, а заодно и чертит результаты на плоттере. Если все ресурсы нужно запрашивать заранее, то процесс целый час не позволит работать накопителю на магнитной ленте и принтеру.

И все-таки некоторые пакетные системы на мэйнфреймах требуют, чтобы пользователи объявляли свои аппетиты в отношении ресурсов в первой строке каждого задания. Затем система немедленно запрашивает все ресурсы и сохраняет их до окончания задачи. Этот способ накладывает ограничения на деятельность программиста и излишне расточителен, зато предотвращает безвыходные ситуации.

Немного отличный метод, позволяющий нарушить условие удержания и ожидания, вытекает из наложении следующего требования на процесс, запрашивающий ресурс: процесс сначала должен временно освободить все используемые им в данный момент ресурсы. Затем этот процесс пытается сразу получить все необходимое.

Попытка исключить третье условие (нет принудительной выгрузки ресурса) подает еще меньше надежд, чем устранение второго условия. Если процесс получил принтер и в данный момент печатает выходные данные, насильственное изъятие принтера по причине недоступности требуемого плоттера в лучшем случае сложно, а в худшем — невозможно.

Остается только одно условие. Циклическое ожидание можно устранить несколькими путями. Один их них: просто следовать правилу, гласящему, что процессу дано право только на один ресурс в конкретный момент времени. Если нужен второй ресурс, процесс обязан освободить первый. Но подобное ограничение неприемлемо для процесса, копирующего огромный файл с магнитной ленты на принтер.

Другой способ уклонения от циклического ожидания заключается в поддержке общей нумерации всех ресурсов, как показано на рис. 3.7, *а*. Тогда действует

следующее правило: процессы могут запрашивать ресурс, когда хотят этого, но все запросы должны быть сделаны в соответствии с нумерацией ресурсов. Процесс может запросить сначала принтер, затем накопитель на магнитной ленте, но не вправе сначала потребовать плоттер, а затем принтер.

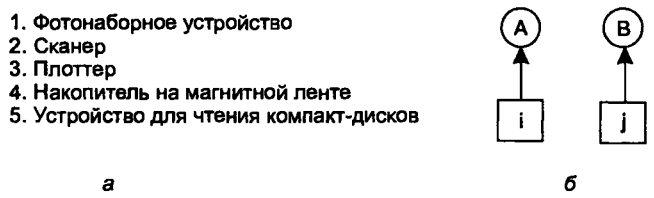


Рис. 3.7. *а* — пронумерованные ресурсы; *б* — граф ресурсов

При выполнении такого соглашения граф распределения ресурсов никогда не будет иметь циклов. Покажем, что это так, в случае двух процессов (рис. 3.7, *б*). Мы попадаем в тупик, только если процесс A запросит ресурс j, а процесс B обратится к ресурсу i. Предположим, что ресурсы i и j различны, значит, они имеют разные номера. Если i > j, тогда процессу A не позволяется запрашивать ресурс j, потому что его номер меньше, чем номер уже имеющегося у него ресурса. Если же i < j, процесс B не может запрашивать ресурс i, так как этот номер меньше номера уже занятого им ресурса. Так или иначе, взаимоблокировка исключена.

При работе с несколькими процессами сохраняется та же самая логика. В каждый момент времени один из предоставленных ресурсов будет иметь наивысший номер. Процесс, использующий этот ресурс, уже никогда не запросит другие занятые ресурсы. Он или закончит свою работу или, в худшем случае, запросит ресурс с еще большим номером, а любой такой ресурс окажется доступен. В итоге процесс завершит работу и освободит свои ресурсы. На этот момент сложится ситуация, когда ресурс с высшим номером уже занят каким-то другим процессом, который также сможет нормально завершиться. То есть существует алгоритм, по которому все процессы отрапортуют о выполнении без попадания в тупик.

Вариантом этого алгоритма является схема, в которой отбрасывается требование приобретения ресурсов в строго возрастающем порядке, но сохраняется условие, что процесс не может запросить ресурсы с меньшим номером, чем уже у него имеющиеся. Если процесс на начальной стадии запрашивает ресурсы 9 и 10, затем освобождает их, то это равнозначно тому, как если бы он начал работу заново, поэтому нет причины теперь запрещать ему запрос ресурса 1.

Несмотря на то что систематизация ресурсов с помощью их нумерации устраняет проблему взаимоблокировки, бывают ситуации, когда невозможно найти порядок, удовлетворяющий всех. Когда ресурсы включают в себя области таблицы процессов, дисковое пространство для подкачки данных, закрытые записи базы данных и другие абстрактные ресурсы, число потенциальных объектов интереса и вариантов их применения может быть настолько огромным, что никакая систематизация не спасет.

В табл. 3.3 подведены итоги различных методов для предотвращения тупиков.

Таблица 3.3. Методы предотвращения тупиков

Условие	Метод
Взаимное исключение	Организовывать подкачку данных
Удержание и ожидание	Запрашивать все ресурсы на начальной стадии
Нет принудительной выгрузки ресурса	Отобрать ресурсы
Циклическое ожидание	Пронумеровать ресурсы и упорядочить

3.3.6. Избежание взаимоблокировок

Рассматривая обнаружение взаимоблокировок, мы неявно предполагали, что когда процесс запрашивает ресурсы, он требует их все сразу (матрица R на рис. 3.6). Однако в большинстве систем ресурсы запрашиваются поочередно, по одному. Система должна уметь решать, является ли предоставление ресурса безопасным или нет, и удовлетворять процесс только в первом случае. Таким образом, неизбежен новый вопрос: существует ли алгоритм, который никогда не повлечет ситуации взаимоблокировки, все время делающий правильный выбор? Ответом является условное «да» — мы можем избежать тупиков, но только если заранее будет доступна определенная информация. В следующем подразделе мы изучим способы уклонения от взаимоблокировок с помощью аккуратного предоставления ресурсов.

Алгоритм банкира для одного вида ресурсов

Алгоритм планирования, позволяющий избегать взаимоблокировок, был разработан Дейкстрой (Dijkstra, [27]) и носит название *алгоритма банкира*. Модель основана на примере банкира в маленьком городке, имеющего дело с группой клиентов, которым он выдал ряд кредитов. Алгоритм проверяет, ведет ли выполнение каждого запроса к небезопасному состоянию. Если да, запрос отклоняется. Если удовлетворение запроса ничем не грозит, ресурс предоставляется процессу. На рис. 3.8, *а* мы видим четырех клиентов, каждый из которых получил определенное количество единиц кредита (например, 1 единица равна 1 К долларов). Банкир знает, что не всем клиентам понадобится вся сумма немедленно, поэтому он зарезервировал только 10 единиц, а не все 22, которые требуются клиентам. (Чтобы провести аналогию с компьютерной системой, считаем, что клиенты — это процессы, единицами, скажем, являются накопители на магнитной ленте, а банкир — это операционная система.)

Клиенты вращаются в соответствующем бизнесе, время от времени прося у банка ссуды (то есть запрашивая ресурсы). В некоторый момент возникает ситуация, показанная на рис. 3.8, *б*. Список покупателей, фиксирующий, кому одолжены ресурсы, а также максимальный доступный кредит, называется *состоянием* системы.

Состояние системы считается *безопасным*, если существует последовательность состояний, когда все просители берут максимальный заем, до исчерпания кредита. Рассмотренное состояние безопасно, поскольку остались две единицы, и банкир может задержать все обращения, кроме запросов клиента или процесса Марвин, таким образом, позволяя процессу Марвин завершиться и вернуть все четыре отданных ему ресурса. Имея на руках четыре единицы, банкир может от-

дать их или клиенту Сьюзан, или Сьюзан, обеспечивая их необходимыми единицами и т. д.

	Имеет	Max
Энди	0	6
Барбара	0	5
Марвин	0	4
Сьюзан	0	7

Свободно: 10

а

	Имеет	Max
Энди	1	6
Барбара	1	5
Марвин	2	4
Сьюзан	4	7

Свободно: 2

б

	Имеет	Max
Энди	1	6
Барбара	2	5
Марвин	2	4
Сьюзан	4	7

Свободно: 1

в

Рис. 3.8. Три состояния распределения ресурсов: а — безопасное; *б* — безопасное; *в* — небезопасное

Рассмотрим, что могло бы произойти, если бы в ситуации на рис. 3.8, *б* был удовлетворен запрос еще одной единицы для клиента Сьюзан. Мы попали бы в состояние рис. 3.8, *в*, не являющееся безопасным. Если бы все клиенты вдруг запросили максимальные ссуды, то банкир не сумел бы их обеспечить, и мы попали бы в тупик. Небезопасное состояние *не обязано* приводить к взаимоблокировке, так как клиентам не обязательно потребуется весь доступный кредит, но банкиру не следует рассчитывать на такую ситуацию.

Алгоритм банкира рассматривает каждый запрос по мере поступления и проверяет, приведет ли его удовлетворение к безопасному состоянию. Если да, процесс получает ресурс, иначе запрос откладывается на более позднее время. Чтобы понять, является ли состояние безопасным, банкир оценивает, достаточно ли ресурсов для завершения работы какого-либо клиента. Если да, эти ссуды считаются погашенными, после чего проверяется следующий ближайший к верхнему пределу займа клиент и т. д. Если, в конце концов, все ссуды могут быть погашены, состояние является безопасным, и исходный запрос можно удовлетворить.

Траектории ресурсов

Предыдущий алгоритм был описан в терминах одного класса ресурсов (то есть в нашем распоряжении только принтеры или только ленточные накопители, но не то и другое вместе). На рис. 3.9 представлена модель для системы с двумя процессами и двумя ресурсами, например принтером и плоттером. Горизонтальная ось отображает номера команд, выполняемых процессом A. По вертикальной оси отложены номера команд, выполняемых процессом B. В команде I_1 процесс A запрашивает принтер, в команде I_2 ему требуется плоттер. Принтер и плоттер освобождаются командами I_3 и I_4 соответственно. Процессу B необходим плоттер с команды I_5 по команду I_7 и принтер с команды I_6 по команду I_8.

Каждая точка на диаграмме представляет совместное состояние двух процессов. Изначально система находится в точке p, когда ни один процесс еще не выполнил ни одну инструкцию. Если планировщик запустит процесс A первым, мы

попадем в точку q, в которой процесс A выполнил какое-то количество команд, а процесс B еще ничего не сделал. В точке q траектория становится вертикальной, показывая, что планировщик решил запустить в работу процесс B. При наличии одного процессора все отрезки траектории могут быть только вертикальными или горизонтальными, но не наклонными. Кроме того, движение всегда происходит на «север» или «восток» (вверх и вправо) и никогда на «юг» или «запад» (вниз и влево), так как процессы не могут работать в обратном направлении.

Когда процесс A пересекает линию I_1 на отрезке от точки r до точки s, он запрашивает и получает принтер. Когда процесс B достигает точки t, он запрашивает плоттер.

Особенно интересны заштрихованные области. Область со штриховкой из верхнего левого угла в правый нижний представляет промежуток времени, когда оба процесса занимают принтер. Правило взаимного исключения делает попадание в эту область невозможным. Вторая заштрихованная область соответствует тому, что оба процесса используют плоттер, и это также невозможно.

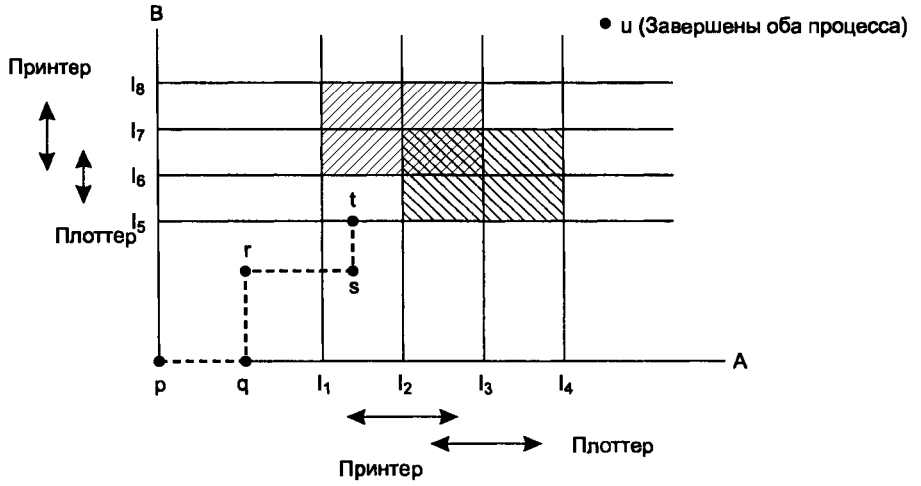


Рис. 3.9. Две траектории ресурсов процессов

Если система войдет в прямоугольник, ограниченный линиями I_1 и I_2 по сторонам и линиями I_5 и I_6 сверху и снизу, она в конце концов доберется до пересечения линий I_2 и I_6. В этот момент процесс A запросит плоттер, а процесс B потребует принтер, но оба ресурса будут к тому времени заняты. Получается, что тупиковым является целый прямоугольник и в него нельзя входить. В точке t единственно безопасный вариант состоит в том, чтобы оставить процесс A работать до тех пор, пока он не достигнет команды I_4. После нее любая траектория дойдет до точки u.

Важный для понимания момент заключается в том, что в точке t процесс B запрашивает ресурс. Система должна принять решение: предоставлять его или нет. Если выдается разрешение, система попадает в небезопасную область и в итоге блокируется. Чтобы избежать тупика, нужно приостановить процесс B до тех пор, пока процесс A не запросит и не освободит плоттер.

Алгоритм банкира для нескольких видов ресурсов

Такую графическую модель трудно применить для общего случая, когда имеются несколько процессов и несколько различных классов ресурсов и в каждом классе может быть несколько представителей (например, два плоттера и три накопителя). Но алгоритм банкира поддается обобщению для управления системой с несколькими видами ресурсов. На рис. 3.10 показано, как он работает.

На рисунке изображены две матрицы. Матрица слева показывает, сколько ресурсов каждого вида занимает в настоящее время каждый из пяти процессов. Матрица справа показывает количество ресурсов, которое нужно добавить каждому процессу для успешного завершения. Эти матрицы на рис. 3.6 назывались C и R. Как и в случае одного вида ресурсов, процессы должны точно определять необходимое суммарное количество ресурсов до начала работы для того, чтобы система могла рассчитать правую матрицу в каждый момент времени.

Процесс	Накопители на магнитных дисках	Плоттеры	Сканеры	Устройства для чтения компакт-дисков
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Распределенные ресурсы

Процесс	Накопители на магнитных дисках	Плоттеры	Сканеры	Устройства для чтения компакт-дисков
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Ресурсы, которые еще нужны

E = (6342)
P = (5322)
A = (1020)

Рис. 3.10. Алгоритм банкира в системе с несколькими типами ресурсов

Три вектора, изображенные справа от матриц, показывают соответственно существующие ресурсы (вектор E), занятые ресурсы (вектор P) и доступные ресурсы (вектор A). Из вектора E мы видим, что система имеет шесть накопителей на магнитной ленте, три плоттера, четыре принтера и два устройства для чтения компакт-дисков. Из них заняты в данный момент пять накопителей, три плоттера, два принтера и два устройства для чтения компакт-дисков. Чтобы увидеть этот факт, нужно просуммировать четыре столбца, соответствующие ресурсам, в левой матрице. Вектор доступных ресурсов является разницей между тем, что присутствует в системе, и тем, что используется в настоящее время.

Теперь можно изложить алгоритм для проверки безопасности состояния системы.

1. Ищем в матрице R строку, соответствующую процессу, чьи неудовлетворенные потребности ресурсов меньше или равны вектору A. Если такой строки не существует, система в конце концов попадет в тупик, так как ни один процесс не может проработать до успешного завершения.

2. Допускаем, что процесс, строку которого выбрали в пункте 1, запрашивает все необходимые ресурсы (гарантируется, что это возможно) и заканчивает работу. Отмечаем этот процесс как завершенный и прибавляем все его ресурсы к вектору A.
3. Повторяем шаги 1 и 2 до тех пор, пока или все процессы будут помечены как завершенные — и состояние в этом случае является безопасным, или произойдет взаимоблокировка — тогда состояние небезопасно.

Если на первом шаге можно выбрать несколько процессов, не имеет значения, какой из них будет взят: общий резерв доступных ресурсов или увеличится, или, в худшем случае, останется неизменным.

Вернемся к примеру на рис. 3.10. Текущее состояние является безопасным. Предположим, что процесс B в данный момент запрашивает принтер. На этот запрос можно ответить положительно, потому что получающееся в результате состояние все еще будет безопасным (процесс D может доработать до конца, затем процесс A или E, затем остальные).

Теперь представим, что после того, как процесс B получил один из двух оставшихся принтеров, процесс E потребует последний принтер. Удовлетворение этого запроса сократит вектор доступных ресурсов до (1 0 0 0), что приведет к взаимоблокировке процессов. Ясно, что следует отложить на время запрос процесса E.

Дейкстра (Dijkstra) впервые опубликовал алгоритм банкира в 1965 году. С тех пор практически каждая книга по операционным системам описывает его в деталях. Различным аспектам этого алгоритма было посвящено бессчетное количество статей. К сожалению, мало у кого из авторов хватило смелости показать, что хотя алгоритм замечателен в теории, на практике он, по существу, бесполезен, поскольку нечасто можно определить заранее, сколько ресурсов потребуется процессам в будущем. Кроме того, количество процессов не фиксировано, оно динамически изменяется по мере входа пользователей в систему и выхода из нее. И, более того, ресурсы, про которые считалось, что они доступны, могут внезапно исчезнуть (например, накопитель на магнитной ленте может сломаться). Таким образом, на практике немногие системы, если это вообще имеет место, используют алгоритм банкира для уклонения от взаимоблокировок.

Обобщая, скажем, что описанные выше алгоритмы, объединенные названием «предотвращение», накладывают чересчур сильные ограничения. Те же алгоритмы, которые входят в группу «избежание», требуют для своей работы информацию, которая не всегда имеется. Если вам удалось придумать общий алгоритм, который в реалии работает так же хорошо, как и в теории, сообщите о нем в ближайший научный компьютерный журнал.

Для некоторых процессов разработаны превосходные специализированные алгоритмы. Например, в системах управления базами данных часто встречается ситуация, когда сначала запрашивается блокировка нескольких записей, а затем эти записи обновляются. Но, когда одновременно работает несколько процессов, возникает реальная опасность тупика.

Часто используемый подход называется *двухфазным блокированием*. В первой фазе, то есть на первом этапе, процесс пытается заблокировать все требуемые записи по одной за раз. Если операция успешна, процесс переходит ко второму

этапу, выполняя обновление блокировок и освобождение ресурсов. Никакой полезной работы на первом этапе не совершается.

Если во время первой фазы какая-либо необходимая запись оказывается уже заблокированной, процесс просто сбрасывает все свои блокировки и начинает первую фазу заново. В некотором смысле этот метод похож на схему, в которой запрос всех необходимых ресурсов происходит загодя или, по крайней мере, перед тем, как произойдет что-то необратимое. В некоторых версиях двухфазного блокирования, если блокировка встретилась во время первой фазы, не происходит возврата ресурсов и возобновления работы процесса. В таких версиях может возникнуть тупиковая ситуация.

Но эту стратегию нельзя обобщить. В системах реального времени и системах контроля процессов, например, недопустимо частично завершить процесс из-за того, что ресурс недоступен, а потом начинать все заново. Также недопустимо перезапускать процесс, если он прочел сообщение из сети или написал его, обновил файлы и сделал что-нибудь еще, что не может быть безопасно повторено. Алгоритм работает только в тех ситуациях, когда программист очень тщательно подготовил все таким образом, что программу нетрудно остановить в любой точке первой фазы и запустить заново. Многие программы не могут быть структурированы таким образом.

3.4. Обзор ввода/вывода в MINIX

Структура системы ввода/вывода в MINIX показана на рис. 3.4. Четыре верхних уровня этой структуры соответствуют четырем уровням с рис. 2.14. В последующих разделах мы вкратце рассмотрим каждый из этих уровней, делая акцент на драйверах устройств. Обработка прерываний в MINIX была рассмотрена в предыдущей главе, а аппаратно-зависимый ввод/вывод будет обсуждаться в главе 5.

3.4.1. Обработчики прерываний в MINIX

Большинство драйверов инициируют ввод/вывод и переходят в состояние блокировки, ожидая, когда прибудет сообщение. Обычно это сообщение генерируется обработчиком прерываний устройства. Существуют другие драйверы, которые не запускают физических процессов ввода/вывода (пример — чтение с RAM-диска или вывод текста на «отображаемый в память экран» — видеопамять), не опираются на прерывания и не ждут сообщений от устройств. Механизм генерации сообщений и переключения задач, управляемый прерываниями, в предыдущей главе обсуждался очень подробно, и ниже нечего добавить. Но обработчики прерываний не только генерируют сообщения. Часто они делают некоторую обработку входных и выходных данных на самом низком уровне. Здесь мы рассмотрим это применение в общем, а детально мы вернемся к этой теме, когда будем демонстрировать код для различных устройств.

Обратимся теперь к некоторым реальным устройствам ввода/вывода. Мы начнем с дисков. Затем мы также познакомимся с часами, клавиатурами и экранами.

Существует множество типов дисков. К наиболее часто встречающимся относятся магнитные диски (жесткие и гибкие). Их особенностью является одинаковая скорость чтения и записи, что делает их идеальными в качестве дополнительной памяти (страничная подкачка файлов, файловые системы и т. д.). Иногда, с целью создания высоконадежного устройства хранения, используются наборы жестких магнитных дисков. Для распространения программ, данных и фильмов применяются различные виды оптических дисков. В следующих разделах мы сначала познакомимся с аппаратной частью, а затем с программным обеспечением для этих устройств.

Для дисков ввод и вывод в основном сводятся к передаче устройству соответствующей команды и ожиданию ее завершения. Большую часть работы выполняет контроллер диска, поэтому обработчик прерывания весьма прост. Мы видели, что весь код обработчика прерываний для жесткого диска умещается в три строки и содержит единственную операцию ввода/вывода, считывающую единственный байт, с целью определить состояние контроллера. Конечно, если бы все прерывания обрабатывались так просто, наша жизнь была бы намного легче.

Но иногда бывает нужно выполнять более сложные действия. Механизм передачи сообщений имеет свою цену. Когда прерывания происходят часто, а объем передаваемых данных невелик, имеет смысл усложнить обработчик и отложить передачу сообщения до следующего прерывания, чтобы дать обслуживающей устройство задаче больше времени. В MINIX таким путем обрабатываются прерывания от таймера. При обработке большинства сигналов от таймера основная задача — обновление значения системного времени. Это можно сделать и не прибегая к задаче таймера. Поэтому обработчик прерываний таймера, вместо того чтобы посылать сообщение, увеличивает значение переменной, названной pending_ticks. Тогда текущее время рассчитывается как сумма времени, которое было записано при последнем выполнении задачи таймера, и значения в переменной pending_ticks. Когда управление все-таки передается задаче таймера, она увеличивает системное время на величину pending_ticks и обнуляет эту переменную. Обработчик прерывания таймера проверяет значения некоторых других переменных и передает сообщение задаче таймера только в том случае, если для нее есть реальная работа, например передача сигнала срабатывания таймера или планирование очередного процесса. Кроме того, может быть отправлено сообщение задаче терминала.

Рассматривая задачу терминала, мы видим еще одну вариацию на тему обработки прерываний. Эта задача обслуживает несколько различных типов устройств, включая клавиатуру и линии связи RS-232. Каждое из них имеет собственный обработчик прерываний. Клавиатура полностью соответствует описанию устройства, которое с каждым прерыванием выполняет очень малый объем ввода/вывода. На PC прерывание происходит каждый раз, когда нажимается или отпускается клавиша. Это относится и к специальным клавишам, таким как SHIFT или CTRL. Но если отбросить специальные клавиши, можно сказать, что с каждым прерыванием передается примерно половина символа. Так как с таким объемом информации задача терминала мало что может сделать путного, имеет смысл передавать сообщение только в том случае, если оно содержит законченную информацию. Детали мы рассмотрим позже, а сейчас скажем лишь, что обработчик

прерываний клавиатуры считывает данные и выполняет низкоуровневую фильтрацию, отбрасывая события, которые можно игнорировать, например отпускание обычной клавиши. (Игнорировать отпускание специальной клавиши, такой как SHIFT, нельзя.) Затем коды событий помещаются в очередь, которая позже обрабатывается задачей терминала.

Видно, что обработчик прерываний клавиатуры не вписывается в представленную выше простую парадигму, так как он вообще не посылает никаких сообщений. Вместо этого, добавив очередной код в очередь, обработчик модифицирует значение переменной tty_timeout, которая проверяется задачей таймера. Если очередь не меняется, эта переменная также не модифицируется. Когда произойдет следующее прерывание, обработчик прерываний таймера посылает терминалу сигнал, если очередь была изменена. Обработчики прерываний от подобных терминалу устройств, например от линий RS-232, функционируют подобным же образом. Заметьте, что сообщения задаче терминала передаются по мере получения новых символов, но сообщение не обязательно генерируется на каждый отдельный символ. Если данные поступают быстро, то с одним сообщением может быть передано несколько накопленных за прошедший период символов. Кроме того, состояние всех терминалов проверяется каждый раз, когда задача терминала получает сообщение.

3.4.2. Драйверы устройств в MINIX

Для каждого из классов устройств ввода/вывода в MINIX существует отдельная задача (драйвер). Эти драйверы являются полноценными процессами, каждый со своим состоянием, регистрами, стеком и т. д. Друг с другом драйверы при необходимости взаимодействуют при помощи стандартного механизма передачи сообщений. Код простых драйверов собран в одном файле, например clock.c. Другие драйверы, такие как для RAM-диска, жесткого диска и дискет, имеют отдельные файлы с кодом для поддержки устройств каждого типа, а также набор общих подпрограмм в driver.c, необходимых для всех драйверов. Таким образом, уровень драйверов устройств на рис. 3.4 разбивается на два подуровня: аппаратно-зависимый и аппаратно-независимый. Такое разбиение упрощает адаптацию к различным конфигурациям оборудования. При этом, хотя используется довольно большое количество общего кода, драйвер каждого типа дисков работает как отдельная задача для поддержки быстрой передачи данных.

Подобным же образом организован и исходный код задачи терминала, где аппаратно-независимый код помещен в файле tty.c, а в отдельных файлах находится код для поддержки различных типов устройств, таких как клавиатуры, последовательные интерфейсы и псевдотерминалы. Но в этом случае все типы устройств обслуживает один процесс.

Кроме того, для групп сходных устройств, например для дисков и терминалов, есть еще и заголовочные файлы. Файл driver.h поддерживает все драйверы блочных устройств, а файл tty.h предоставляет общие определения для всех типов терминалов.

Основное различие между драйверами устройств и обычными процессами в том, что все драйверы скомпонованы в один файл и, таким образом, имеют об-

щее адресное пространство. Благодаря этому, если два разных драйвера вызывают общую процедуру, в ядре MINIX будет только одна ее копия.

Подобная организация обеспечивает высокую модульность и умеренно эффективна. Также это одно из мест, где MINIX сильно отличается от UNIX. В MINIX процесс, для того чтобы прочитать файл, посылает сообщение файловой системе. В свою очередь, файловая система может послать сообщение драйверу диска, запрашивая чтение необходимого блока. Эта последовательность (несколько упрощенная по сравнению с реальностью) изображена на рис. 3.11, *а*. За счет того, что взаимодействие происходит через механизм сообщений, обеспечивается стандартный интерфейс коммуникаций между частями системы. Тем не менее, по той причине, что все драйверы находятся в адресном пространстве ядра, они при необходимости легко могут обращаться к таблице процессов и прочим структурам данных.

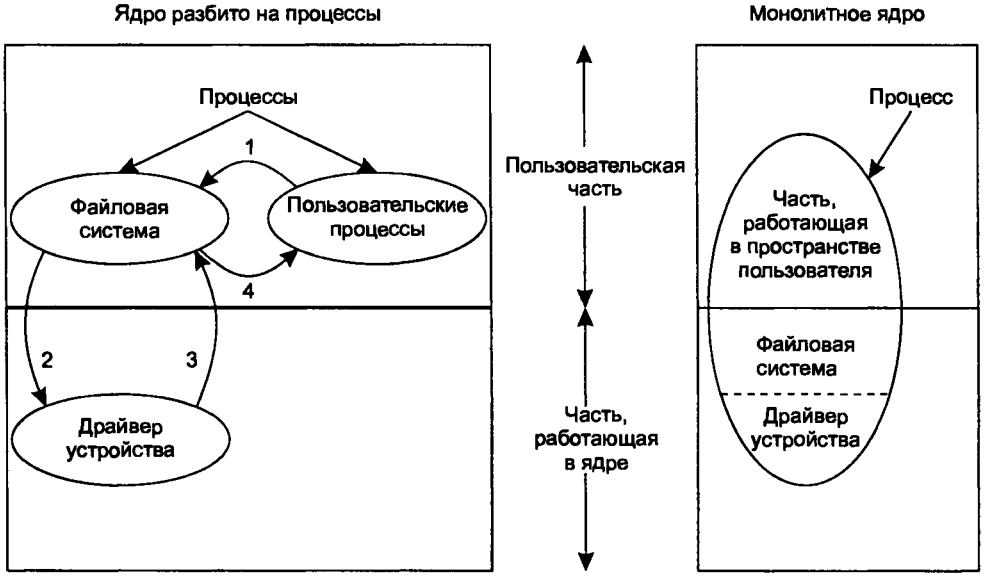


Рис. 3.11. Два способа построения взаимодействия пользователя и системы: *а* — сообщения с запросами и ответами между тремя независимыми процессами; *б* — передача управления в адресное пространство ядра из адресного пространства пользователя по ловушке. Файловая система вызывает драйвер ядра как устройство. Вся операционная система есть часть каждого процесса

В UNIX у каждого процесса есть две части: одна в пространстве пользователя и другая в пространстве ядра, как показано на рис. 3.11, *б*. Когда процесс делает

системный вызов, операционная система некоторым волшебным образом переключается из пользовательской части в «ядерную». Такой механизм является пережитком MULTICS, где переключение было простым вызовом процедуры, а не ловушкой с сохранением состояния пользовательской части процесса, как в UNIX.

В UNIX драйверы устройств являются простыми подпрограммами ядра, которые может вызывать часть процесса, загруженная в пространстве ядра. Когда драйверу надо дождаться прерывания, он вызывает процедуру, и процесс засыпает до тех пор, пока прерывание его не разбудит. Заметьте, что пользовательский процесс при этом также приостанавливается, так как это две части одного процесса.

Можно бесконечно перечислять аргументы о преимуществах монолитных систем (таких как UNIX) перед структурированными в виде процессов (наподобие MINIX), и наоборот. Используемый в MINIX подход обеспечивает лучшую модульность, более прозрачный интерфейс между компонентами и легко расширяется на распределенные системы, где разные процессы могут выполняться на разных машинах. Подход UNIX более эффективен, так как вызов подпрограммы гораздо быстрее, чем передача сообщения. MINIX разбита на множество процессов, так как мы убеждены, что по мере роста производительности компьютерных систем ясная структура программного обеспечения будет важнее, чем небольшая жертва производительности. Но знайте, что многие разработчики операционных систем не разделяют это убеждение.

В этой главе обсуждаются драйверы RAM-диска, жесткого диска, таймера и терминала. Стандартная комплектация MINIX включает в себя также драйверы для флоппи-дисковода и принтера, которые мы подробно не рассматриваем. Кроме того, имеются драйверы для последовательного интерфейса RS-232, интерфейса SCSI, приводов CD-ROM, адаптера Ethernet и звуковой карты. Они могут быть задействованы при перекомпиляции MINIX.

Все эти задачи взаимодействуют с другими частями MINIX одинаковым образом: путем отправки сообщений. Сообщения с запросами содержат различные поля, в которые помещаются код операции (например, чтение или запись) и ее параметры. Получив сообщение, задача пытается выполнить запрос и отправляет ответное сообщение.

Поля запроса и ответа для блочного устройства приведены в табл. 3.4. Сообщение с запросом содержит адрес буфера для обмена данными. Ответ включает в себя информацию о состоянии, которую вызвавший процесс может использовать для проверки правильности выполнения запроса. Для символьных устройств поля сообщений могут несколько отличаться от задачи к задаче, но в целом имеют сходную структуру. Например, сообщения для задачи таймера содержат значения времени, а сообщения для задачи терминала — адрес структуры данных, описывающей все возможные параметры, такие как символы, используемые функциями строчного редактирования.

Назначение задач — воспринимать и выполнять запросы от других процессов (в норме — от файловой системы) и выполнять их. Для блочных устройств все задачи написаны так, чтобы получить запрос, выполнить его и послать ответ. Такая структура означает простой, линейный алгоритм задач. Когда происходит аппаратное прерывание, задача вызывает операцию receive, чтобы указать, что

далее она ожидает только сообщения от прерываний. Новые запросы при этом будут отложены до тех пор, пока текущий не будет обслужен. Задача терминала работает несколько по-другому, так как в этом случае одна задача обслуживает несколько устройств. Поэтому она может воспринимать задание на ввод с клавиатуры и тогда, когда еще не завершено чтение с последовательного интерфейса. Тем не менее для каждого отдельного устройства новая задача начинает выполняться только после завершения старой.

Таблица 3.4. Поля запросов от файловой системы к драйверам блочных устройств и ответов, отправляемых файловой системе

Поле	Тип	Значение
Запрос		
m.m_type	int	Запрошенное действие
m.DEVICE	int	Младший номер устройства
m.PROC_ADR	int	Процесс, запрашивающий ввод/вывод
m.COUNT	int	Количество байт или код ioctl
m.POSITION	long	Позиция устройства
m.ADDRESS	char*	Адрес в пределах процесса, сделавшего запрос
Ответ		
m.m_type	int	Всегда имеет значение TASK_REPLY
m.REP_PROC_NR	int	То же, что и PROC_NR в запросе
m.REP_STATUS	int	Количество переданных байтов или код ошибки

Структура основной части кода у каждой задачи одинакова и в общих чертах приведена в листинге 3.1. При запуске системы управление передается каждому из драйверов, с целью дать им возможность инициализировать внутренние таблицы и тому подобные данные. Завершив инициализацию, каждая из задач делает вызов receive, чтобы получить сообщение, и переходит в состояние блокировки. Когда сообщение приходит, запоминается, какой процесс его отправил, после чего вызывается подпрограмма для выполнения необходимых действий. После того как обработка запроса завершена, вызвавшему процессу отправляется сообщение с результатами, а задача переходит к началу цикла и ожидает новых сообщений.

За выполнение различных действий, поддерживаемых драйвером, ответственны подпрограммы с именами dev_xxx. Код завершения помещается в поле REP_STATUS ответа, он равен количеству переданных байтов (ноль или положительное число), если операция закончилась успехом, или номер ошибки (отрицательное число) в противном случае. Количество переданных байтов может отличаться от запрошенного, например, когда достигнут конец файла. В случае терминалов за один запрос возвращается не больше одной строки, даже если запрошено большее количество информации.

Листинг 3.1. Общая структура программы задач ввода/вывода

```
message mess;                                   /* Буфер сообщений */

void io_task() {
  initialize();                                 /* Вызывается только один раз */
```

```
  while (TRUE) {
    receive(ANY, &mess);                    /* Ожидаем запрос */
    caller = mess.source;                   /* Процесс, сделавший запрос */
    switch(mess.type) {
      case READ:
         rcode = dev_read(&mess); break;
      case WRITE:
         rcode = dev_write(&mess); break;
      /* Тут находится код для выполнения других действий,
         таких как OPEN, CLOSE и IOCTL */
      default:
         rcode = ERROR;
    }

    mess.type = TASK_REPLY;
    mess.status = rcode;                    /* Код возврата */
    send(caller, &mess);                    /* Отправляем ответ */
  }
}
```

3.4.3. Аппаратно-независимый код ввода/вывода в MINIX

Весь аппаратно-независимый код ввода/вывода в MINIX является частью файловой системы. Система ввода/вывода так тесно связана с файловой системой, что они обе объединены в один процесс. Функции файловой системы перечислены в табл. 3.2, за исключением функций захвата и освобождения отдельных устройств, которых в существующей конфигурации MINIX нет. Тем не менее, если такая потребность возникнет, эти функции могут быть легко добавлены в код соответствующих драйверов.

В дополнение к взаимодействию с драйверами, к буферизации и выделению блоков, файловая система также решает задачи защиты и управления *i*-узлами, каталогами и монтированными файловыми системами. Подробно эта тема будет рассмотрена в главе 5.

3.4.4. Программы ввода/вывода пользовательского уровня в MINIX

Здесь также применяется обобщенная модель, описанная ранее в этой главе. Для выполнения системных вызовов и стандартных функций C, требуемых стандартом POSIX (например, функций форматного ввода/вывода printf и scanf), предоставляются библиотечные процедуры. В стандартной конфигурации MINIX имеется также демон lpd, который занимается поддержкой очереди печати документов, отправленных на печать командой lp. В стандартной комплектации MINIX имеется еще ряд демонов, ответственных за различные сетевые функции. Работа этих демонов требует поддержки сетевых функций в ядре, но они могут быть легко задействованы при перекомпиляции. Для этого необходимо включить в систему сетевой сервер, работающий с тем же уровнем привилегий, что и менед-

жер памяти или файловая система, и, как и другие серверы, в режиме пользовательского процесса.

3.4.5. Тупики в MINIX

Как UNIX, так и MINIX относятся к тупикам одинаково, то есть полностью игнорируют проблему. В MINIX нет выделенных устройств ввода/вывода, хотя, если кто-то хочет «подвесить» стандартный промышленный ленточный привод DAT на персональном компьютере, это не составит особенного труда. Говоря коротко, единственное место, в котором ожидаемы взаимоблокировки, это неявные совместно используемые ресурсы, такие как ячейки таблицы процессов, ячейки таблицы *i*-узлов и т. д. Ни один из известных алгоритмов борьбы с тупиковыми ситуациями не подходит для подобных ресурсов, которые явно не запрашиваются.

В действительности, сказанное выше — не совсем правда. Риск того, что пользовательский процесс завернет в тупик, это одно, но в самой операционной системе есть несколько мест, где необходимо принимать значительные меры предосторожности, чтобы избежать проблем. Главное — это взаимодействие между файловой системой и менеджером памяти. Обрабатывая системный вызов exec, менеджер памяти посылает файловой системе запросы на чтение двоичных файлов (исполняемого кода программы). При этом, если файловая система не находится в бездействии, менеджер памяти блокируется, ожидая ее освобождения. Если файловая система в то же время попытается отправить сообщение менеджеру памяти, она также заблокируется. Итог — тупик.

Эта опасность устраняется следующим образом: файловая система никогда *не запрашивает* сообщения от менеджера памяти, а только *отвечает* на них, за одним небольшим исключением. Исключение относится к запуску системы, когда менеджер памяти сообщает файловой системе размер доступной оперативной памяти. Файловая система в этот момент гарантированно будет ожидать сообщения.

Разграничивать доступ к устройствам можно и без поддержки операционной системы. В качестве по-настоящему глобальной переменной, доступной всем процессам, изумительно подходит имя файла. Наличие или отсутствие определенного файла может быть легко замечено любым процессом. В MINIX, как и в UNIX-системах, существует специальный каталог, /usr/spool/locks/, где процессы могут создавать подобные *файловые семафоры* (lock file), чтобы обозначить занятость определенного ресурса. Файловая система MINIX также поддерживает рекомендательную блокировку файлов в стиле POSIX. Но ни один из этих механизмов не является принудительным. Все зависит от того, обращает ли программа внимание на эти механизмы или нет, и не существует способа запретить программе использовать ресурс, уже занятый другой программой. Это не то же самое, что и выгрузка (preemption) ресурса, так как ничто не мешает первому процессу делать повторные попытки его использования. Другими словами, не обеспечивается взаимное исключение. В результате процесс, неправильно работающий с блокировкой, может получить неправильные результаты, но тупика не возникнет.

3.5. Блочные устройства в MINIX

В последующих разделах мы вернемся к теме главы, драйверам устройств, и подробно изучим некоторые из них. В MINIX поддерживается несколько типов блочных устройств, поэтому мы начнем с обсуждения общих аспектов. Затем мы приступим к изучению RAM-диска, жесткого диска и флоппи-дисковода. Каждое из этих устройств по-своему интересно. RAM-диск необычен тем, что это устройство обладает всеми атрибутами обычного блочного устройства, за исключением того, что при его работе никакого реального ввода/вывода не происходит. Этот «диск» целиком находится в оперативной памяти. Поэтому драйвер устроен просто и является хорошей отправной точкой для изучения. Жесткий диск демонстрирует, на что похож драйвер реального диска. Некоторым может показаться, что дискетный привод (накопитель на гибких дисках) проще, чем жесткий диск, но это не так. Мы не будем обсуждать драйвер гибкого диска в подробностях, но укажем на несколько имеющихся в нем сложных мест.

После раздела, посвященного драйверам блочных устройств, мы обсудим другие классы устройств в MINIX. Таймер важен тем, что он присутствует в каждой системе и полностью отличается от всех остальных устройств. Это также любопытный пример устройства, которое не является ни блочным, ни символьным. В завершение мы рассмотрим драйвер терминала, который важен для любой системы и, к тому же, является удачным примером драйвера символьного устройства.

Каждый из разделов включает в себя описание соответствующего оборудования, принципов построения программы драйвера, обзор реализации и сам код. Такая организация делает информацию полезной и для тех, кто не заинтересован в деталях.

3.5.1. Обзор драйверов блочных устройств в MINIX

Ранее мы упомянули, что основной код у большинства задач ввода/вывода имеет общую структуру. У MINIX в ядре всегда в наличии три встроенных задачи ввода/вывода: для RAM-диска, флоппи-дисковода и один из нескольких драйверов жесткого диска. Кроме того, при необходимости в ядро нетрудно добавить драйверы компакт-дисков или SCSI. Несмотря на то что все драйверы работают как отдельные процессы, они разделяют одно адресное пространство и поэтому могут использовать общий код, например различные вспомогательные подпрограммы.

Конечно, каждому драйверу необходимо выполнить некоторые действия для инициализации. Драйвер RAM-диска должен зарезервировать необходимый объем памяти, драйвер жесткого диска — определить параметры устройства и т. д. Все драйверы дисков вызываются, инициализируются и, завершив все необходимые действия, входят в бесконечный цикл. Этот цикл никогда не завершается, в нем нет команды выхода. В цикле драйвер получает сообщение, вызывается одна из функций обработки и генерируется ответное сообщение.

Основной цикл, управляющий работой каждой из задач в ядре, не просто является копией некой библиотечной функции, встроенной в код. В исполняемом коде MINIX имеется только один такой цикл. Каждый драйвер передает в главный цикл параметр, представляющий собой указатель на таблицу адресов функций, которые драйвер будет использовать для обработки запросов. Далее функции вызываются из тела цикла при помощи косвенного вызова:

```
code = (*entry_points->dev-read)(&mess);
```

Пример такого кода можно увидеть в листинге 3.2, демонстрирующем структуру главного цикла. У каждого из драйверов при этом вызывается своя функция dev_read, хотя главный цикл един для всех драйверов. Но некоторые операции, например CLOSE, достаточно просты, чтобы для всех устройств можно было использовать одну и ту же функцию.

Листинг 3.2. Общая подпрограмма главного цикла для задач ввода/вывода

```
message mess;                                   /* Буфер сообщения */

void shared_io_task(struct driver_table *entry_points) {
/* Перед входом в эту функцию каждая из задач
  инициализируется */
  while (TRUE) {
    receive (ANY, &mess);
    caller = mess.source;
    switch(mess.type) {
      case READ:
        rcode = (*entry_points->dev_read)(&mess); break;
      case WRITE:
        rcode = (*entry_points->dev_write)(&mess); break;
      /* Тут находится код для выполнения других действий,
         таких как OPEN, CLOSE и IOCTL */
      default:
         rcode = ERROR;
    }
    mess.type = TASK_REPLY;
    mess.status = rcode;                        /* Код возврата */
    send(caller, &mess);                        /* Отправляем ответ */
  }
}
```

Сведение главного цикла к единственной копии — хорошая иллюстрация концепции процесса, представленной в главе 1 и обсуждаемой на протяжении главы 2. В памяти находится только один код главного цикла для всех драйверов блочных устройств, но этот код исполняется как главный цикл трех или более процессов. При этом каждый из процессов работает независимо, и каждое действие производится со своим набором данных и стеком.

Есть шесть различных действий, которые могут быть запрошены у каждого драйвера устройства. Им соответствуют значения, перечисленные для поля m.m_type в табл. 3.4. Вот они: OPEN, CLOSE, READ, WRITE, IOCTL, SCATTERED_IO.

Большая часть этих операций, вероятно, знакома читателям, имеющим опыт программирования. На уровне драйвера устройства они сводятся к системным вызовам с соответствующим именем. Смысл операций READ и WRITE, например,

вполне понятен. Эти операции передают блок данных из памяти вызвавшего процесса в устройство или наоборот. Обычно операция WRITE не передает управление в вызвавший процесс до тех пор, пока передача данных не завершена. Но операционная система может буферизовать передаваемые данные, и вызов может завершиться сразу, а данные будут переданы позже. Это никак не сказывается на сделавшей вызов программе, сразу после него она вправе использовать буфер, так как операционная система уже скопировала нужные ей данные. Операции OPEN и CLOSE для устройства означают то же, что и системные вызовы open и close для файлов. Операция OPEN проверяет, что устройство доступно, и возвращает код ошибки, если это не так. Операция CLOSE должна гарантировать, что все буферизованные данные переданы до полного завершения обмена информацией с устройством.

Возможно, назначение операции IOCTL не столь очевидно. У многих устройств ввода/вывода есть параметры работы, значение которых иногда нужно определять и, предположительно, менять. Выполнение этих действий обеспечивает операция IOCTL. Хороший пример использования IOCTL — управление скоростью последовательного интерфейса или параметров контроля четности. Для блочных устройств IOCTL применяется реже. В частности, в MINIX при помощи IOCTL можно управлять тем, как диск разбит на разделы (хотя это делается просто при помощи операций чтения и записи).

Операция SCATTERED_IO, без сомнений, известна менее всех других. Дело в том, что трудно достигнуть приемлемой производительности блочного устройства, если запрашивать данные последовательно, отдельными порциями. Исключение составляют лишь исключительно быстрые устройства, такие как RAM-диск. Запрос SCATTERED_IO позволяет системе делать запросы на чтение или запись нескольких блоков. В случае операции READ система пытается предугадать, какие блоки может запросить процесс, и считывает их заранее. В таком запросе не все затребованные данные обязательны. Каждый из запросов блока данных, передаваемых драйверу устройства, можно модифицировать, установив бит, сообщающий, что запрос необязателен. В результате файловая система может создать запрос, который в переводе на человеческий язык звучал бы так: «Было бы неплохо прочитать все эти данные, но все они вовсе не нужны мне прямо сейчас». Далее устройство само решает, что лучше делать. Например, гибкий диск может прочитать все данные в пределах одной дорожки, говоря тем самым: «Эти данные я для тебя прочитаю, а остальные попроси потом, мне слишком долго переходить на другую дорожку».

Когда необходимо записать данные, не возникает вопроса, обязательно ли записывать каждый отдельный блок. Но система вправе буферизовать несколько запросов на запись в надежде на то, что сразу их удастся обработать быстрее, чем по отдельности. При запросе SCATTERED_IO, будь то для чтения или для записи, список запрошенных блоков сортируется, что позволяет считывать данные более эффективно, чем если бы они считывались в порядке поступления. Кроме того, передача нескольких блоков за один раз уменьшает количество сообщений, пересылаемых внутри MINIX.

3.5.2. Общие программы драйверов блочных устройств

Определения, присущие всем драйверам блочных устройств, находятся в файле driver.h. Самая главная часть этого файла — структура driver, посредством которой в главный цикл передается список адресов функций, выполняющих запросы. Дополнительно здесь определяется структура device, в которой хранится наиболее важная информация о разделах, базовый адрес и размер в байтах. Формат этой структуры был выбран так, чтобы основанные на оперативной памяти устройства могли использовать данные без преобразования, обеспечивая высокую скорость отклика. У реальных дисков существует такое большое количество разнообразных, влияющих на время отклика факторов, что преобразование байтовых адресов в секторы не играет принципиальной роли.

Главный цикл и общие функции всех драйверов блочных устройств находятся в файле driver.c. Выполнив необходимые действия по инициализации, каждый драйвер вызывает процедуру driver_task, передавая в качестве аргумента вызова заполненную структуру driver. Далее определяется адрес буфера DMA-устройства и управление переходит в главный цикл, откуда обратной дороги, то есть команды возврата, нет.

Единственный процесс, который должен отправлять сообщения драйверам, — это файловая система. Координация производится оператором switch в теле главного цикла. Оператор switch игнорирует остающиеся аппаратные прерывания, а сообщения от других отправителей вызывают вывод на экран предупреждения. На первый взгляд это кажется достаточно безвредным, но процессы, которые отправили сообщение неверному адресату, вероятно, навсегда останутся в состоянии блокировки, ожидая ответа. Второй оператор switch обрабатывает запросы. Первые три типа сообщений, DEV_OPEN, DEV_CLOSE и DEV_IOCTL, приводят к косвенному вызову одной из функций, адреса которых передаются в структуре driver. Сообщения DEV_READ, DEV_WRITE и SCATTERED_IO передают управление в do_rdwt или do_vrdwt. Заметьте, что в любые вызовы, косвенные или прямые, передается структура driver, поэтому все вызываемые подпрограммы могут к ней обращаться.

После выполнения запрошенных в сообщении действий может потребоваться определенная подготовка перед переходом к следующей операции. Например, в случае гибкого диска можно запустить таймер, который по истечении некоторого времени отключит мотор привода, если за это время не будет других запросов. Для выполнения подобных действий также используется косвенный вызов. После этого вызова вырабатывается сообщение, которое отправляется в ответ процессу, сделавшему вызов.

Первое действие, которое выполняет каждая задача при входе в главный цикл, — это вызов функции init_buffer, назначающий буфер для DMA. Все задачи пользуются одним и тем же буфером, если пользуются им вообще, так как некоторые из задач могут не поддерживать DMA. Поэтому инициализация буфера в каждой из задач избыточна, но никакого вреда она не приносит. Писать код, который предварительно бы проверял необходимость инициализации, более неуклюже.

Причина, по которой эта инициализация вообще нужна, кроется в странностях работы оригинальных IBM PC, для которых было необходимо, чтобы буфер не пересекал границу в 64 Кбайт. Другими словами, если размер буфера 1 Кбайт, он может начинаться по адресу 64510, но не по адресу 64514, так как в последнем случае будет превышена граница, находящаяся по адресу 65536.

Это неприятное требование проистекает из того, что в IBM PC применялся старый чип контроллера прерываний Intel 8237A с 16-битным счетчиком. Так как для DMA используются абсолютные, а не относительные сегментные адреса, требовался счетчик большей емкости. На старых машинах, способных адресовать только 1 Мбайт памяти, младшие 16 бит адреса DMA загружались в 8237A, а оставшиеся старшие 4 бита помещались в 4-разрядный регистр-переключатель. У более современных машин применяется 8-разрядный переключатель, что позволяет адресовать 16 Мбайт. Но когда адрес в 8237A переходит от 0xFFFF к 0x0000, не формируется признак переноса, и адрес DMA внезапно прыгает вниз на 64 Кбайт.

Переносимая программа на языке C не может оперировать абсолютными адресами, поэтому не в силах предотвратить недопустимого положения буфера. Эта проблема решается так: выделяется буфер вдвое большего размера, чем нужно, а специальный указатель tmp_buf содержит адрес реально используемого буфера. Функция init_buffer сначала пробует установить этот указатель так, чтобы он ссылался на начало выделенного буфера, затем проводит проверку, допустимо ли такое положение. Если нет, то значение указателя tmp_buf увеличивается на количество реально затребованных байтов. Таким образом, всегда остается неиспользованный блок памяти, но граница 64 Кбайт никогда не попадает внутрь буфера.

У новых компьютеров семейства IBM PC более совершенные контроллеры DMA, поэтому код мог бы быть упрощен, а количество запрашиваемой памяти — уменьшено. Но нет гарантии, что любая машина будет нечувствительна к данной проблеме. Тем не менее, если вы рассчитываете на это, вы должны знать, как ошибка проявит себя, если она все-таки есть. Если требуется буфер объемом 1 Кбайт, то с вероятностью 1/64 выделенный буфер будет непригоден для компьютера со старым контроллером DMA. Каждый раз, когда код ядра меняется так, что изменяется размер его исполняемых файлов, ошибка может проявить себя с той же вероятностью. Ошибка может не проявлять себя, поэтому, когда в следующем месяце или году она возникнет, ее свяжут с последними изменениями в коде, а не с реальной причиной. Подобные «особенности» аппаратного обеспечения опасны затратой недель на поиск ошибок, особенно если в технической документации не говорится о них ни слова (как в этом случае).

Следующая функция в dirver.c — это do_rdwt. Она, в свою очередь, косвенным вызовом запускает одну из трех функций dr_prepare, dr_schedule или dr_finish, опираясь на переданные адреса из структуры driver. В дальнейшем мы будем использовать запись вида *function_pointer, чтобы обозначить функцию, вызываемую через указатель.

Далее проверяется, что количество переданных байтов больше нуля, после чего do_rdwt вызывает *dr_prepare. Этот вызов не должен «провалиться», так как

единственная причина, по которой он может не выполниться, — неправильно указанное в операции OPEN устройство. Далее заполняется стандартная структура iorequest_s, описание которой находится в файле include/minix/type.h. Затем делается следующий косвенный вызов, *dr_schedule. В разделе ниже будет показано, что обработка запросов к оборудованию в том порядке, в каком они поступают, не обязательно эффективна. Эта подпрограмма служит для того, чтобы запросы к каждому конкретному устройству могли обрабатываться в оптимальном порядке. Например, для RAM-диска dr_schedule ссылается на функцию, которая непосредственно вызывает передачу данных, а следующий косвенный вызов, *dr_finish, представляет собой пустую функцию. У реальных дисков *dr_finish выполняет все задержанные запросы на считывание, поступившие с предшествующими вызовами *dr_schedule. Как мы увидим, иногда *dr_finish может вообще не передавать всех запрошенных данных.

Какой бы вызов ни занимался реальным обменом информацией, изменяется значение поля io_nbytes в структуре iorequest_s. При этом отрицательное число индицирует ошибку при передаче данных, а положительное означает разницу между запрошенным количеством байтов и количеством, которое было реально передано. Причем нулевое значение не обязательно свидетельствует об ошибке. Например, это может значить, что достигнут конец устройства. После возврата в главный цикл отрицательное значение (код ошибки) помещается в поле REP_STATUS, если ошибка произошла. Если ошибки не было, полученное значение вычитается из запрошенного количества байтов (поле COUNT исходного сообщения) и результата (равный реально переданному объему информации) и записывается в поле ответного сообщения REP_STATUS.

Следующая функция, do_vrdwt, обрабатывает разрозненные запросы ввода/вывода. В сообщении драйверу, содержащем подобный запрос, в поле ADDRESS помещается адрес массива структур типа iorequest_s. Каждая из этих структур хранит информацию для выполнения одного запроса: адрес буфера, величину смещения для устройства, количество байтов, а также признак, нужно ли чтение или запись. Все операции в одном запросе должны быть либо на чтение, либо на запись, и перед выполнением они сортируются в порядке следования блоков на устройстве. Функция do_vrdwt сложнее функции для чтения или записи, так как массив запросов должен быть предварительно скопирован в память ядра. Но, после того как массив скопирован, запросы выполняются при помощи тех же самых косвенных вызовов *dr_prepare, *dr_schedule и *dr_finish. Различие только в том, что второй вызов, *dr_schedule, делается несколько раз в цикле, по одному разу для каждого запроса, или пока не случится ошибка. После завершения цикла однократно вызывается *dr_finish, и затем массив запросов копируется обратно на то место, откуда он взят. При этом в каждом из элементов массива меняется поле io_nbytes, отражая количество реально переданных байтов. Суммарное количество для всех запросов не подсчитывается, хотя вызывающая программа сама может рассчитать это значение по полям массива.

Как уже было сказано выше, при выполнении разрозненного запроса не все переданные в вызов *dr_schedule гарантированно выполняются, когда делается завершающий вызов *dr_finish. Поле io_request в структуре iorequest_s содер-

жит фланговый бит, который позволяет пометить данный запрос как необязательный.

Следующие несколько функций из файла driver.c обеспечивают выполнение описанных выше операций. Вызов *dr_name возвращает имя устройства. Если у устройства нет фиксированного имени, функция no_name извлекает имя устройства из таблицы задач. Некоторым устройствам не нужен ряд функций. Например, RAM-диск не должен ничего делать по запросу DEV_CLOSE. В качестве пустой функции используется do_nop, возвращаемое ею значение зависит от типа запроса. Функции nop_finish и nop_cleanup — пустые функции для устройств, которым не требуется обрабатывать вызовы *dr_finish или *dr_cleanup.

Иным дисковым функциям может оказаться необходимой функция задержки, например, при ожидании, когда мотор дисковода наберет скорость. Поэтому в driver.c помещена следующая функция, clock_mess, при помощи которой отправляются сообщения задаче таймера. Функции передается количество отсчетов («тиков») таймера и адрес функции, которая будет вызвана по истечении указанного интервала.

Наконец, функция do_diocntl осуществляет для блочных устройств запросы DEV_IOCTL. Она сообщает об ошибке, если затребована любая другая операция помимо считывания (DIOGETP) информации о разделе или записи (DIOSETP). Чтобы удостовериться, что устройство задано корректно, и получить указатель на структуру device с информацией о базовом адресе и размере раздела, do_dioctl вызывает *dr_prepare. При обработке запроса на чтение вызывается специфичная для устройства функция *dr_geometry, с целью получить информацию о цилиндрах, головках и секторах, относящихся к разделу диска.

3.5.3. Библиотека поддержки драйверов

Файлы drvlib.h и drvlib.c содержат системно-зависимый код, поддерживающий работу с разделами дисков на IBM PC-совместимых компьютерах.

Разбиение диска на разделы позволяет получить на одном накопителе несколько логических. Обычно на разделы дробятся жесткие диски, хотя MINIX поддерживает и разбиение дискет. Вот несколько причин в пользу разбиения на разделы.

1. Стоимость мегабайта меньше для больших дисков. Если используются две или более операционные системы с различными файловыми системами, дешевле пользоваться одним большим диском, поделенным на несколько частей, чем несколькими маленькими.
2. У операционной системы могут иметься ограничения на максимальный размер устройства. Обсуждаемая здесь версия MINIX поддерживает максимальный размер файловой системы 1 Гбайт, а у более старых версий было ограничение до 256 Мбайт. В результате оставшееся дисковое пространство расходовалось впустую.

3. Операционная система может использовать две или более файловые системы. Например, для обыкновенных файлов — стандартная файловая система, а для области подкачки — система с совершенно другой структурой.
4. Бывает удобным поместить часть операционной системы на отдельное устройство. Например, для упрощения создания резервных копий можно записать корневую файловую систему MINIX на отдельный маленький раздел. Это также позволяет скопировать ее на RAM-диск во время загрузки.

Поддержка разделов зависит от платформы, причем эта зависимость не относится к аппаратному обеспечению. К накопителям поддержка разделов не привязана. Но если на одном и том же наборе устройств должно работать несколько операционных систем, они должны иметь одинаковое представление о формате таблицы разделов. Для IBM PC стандарт задает команда MS-DOS fdisk, и другие операционные системы, такие как MINIX, OS/2 и Linux, поддерживают этот формат и могут сосуществовать с MS-DOS. Когда MINIX переносится на другую платформу, имеет смысл использовать формат таблицы разделов, совместимый с другими операционными системами, работающими на данной платформе. Чтобы упростить перенос системы, код для поддержки таблицы разделов на IBM PC-совместимых системах помещен в отдельный файл drvlib.c, а не включен в driver.c.

Основные структуры данных описываются в файле include/minix/partition.h. Формат этих структур определяется разработчиками программно-аппаратных средств платформы. Этот заголовочный файл при помощи директивы препроцессора #include включается в файл drvlib.h. Сюда входит информация о геометрии всех разделов (цилиндры-головки-секторы), а также информация о типе файловой системы на каждом из разделов и метка, обозначающая активный раздел, который будет загрузочным. После проверки файловых систем большая часть этой информации MINIX не нужна.

Функция partition в файле drvlib.h вызывается, когда в первый раз открывается блочное устройство. В аргументы этой функции входит структура driver, поэтому первая в состоянии вызывать специфичные для данного устройства функции. Также в функцию передается младший номер устройства и параметр, задающий тип разбиения на разделы (существуют различные способы разбиения для гибкого диска, основного раздела и подраздела). Чтобы проверить правильность устройства и поместить в структуру device базовый адрес и размер раздела, вызывается функция *dr_prepare. Затем, чтобы детектировать наличие таблицы разделов и считать ее, вызывается get_part_table. Если таблицы разделов не оказалось, работа завершена. В противном случае вычисляется младший номер устройства для первого раздела (правила нумерации устройств определяются упомянутым ранее типом разбиения на разделы). Для случая основных разделов таблица сортируется так, чтобы порядок разделов соответствовал очередности, используемой другими операционными системами.

В этом месте делается другой вызов *dr_prepare, на этот раз ему в качестве параметров передается только что вычисленный для первого раздела номер устройства. Если устройство корректно, проверяются все записи в таблице на этом устройстве, с целью удостовериться, что хранящиеся в таблице ссылки не выходят за пределы, отведенные устройству. Если все данные корректны, таблица

в памяти соответствующим образом изменяется. Может показаться, что такие проверки сродни маниакальной идее, но таблицы разделов могут создаваться различными операционными системами. Программист, работавший с другой системой, мог попытаться каким-то необычным образом использовать таблицу разделов, или же таблица просто испорчена по той или иной причине. Мы с большим доверием относимся к значениям, вычисляемым в MINIX. Лучше подстраховаться, чем сожалеть.

В том же самом цикле, для всех разделов, идентифицированных как разделы MINIX, рекурсивно вызывается функция partition, чтобы получить информацию о подразделах. Если раздел идентифицирован как расширенный (extended partition), вместо нее вызывается функция extpartition.

Последняя в действительности не имеет никакого отношения к операционной системе MINIX. Расширенные разделы ведут происхождение от MS-DOS, как один из вариантов создания подразделов. Поэтому, чтобы обеспечить в MINIX работу с файлами MS-DOS, системе необходимо знать о дополнительных возможностях.

Следующая функция, get_part_table, получает тот сектор устройства (или вложенного устройства), в котором расположена таблица разделов, при помощи вызова do_rdwt. Передаваемое функции смещение должно быть равно 0, если функция вызывается для основного раздела, а для подразделов смещение принимает отличное от нуля значение. Функция проверяет наличие сигнатуры (0xAA55) и возвращает результат проверки (истина или ложь). Кроме того, если таблица найдена, функция копирует таблицу по адресу, переданному ей через другой аргумент.

Наконец, функция sort сортирует по нижнему сектору все записи в таблице разделов. Записи, помеченные как не имеющие раздела, исключаются из сортировки и идут последними, даже если в поле, по которому велась сортировка, у них записано нулевое значение. Для сортировки применяется простой пузырьковый алгоритм. Более сложные способы для сортировки таблицы из четырех записей ни к чему.

3.6. RAM-диски

Теперь мы вернемся к изучению отдельных блочных устройств и подробно рассмотрим некоторые из них. Сначала мы обратимся к драйверу RAM-диска. Его типовая область применения — доступ к произвольной области памяти. Обычно часть памяти резервируется и используется как обычный диск. Такой диск не обеспечивает долговременного хранения данных, но зато не изнашивается и работает очень быстро.

В таких системах, как MINIX, которым достаточно только гибкого диска, RAM-диск дает еще одно преимущество. Если поместить корневую файловую систему на RAM-диск, то дисковод можно демонтировать, чтобы сменить дискету. Если же поместить корневое устройство на флоппи-дисковод, пользоваться им станет невозможно, так как дискету нельзя будет демонтировать. В допол-

нение к сказанному, размещение корневого устройства на RAM-диске придает системе большую гибкость: она приобретает способность работать с любой комбинацией дисководов и жестких дисков. Кроме того, хотя у большинства современных компьютеров есть жесткий диск (за исключением некоторых встроенных систем), RAM-диск может оказаться полезным во время установки, когда жесткий диск еще не готов для работы MINIX, или если необходимо временно запустить систему без полной установки.

3.6.1. Аппаратное и программное обеспечение RAM-диска

Идея устройства RAM-диска проста. Любое блочное устройство — это накопитель с двумя командами: прочитать блок и записать блок. Обычно эти блоки находятся на вращающихся дисках, таких как дискеты или пластины жестких дисков. RAM-диск проще. Он хранит данные в предварительно выделенной области оперативной памяти. Преимущество такого диска в том, что он обеспечивает высокую скорость доступа (так как не требуется перемещать головки и вращать носитель) и может быть использован для хранения данных, к которым часто совершаются обращения.

Отступая в сторону, нужно вкратце рассказать о различии между операционными системами, поддерживающими монтируемые файловые системы, и системами без такой поддержки (как MS-DOS и Windows). Когда файловые системы монтируются, корневая файловая система всегда находится в фиксированном месте, а прочие (например, диски) встраиваются в указанное место в дереве файлов, образуя единую файловую структуру. Смонтировав некоторую файловую систему, пользователь может больше не задумываться, на каком устройстве она расположена.

В противовес такому подходу, в других операционных системах пользователь должен указывать положение каждого файла либо явно, например, как `B:\DIR\FILE`, либо пользуясь различными умолчаниями (текущее устройство, текущий каталог и т. д.). На маленьких системах, с одним или двумя дисками, это можно вытерпеть, но на мощных компьютерах с дюжинами устройств следить все время за всеми устройствами невыносимо. Заметьте, что UNIX работает на компьютерах от IBM PC до рабочих станций и суперкомпьютеров, а MS-DOS используется только на малых системах.

На рис. 3.12 показано устройство RAM-диска. Диск разбивается на n блоков, в зависимости от того, сколько памяти для него выделено. Размер каждого блока равен размеру блока, используемому на реальных дисках. Когда драйвер получает запрос на чтение или запись блока, он просто вычисляет адрес и производит чтение или запись по этому адресу, вместо того чтобы работать с дискетой или жестким диском. Обмен данными осуществляется при помощи ассемблерной подпрограммы, копирующей данные с максимальной возможной скоростью.

Драйвер RAM-диска может поддерживать несколько областей памяти, которые различаются присвоенным им младшим номером устройства. Обычно эти

области различны, но иногда бывает удобно, чтобы они перекрывались, как мы увидим в следующем разделе.

Рис. 3.12. RAM-диск

3.6.2. Обзор драйвера RAM-диска в MINIX

Рассматриваемый нами драйвер в действительности представляет собой четыре тесно связанных драйвера, объединенные в одном. Каждое сообщение для этого драйвера указывает одно из следующих устройств:

```
0: /dev/ram   1  /dev/mem   2: /dev/kmem   3: devnull
```

Первый из специальных файлов, /dev/ram, является настоящим RAM-диском. Ни размер, ни положение занимаемой им области памяти в драйвере не прописаны. Они определяются файловой системой при загрузке MINIX. По умолчанию размер RAM-диска равен размеру образа корневой файловой системы, чтобы можно было скопировать на него корневую систему. Варьирование параметрами загрузки позволяет установить больший размер RAM-диска, или если не копировать на него образ корневой системы, то можно назначить диску произвольный размер, который и в памяти умещается, и оставляет достаточно места для работы MINIX. Определив размер диска, система выделяет в памяти область достаточно большого размера, до того как начнет работу менеджер памяти. Такая стратегия позволяет увеличивать или уменьшать размер RAM-диска даже без перекомпиляции системы.

Следующие два устройства применяются для доступа к оперативной памяти или к памяти ядра соответственно. Открыв устройство /dev/mem и считывая из него данные, можно обратиться к любой области памяти, начиная с абсолютного нулевого адреса. Обычные пользовательские программы не имеют такой потребности, но системным программам, служащим для отладки операционной систе-

мы, это может оказаться полезным. Записывая данные в /dev/mem, можно изменить векторы прерываний. Не стоит и говорить, что делать это должен только опытный пользователь, твердо знающий, что он делает.

Файл /dev/kmem полностью подобен файлу /dev/mem, за тем исключением, что нулевой адрес в файле соответствует нулевому адресу в пространстве ядра, положение последнего зависит от размера кода MINIX. Этот файл также привлекается по большей части для целей отладки рядом специальных программ. Обратите внимание на то, что области памяти, соответствующие этим двум устройствам, перекрываются. Если вы точно знаете, где в памяти находится ядро, то, если прочитать из /dev/mem данные по означенному адресу, можно увидеть точно те же данные, которые находятся в начале файла /dev/kmem. При перекомпиляции системы размер и положение ядра в памяти могут измениться, но /dev/kmem все равно будет содержать область памяти ядра.

Последний файл в группе, /dev/null, служит специально для того, чтобы выбрасывать ненужные данные. Он часто используется в программах оболочки, чтобы скрыть вывод программы, когда он не нужен. Например, команда

```
a out >/dev/null
```

запустит программу a.out, но все, что она выводит, будет игнорироваться. Драйвер RAM-диска считает, что размер этого устройства равен 0, поэтому никаких данных на него не записывается и не считывается с него.

Код, обслуживающий устройства /dev/ram, /dev/mem и /dev/kmem, идентичен. Единственное различие между этими тремя устройствами в том, что они работают с разными областями памяти, задаваемыми массивами ram_origin и ram_limit, индексируемыми младшим номером устройства.

3.6.3. Реализация драйвера RAM-диска в MINIX

Как и у других драйверов, главный цикл драйвера RAM-диска находится в файле driver.c. Поддержка специальных функций RAM-диска обеспечивается файлом memory.c. В массиве m_geom хранятся базовые адреса и размеры всех четырех специальных устройств. Адреса подпрограмм, обеспечивающих работу устройств, записываются в переменную m_dtab, имеющую тип driver. Эта структура будет использоваться в главном цикле для организации соответствующих устройству вызовов. Четыре вызова являются либо пустыми, либо очень простыми, их код помещен в driver.c, что является еще одним подтверждением того, что код драйвера не слишком сложен. Головная процедура mem_task выполняет локальную инициализацию, вызывая одну функцию. После этого она передает управление в главный цикл, который получает сообщения, вызывает для их обслуживания необходимые подпрограммы и отправляет ответные сообщения. Главный цикл не возвращает управление обратно в mem_task.

Выполняя операцию чтения или записи, главный цикл драйвера делает три вызова: один для подготовки устройства, один, чтобы запланировать операцию ввода/вывода, и еще один, чтобы завершить операцию. В данном случае первому вызову соответствует функция m_prepare. Она убеждается, что было выбрано

правильное младшее устройство, и возвращает адрес структуры, содержащей базовый адрес и размер запрошенной области памяти. Второй вызов обслуживает функция m_schedule, которая и выполняет всю работу. Имя этой функции не соответствует тому, что она делает. По определению, она должна запланировать операцию ввода/вывода, но для RAM-диска в планировании нет необходимости, так как никакого диска нет.

Работа RAM-диска настолько проста и выполняется так быстро, что никогда не возникает потребности отложить запрос, поэтому сначала функция сбрасывает бит, который при «разрозненном» вводе/выводе обозначает необязательное выполнение запроса. В сообщении, которое передается задаче, адрес буфера назначения указывается в адресном пространстве пользовательского процесса. Поэтому далее в коде функции этот адрес преобразуется в абсолютный и делается проверка его правильности. За сам обмен данными ответственны завершающие команды функции, обрамленные условным оператором. Передача данных сводится к простому копированию блока из одного места в другое.

Третий, завершающий шаг RAM-диску не нужен, поэтому в соответствующее поле переменной m_dtab записывается адрес пустой функции nop_finish.

За открытие RAM-диска отвечает функция m_do_open. Самые главные действия в ней выполняет вызов m_prepare, который проверяет, что устройство указано корректно. Если указано одно из устройств /dev/mem или /dev/kmem, то, чтобы получить более высокий уровень привилегий, делается вызов функции enable_iop (ее код находится в файле protect.c). Этот вызов нужен не для того, чтобы обратиться к памяти, он служит для решения другой задачи. Вспомните, что у процессоров Pentium есть четыре уровня привилегий. При этом пользовательские программы работают на последнем уровне. Кроме того, у процессоров Intel, в отличие от других архитектур, порты ввода/вывода занимают отдельное от основной памяти адресное пространство, и для обращения к ним применяется специальный набор команд. В обычной ситуации попытка пользовательского процесса обратиться к порту ввода/вывода приводит к исключению ограничения доступа. Тем не менее существуют причины, по которым может потребоваться разрешить пользовательским процессам доступ к портам, что в особенности относится к маленьким системам. Это и обеспечивает функция enable_iop, которая меняет биты уровня защиты ввода/вывода процессора (I/O Protection Level, IOPL) таким образом, чтобы необходимые действия были разрешены. В результате процесс, открывший устройство /dev/mem или /dev/kmem, в дополнение получает разрешение обращаться к портам ввода/вывода. В тех системах, где порты принадлежат общему адресному пространству, для доступа к ним достаточно установить для файлов памяти биты разрешения доступа rwx. Мы рассказали о таких возможностях потому, что, если их не учитывать, они могли бы стать уязвимостью. Если вы, например, планируете применять MINIX для управления системой безопасности банка, вам лучше перекомпилировать ядро без такой функциональности.

Следующая функция, m_init, вызывается только один раз, когда в первый раз делается вызов mem_task. Эта функция устанавливает базовый адрес и размер устройства /dev/kmem, присваивая размеру значение 1 Мбайт, 16 Мбайт или 4 Гбайт (n – 1), в зависимости от того, в каком режиме работает MINIX: 8088,

80286 или 80386. Величина устройства в данном случае определяется максимальным адресуемым объемом памяти и не имеет ничего общего с количеством оперативной памяти, установленной на машине.

RAM-диск поддерживает несколько операций ioctl, код для которых находится в функции m_ioctl. Операция MIOCRAMSIZE предоставляет файловой системе удобный способ управлять размером RAM-диска. Операция MIOCSPSINFO используется как файловой системой, так и менеджером памяти, чтобы поместить адреса своих компонентов в таблицу процессов в таблицу psinfo. Стандартная утилита UNIX ps получает эти данные при помощи операции MIOGSPSINFO. Программа ps — это стандартная утилита, но микроядерная архитектура MINIX затрудняет ее работу, так как необходимая ей информация о процессах хранится в нескольких различных местах. Системный вызов ioctl предоставляет удобный способ решить эту проблему. Без него пришлось бы при каждой перекомпиляции системы перекомпилировать ps.

Последняя функция в файле memory.c это m_geometry. Для RAM-диска понятия цилиндра, дорожки и сектора не имеют смысла, но, если система запросит эту информацию, он должен притвориться, что они есть.

3.7. Диски

Пример RAM-диска познавателен в качестве вступления (так как он прост), но у реальных дисков есть ряд особенностей, которые еще не были рассмотрены. В последующих разделах мы скажем несколько слов об устройстве жестких дисков, а затем перейдем к рассмотрению их драйверов и, в частности, драйверов для MINIX. Гибкие диски мы подробно обсуждать не будем, но выделим некоторые моменты, отличающие их от жестких дисков.

3.7.1. Аппаратная часть дисков

Магнитные диски организованы в цилиндры, каждый из которых содержит столько дорожек, сколько есть у устройства головок, установленных вертикально. Дорожки делятся на секторы, их количество обычно варьируется от 8 до 32 у гибких дисков и до нескольких сотен у жестких дисков. Число головок варьируется от 1 до 16.

У некоторых магнитных дисков мало электроники, они предоставляют на выходе простой поток битов. Контроллер такого диска выполняет совсем немного работы. На других дисках, в частности на *IDE-дисках* (IDE, Integrated Drive Electronics — встроенный интерфейс накопителей) само устройство содержит микроконтроллер, выполняющий значительный объем работ и позволяющий собственно контроллеру обращаться к нему с набором команд высокого уровня.

IDE-диски обладают одним свойством, несущим важные последствия для драйверов: контроллер способен выполнять одновременно поиск дорожки на двух и более дисках. Это свойство известно под названием *поиска с перекрытием*. В то время как контроллер и программное обеспечение ожидают окончания операции

поиска на одном устройстве, контроллер может инициировать поиск на другом устройстве. Многие контроллеры жестких дисков также умеют совмещать операцию чтения или записи на одном диске с поиском на другом или даже нескольких дисках. Однако контроллеры гибких дисков не могут одновременно читать и писать на двух дисководах. (Чтение или запись требуют от контроллера перемещения битов с максимальной скоростью, на которую он рассчитан, поэтому операция чтения или записи отнимает большую часть его вычислительных мощностей.) В случае IDE-дисков с их встроенными микроконтроллерами ситуация радикально меняется, поэтому несколько жестких дисков способны одновременно работать в одной системе, по крайней мере, переносить данные между диском и буфером контроллера. Вместе с тем, между контроллером и оперативной памятью в каждый момент времени происходит только одна операция по переносу данных. Способность параллельного выполнения двух или более дисковых операций может существенно снизить среднее время доступа.

В табл. 3.5 сравниваются параметры стандартного флоппи-дисковода оригинального компьютера IBM PC с параметрами современного жесткого диска для демонстрации того, насколько сильно изменились магнитные диски за последние два десятилетия. Интересно отметить, что не все параметры улучшились в равной мере. Среднее время поиска дорожки сократилось в семь раз, скорость передачи данных увеличилась в 1300 раз, тогда как емкость диска возросла в 50 000 раз. Это различие вызвано относительно незначительными усовершенствованиями механической части по сравнению с гораздо более ощутимым прогрессом в области увеличения плотности записи.

Таблица 3.5. Параметры 360-Кбайтового НГМД в сравнении с жестким диском Western Digital WD 18300

Параметр	НГМД IBM 360 Кбайт	Жесткий диск WD 18300
Количество цилиндров	40	10601
Дорожек в цилиндре	2	12
Секторов на дорожке	9	281 (среднее)
Секторов на диске	720	35 742 000
Байтов в секторе	512	512
Емкость диска	360 Кбайт	18,3 Гбайт
Время поиска (соседние цилиндры)	6 мс	0,8 мс
Время поиска (среднее)	77 мс	6,9 мс
Период обращения	200 мс	8,33 мс
Время запуска/остановки двигателя	250 мс	20 с
Время передачи сектора	22 мс	17 мкс

Глядя на спецификации современных жестких дисков, следует помнить, что указанная геометрия, используемая программным обеспечением драйвера, может отличаться от физического формата. Например, для показанного в табл. 3.5 жесткого диска рекомендуется устанавливать такие параметры: 1024 цилиндра,

16 головок и 64 сектора в дорожке. Логические номера головок и секторов, которыми оперирует операционная система, преобразуются встроенным в диск контроллером в физические, задействуемые в работе диска. Это еще один пример, как можно обеспечить совместимость новых устройств со старыми системами без изменения встроенных программ. Разработчики оригинальных IBM PC под счетчик секторов в ПЗУ BIOS выделили только 6 бит, поэтому диски, у которых больше 63 секторов в дорожке, вынуждены работать с искусственным набором логических параметров. В данном случае в спецификации производителя указано, что у диска в действительности 4 головки, таким образом, у него должно быть 252 сектора в дорожке (что и указано в таблице). Это упрощение ситуации, так как у подобных дисков на внешних дорожках секторов больше, чем на внутренних. У описываемого таблицей диска четыре физические головки, но цилиндров в действительности немногим более 3000. Цилиндры сгруппированы в дюжину зон и во внутренней зоне они содержат по 57 секторов на дорожку, а во внешней — 105. Эти параметры нельзя найти в спецификации диска, преобразование координат выполняется контроллером автоматически, что избавляет нас от необходимости знать их.

3.7.2. Программное обеспечение жестких дисков

В этом разделе мы изучим некоторые общие вопросы работы драйверов жестких дисков. Прежде всего рассмотрим, сколько времени требуется для считывания блока на диске. Это время складывается из трех слагаемых.

1. Время поиска (перемещение головки на нужный цилиндр).
2. Задержка вращения (время, через которое нужный сектор пройдет под головкой).
3. Время передачи данных.

Для большинства дисковых накопителей наибольший вклад в задержку имеет первый фактор, поэтому уменьшение среднего времени поиска принципиально важно для роста производительности системы.

Дисковые накопители склонны к ошибкам. В силу этого в каждый сектор диска всегда записывается та или иная проверочная информация, в виде контрольной суммы, иначе циклического избыточного кода (CRC). Даже адреса секторов, которые записываются на диск при форматировании, содержат проверочные данные. Контроллер гибкого диска, обнаружив ошибку, сообщает о ней системе, которая должна решить, как нужно поступить. Контроллеры жестких дисков часто берут на себя значительную часть действий по обработке ошибки.

Для жестких дисков последовательное чтение секторов в пределах одной дорожки происходит очень быстро. Поэтому для повышения эффективности работы системы имеет смысл считывать больше секторов, чем запрошено, и кэшировать их в памяти.

Алгоритмы планирования перемещения головки

Если драйвер диска принимает и выполняет запросы по одному в порядке получения, то есть по принципу «первым пришел — первым обслужен» (FCFS, First Come, First Served), тогда мало что можно сделать для оптимизации времени поиска. Однако при высокой загрузке диска допустимо применение другой стратегии. В этом случае высока вероятность поступления новых запросов от других процессов во время перемещения головки для обработки предыдущего запроса. Многие дисковые драйверы содержат таблицу, индексированную по номерам цилиндров, в которой в единый цепной список собираются все поступившие и ждущие обработки обращения к цилиндрам.

С помощью подобной структуры данных можно создать более совершенный алгоритм планирования, чем простое обслуживание в очередности поступления запросов. Рассмотрим, например, диск с 40 цилиндрами. Поступает запрос на чтение блока с цилиндра 11. Во время перемещения головки на 11-й цилиндр делаются новые обращения к цилиндрам 1, 36, 16, 34, 9 и 12. Новые запросы помещаются в таблицу, составленную из отдельных списков для каждого цилиндра. Поступившие запросы помечены крестиками на рис. 3.13.

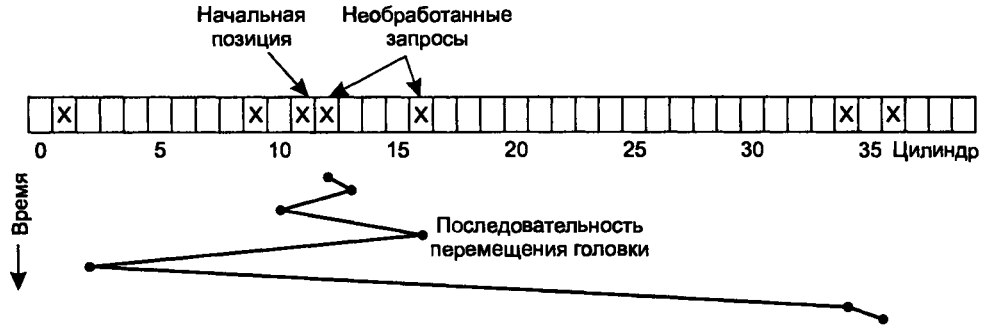


Рис. 3.13. Алгоритм планирования «ближайший цилиндр первым»

Выполнив первый запрос (обращение к цилиндру 11), драйвер диска должен выбрать на обслуживание следующий запрос. Если обслуживать запросы в порядке поступления, то драйвер должен переместить головку на цилиндр 1, затем на цилиндр 36 и т. д. В результате выполнение этого алгоритма потребует перемещения блока головок на 10, 35, 20, 18, 25 и 3 цилиндра, что в сумме составит 111 цилиндров.

Время перемещения головки можно уменьшить, выбирая каждый раз ближайший цилиндр. При той же серии запросов, показанной на рис. 3.13, последовательность их обработки выглядит как ломаная кривая внизу рисунка. При такой последовательности выполнения запросов потребуются перемещения блока головок на 1, 3, 7, 15, 33 и 2 цилиндра, что суммарно равно 61 цилиндру. Применение данного алгоритма, называющегося *SSF* (Shortest Seek First — «ближайший запрос первым»), минимизирует суммарные перемещения головок почти вдвое.

К сожалению, у данного алгоритма есть свои недостатки. Предположим, во время обработки запросов, показанных на рис. 3.13, поступили новые запросы. Например, после обработки обращения к цилиндру 16 приходит новый запрос к цилиндру 8. Этот запрос будет иметь приоритет над обращением к цилиндру 1. Затем, если поступит запрос к цилиндру 13, головка опять начнет перемещаться к цилиндру 13 для обработки нового запроса, а запрос к цилиндру 1 останется необработанным. При сильной загрузке диска головка диска будет большую часть времени находиться где-то в районе средних цилиндров, а запросам к крайним цилиндрам диска придется ждать, пока нагрузка диска случайно не снизится и обращений к середине диска не станет меньше. В результате качество обслуживания запросов к цилиндрам, удаленным от срединной части, может оказаться низким. То есть в конфликт вступают принцип минимизации времени отклика и принцип справедливости.

В высотных зданиях также приходится иметь дело с данной проблемой планирования обслуживания запросов лифта. Вызовы лифта постоянно поступают с разных этажей. Компьютер, управляющий лифтом, должен следить за последовательностью поступления запросов и удовлетворять их либо в порядке подачи заявок, либо обслуживая первым ближайший запрос.

Однако в большинстве лифтов применяются различные алгоритмы, пытающиеся примирить конфликтующие цели эффективности и справедливости. Обычно лифт продолжает двигаться в одном направлении до тех пор, пока на этом направлении более не остается запросов. Затем лифт меняет направление движения. Этот алгоритм, называющийся *элеваторным алгоритмом*, требует от программного обеспечения отслеживания всего одного бита, хранящего информацию о текущем направлении движения, **ВВЕРХ** или **ВНИЗ**. Выполнив очередной запрос, диск или лифт проверяет значение бита. Если в нем содержится значение **ВВЕРХ**, кабина лифта или блок головок перемещается вверх к следующему запросу. Если желающих подняться больше нет, состояние бита инвертируется.

Рисунок 3.14 иллюстрирует работу элеваторного алгоритма с теми же семью запросами, что были использованы в качестве примера на рис. 3.13. Предполагается, что изначально бит направления движения указывал **ВВЕРХ**. При этом цилиндры обслуживаются в порядке 12, 16, 34, 36, 9 и 1, что соответствует перемещению блока головок на 1, 4, 18, 2, 27 и 8 цилиндров и в сумме составляет 60 цилиндров. В данном случае элеваторный алгоритм оказался даже слегка лучше, чем алгоритм SSF, однако на практике он обычно несколько хуже. Достоинство элеваторного алгоритма состоит в том, что при любом заданном наборе запросов верхняя граница необходимых перемещений блока головок для выполнения всех запросов фиксирована и ограничена удвоенным числом цилиндров.

Незначительная модификация этого алгоритма с меньшим разбросом времени отклика состоит в том, чтобы сканировать цилиндры всегда в одном направлении [82]. После обслуживания обращения к цилиндру с самым высоким номером блок головок перемещается к самому нижнему запрашиваемому цилиндру и далее продолжает движение вверх. Таким образом, самый нижний цилиндр как бы считается расположенным выше самого верхнего.

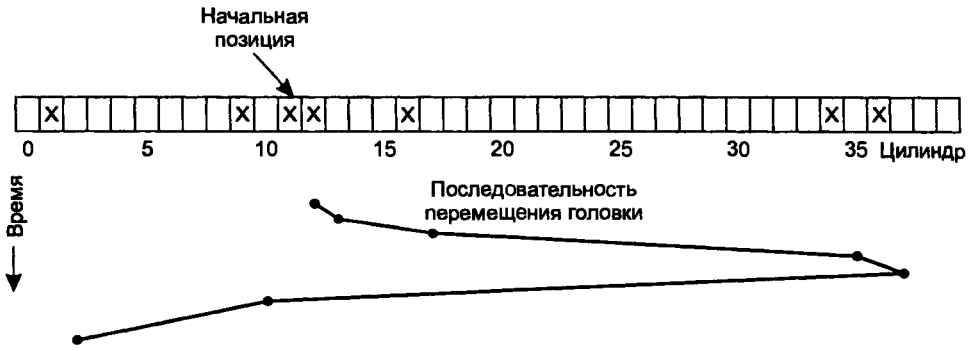


Рис. 3.14. Элеваторный алгоритм планирования обращений к диску

Некоторые контроллеры дисков предоставляют программному обеспечению способ узнавать номер текущего сектора под головкой. Такие контроллеры позволяют использовать дополнительный метод оптимизации. Если есть два или более запросов к одному и тому же цилиндру, драйвер может выбрать из них тот, сектор которого пройдет под головкой первым. Обратите внимание, что при наличии нескольких головок у диска последовательные запросы могут обращаться к различным дорожкам на одном цилиндре. Контроллер может практически мгновенно переключаться с головки на головку, так как такое переключение не требует ни перемещения блока головок, ни ожидания поворота диска.

Если у диска время перемещения головок значительно ниже задержки, связанной со скоростью вращения диска, следует применять другую стратегию оптимизации. Запросы, ожидающие обработки, должны сортироваться по номерам секторов, и как только следующий сектор приблизится к головке, блок головок должен быстро переместиться на нужную дорожку, чтобы считать или записать его.

У современных жестких дисков производительность настолько зависима от задержек перемещения головок и вращения диска, что читать по одному или по два сектора крайне неэффективно. В силу чего многие современные контроллеры дисков читают и сохраняют в кэше по несколько секторов сразу, даже если запрашивается всего один сектор. Обычно при этом считывается запрашиваемый сектор и все остальные секторы дорожки, при условии что для них достаточно места в кэше контроллера. Например, у диска, описанного в табл. 3.5, размер кэша обычно составляет 2 Мбайт или 4 Мбайт. Режим использования кэша динамически определяется контроллером. В простейшем режиме кэш разделяется на две секции, одна для запросов чтения, другая для запросов записи. Если последующий запрос на чтение сектора может быть удовлетворен прямо из кэша контроллера, запрашиваемые данные могут быть выданы немедленно.

Следует отметить, что кэш контроллера диска абсолютно никак не связан с кэшем операционной системы. Кэш контроллера обычно содержит блоки, на которые запрос еще не поступал, но которые было удобно прочитать, так как они случайно оказались под головкой при чтении других блоков. Напротив, любой кэш операционной системы состоит из блоков, на чтение которых были явно

направлены запросы и которые, по мнению операционной системы, могут понадобиться снова в ближайшем будущем (например, блок диска, содержащий каталог).

При одновременном обслуживании контроллером нескольких устройств таблица запросов, ждущих обработки, должна поддерживаться отдельно для каждого устройства. Когда любое из устройств завершает выполнение текущего запроса, ему требуется дать команду переместить блок головок на цилиндр для выполнения следующего запроса (при условии, что контроллер позволяет работать в режиме перекрывающегося поиска). По завершении текущей операции переноса данных может быть выполнена проверка правильного позиционирования блоков головок всех устройств. Если хотя бы один блок установлен, может быть начат следующий перенос данных. В противном случае драйвер должен выдать новую команду поиска цилиндра устройству, только что завершившему операцию переноса данных, после чего перейти в состояние ожидания прерывания от диска, установившего блок головок на нужный цилиндр и готового к перемещению данных.

Важно понимать, что во всех вышеизложенных алгоритмах планирования работы дисков молчаливо подразумевается, что физическая геометрия дисков совпадает с виртуальной. Иначе планирование обращений к диску не имеет никакого смысла, так как операционная система не сумеет определить, какой цилиндр расположен ближе к цилиндру: 39-й, 40-й или 200-й. С другой стороны, если контроллер диска способен принимать несколько внешних запросов одновременно, он в состоянии осуществлять все планирование внутри себя. В этом случае алгоритмы сохраняют силу, но выполняются на более низком уровне, в контроллере.

Обработка ошибок

Драйверу RAM-диска не нужно заботиться об оптимизации поиска сектора, так как в любой момент можно прочитать или записать любой блок, не используя для этого никаких механических движений. Обработка ошибок — еще одна область, в которой RAM-диск намного проще. RAM-диск работает всегда, в то время как для реального диска это не так. Реальные дисковые накопители подвержены самым разнообразным ошибкам. Вот самые распространенные из них.

1. Ошибки программирования (например, запрос несуществующего сектора).
2. Временная ошибка контрольной суммы (например, вызванная пылью, попавшей на головку).
3. Постоянная ошибка контрольной суммы (физическое повреждение блока).
4. Ошибка поиска (допустим, головка была отправлена на 6-й цилиндр, но оказалась на 7-м).
5. Ошибка контроллера (например, контроллер отказался принимать команду).

Обрабатывать все эти ошибки наилучшим образом — задача драйвера диска.

Ошибки программирования возникают, когда драйвер передает контроллеру команду на переход к несуществующему цилиндру, на чтение несуществующего сектора, использование несуществующей головки или на передачу данных в несуществующую область памяти. Большинство контроллеров проверяют переда-

ваемые им параметры и уведомляют о неполадках. Теоретически таких ошибок возникать не должно, но что делать, если контроллер все же сообщит об этом? Для доморощенной системы лучше всего было бы вывести на экран текст «свяжитесь с разработчиком», чтобы ошибку можно было найти или исправить. Для коммерческой системы, работающей на многих тысячах компьютеров по всему миру, такое поведение менее привлекательно. Вероятно, лучшее, что можно сделать в данном случае, это завершить текущий запрос с кодом ошибки и надеяться, что она не будет случаться слишком часто.

Временные ошибки контрольной суммы вызываются пылинками, которые остаются в воздухе между головкой и поверхностью диска. С ними можно бороться, повторив несколько раз операцию. Если же повторным чтением ошибка не устраняется, блок помечается как поврежденный (bad block). Такой блок в дальнейшем избегается.

Один из способов уклонения от работы с поврежденными блоками опирается на специальную программу, которая получает на входе список поврежденных блоков и по этим данным создает файл, состоящий целиком из таких блоков. Они будут отмечены как занятые, и в результате ни одна программа не будет пытаться обращаться к ним. Поэтому, пока ни одна программа не попытается прочитать файл сбойных блоков, ошибка не возникнет.

Не читать порченные блоки — это проще сказать, чем сделать. Часто резервные копии дисков создаются путем подорожечного копирования содержимого диска на ленту или другой диск. Если следовать этой процедуре, избежать поврежденных блоков невозможно. Пофайловое резервное копирование медленнее, но может решить эту проблему, если программа знает имя файла «битых» блоков и не станет пытаться скопировать его.

Другая проблема, не решаемая при помощи файла поврежденных блоков, — это нарушение системной структуры, которая должна занимать фиксированное положение на диске. Практически у любой файловой системы есть структура, положение которой должно быть одним и тем же, с целью облегчения ее поиска. Если файловая система разбита на разделы, то можно переразбить ее и попробовать обойти сбойный блок, но если повреждены первые несколько секторов дискеты или жесткого диска, накопитель станет непригодным к использованию.

«Умные» контроллеры резервируют несколько дорожек, которые недоступны пользовательским программам. При форматировании диска контроллер определяет, какие из секторов содержат ошибки, и автоматически замещает их одним из запасных. Таблица, отображающая поврежденные блоки на запасные, хранится во внутренней памяти контроллера и на диске. Подмена происходит незаметно для драйвера, за исключением того, что не исключена потеря эффективности тщательно отработанного элеваторного алгоритма, если контроллер тайно будет использовать цилиндр 800 вместо запрошенного 3-го. Технология производства магнитных поверхностей исключительно точна, но она все равно не идеальна. Поэтому имеет большое значение механизм сокрытия таких дефектов от пользователя. Для жестких дисков (описание одного из них приведено в табл. 3.5) контроллер отслеживает появление новых сбойных блоков и, если ошибка неустранима, заменяет их запасными блоками. Работая с таким диском, драйвер практически никогда не будет видеть плохие блоки.

Дефектные секторы не являются единственным источником ошибок. Также возникают ошибки поиска цилиндра, вызванные механическими проблемами блока головок. Контроллер следит за положением блока головок. При установке головки на заданный цилиндр он выдает серию импульсов двигателю блока головок, по одному импульсу на цилиндр. Когда блок головок устанавливается в требуемое положение, контроллер считывает истинное значение цилиндра из заголовка первого попавшегося сектора. Если блок головок оказывается не на той дорожке, на которой нужно, возникает ошибка поиска.

Практически все контроллеры жестких дисков автоматически исправляют ошибки поиска, но большинство контроллеров гибких дисков (включая установленные на компьютерах с процессором Pentium) просто выставляют бит ошибки и оставляют все остальное драйверу. Драйвер обрабатывает ошибку, выдавая команду recalibrate. При этом блок головок отодвигается на самую внешнюю дорожку диска до упора. Это положение принимается контроллером за нулевую дорожку, таким образом, контроллер снова начинает понимать, где находится блок головок. Обычно это решает проблему. Если же нет, то либо нужно сменить диск, либо починить дисковод.

Как мы видели, контроллер в действительности представляет собой небольшой специализированный компьютер, полный программного обеспечения, переменных, буферов и, по всей видимости, ошибок. Иногда необычное стечение обстоятельств, например приход прерывания с одного устройства, в то время как другое устройство выполняет команду recalibrate, может разбудить спящую до тех пор ошибку, в результате которой контроллер войдет в бесконечный цикл или забудет, что он делал. Разработчики контроллеров обычно готовятся к худшему и предоставляют на всякий случай специальный контакт на микросхеме, обращение к которому вызывает сброс контроллера. Если никакие другие меры не помогают, драйвер может установить этот бит, отвечающий за подачу сигнала на такой контакт, и сбросить контроллер. Если и это не решило вопроса, то все что драйверу остается сделать — это вывести сообщение и сдаться.

Кэширование дорожек

Время, требующееся на переход к новому цилиндру, обычно много больше, чем время поворота диска и всегда больше времени передачи. Другими словами, если драйвер решил перевести куда-либо головку, многое зависит от того, будет ли читаться только один сектор или вся дорожка. Эффект в особенности заметен в том случае, если контроллер позволяет определить, над каким сектором находится дорожка, чтобы драйвер мог отдать ему команду на чтение следующего сектора, тем самым считывая дорожку целиком за один оборот диска. (Обычно в среднем на чтение отдельного сектора требуется половина оборота диска плюс время на чтение одного сектора.)

Некоторые драйверы поддерживают специальный кэш дорожки, невидимый для программ. Если запрошенный у контроллера сектор находится в этом кэше, обмениваться данными с диском не нужно. Недостаток такого подхода (помимо усложнения драйвера и необходимости выделения памяти для буфера) в том,

что передача данных пользовательской программе из кэша должна производиться процессором, при помощи программного цикла, а не DMA.

По данной причине многие современные контроллеры поддерживают кэш дорожки самостоятельно, в своей внутренней памяти. В этом случае кэширование происходит прозрачно для драйвера и передачу данных можно производить при помощи DMA. Если контроллер поддерживает подобную функцию, не имеет большого смысла дублировать ее в драйвере. Заметьте, что обеспечить чтение одной дорожки целиком могут как контроллер, так и драйвер диска, но не пользовательская программа, так как для пользовательских программ диск представляется линейной последовательностью блоков, без разбиения на дорожки и цилиндры.

3.7.3. Обзор драйверов жестких дисков в MINIX

Драйвер жесткого диска — это первая рассмотренная нами составная часть MINIX, которая работает с широким диапазоном оборудования различных типов. Поэтому, прежде чем обсуждать детали драйвера, мы кратко рассмотрим некоторые проблемы, порождаемые различиями в аппаратном обеспечении. Под обозначением «IBM PC» в действительности скрывается семейство различных компьютеров, которые иногда имеют существенные расхождения. Первые представители семейства, оригинальные PC и PC-XT, работали с 8-битной шиной, соответствующей 8-разрядному внешнему интерфейсу процессора 8088. Следующее поколение, PC-AT, работали с 16-битной шиной, разработанной так, чтобы быть совместимой и со старыми 8-битными устройствами. Новые 16-разрядные устройства со старой шиной работать не могли. Шина AT была изначально рассчитана на системы, основанные на процессоре 80286, и использовалась во многих системах на основе 80386, 8486 и Pentium. Но эти процессоры имеют 32-разрядный интерфейс, поэтому существует несколько вариаций 32-битной шины, например Intel PCI.

Для каждой шины имеется отдельное семейство *адаптеров ввода/вывода*, подключаемых к объединительной плате. Вся периферия для определенного типа шины должна поддерживать стандарты этой шины, но не обязана быть совместимой с более старыми стандартами. В семействе IBM PC, как и в других системах, шина поставляется вместе со встроенным программным обеспечением, находящемся в ПЗУ BIOS (Basic I/O System, базовая система ввода/вывода). BIOS служит «мостом» между различными операционными системами и особенностями конкретного аппаратного обеспечения. Некоторые периферийные устройства содержат расширения BIOS, хранящиеся в чипах ПЗУ на самой карте. Проблема, с которой сталкивается разработчик операционной системы, состоит в том, что BIOS в IBM-совместимых компьютерах (особенно в ранних версиях) разрабатывалась для целей MS-DOS, операционной системы без поддержки многозадачности и работающей в 16-разрядном режиме, который является простейшим из режимов работы процессоров семейства 80x86.

Таким образом, перед создателем новой операционной системы для IBM PC встает несколько вопросов. Первый из них: использовать ли для поддержки обо-

рудования BIOS или писать новые драйверы с нуля. Для разработчиков MINIX выбор был несложен, так как возможности BIOS во многом не соответствовали потребностям MINIX. Конечно, для того чтобы выполнить начальную загрузку системы с дискеты или жесткого диска, монитор загрузки пользуется функциями BIOS, так как здесь практически нет другой альтернативы. Но загруженная система с собственными драйверами ввода/вывода способна на гораздо больше, чем BIOS.

Второй вопрос звучит так: как, без поддержки BIOS, обеспечить работу драйверов с различным оборудованием? Чтобы сделать вопрос более конкретным, рассмотрим контроллеры жестких дисков, которые на системах, поддерживаемых MINIX, могут быть четырех принципиально различных типов. Это оригинальный 8-битный контроллер XT, 16-битный контроллер AT и два различных контроллера для компьютеров серий PS/2. Есть несколько возможных путей решения данной проблемы.

1. Для каждого типа дискового контроллера компилировать отдельную версию операционной системы.
2. Встроить в ядро несколько различных драйверов и дать системе возможность автоматически выбирать нужный во время загрузки.
3. Встроить в ядро несколько различных драйверов и предоставить пользователю право выбрать подходящий.

Как мы увидим, эти решения не являются взаимоисключающими.

Первая практика оптимальна при долговременной работе. Если операционная система работает на конкретной машине, нет необходимости хранить в памяти код драйверов, которые никогда не будут использованы. С другой стороны, такой вариант кошмарно неудобен для распространителя системы. Предоставлять несколько загрузочных дисков и инструктировать пользователя, какой из них в каком случае использовать, — сложно и дорого. Таким образом, более предпочтительны два других решения, по крайней мере, для начальной установки.

При втором подходе ОС должна определить имеющееся оборудование, считывая ПЗУ на периферийных картах или обмениваясь с ними данными через порты ввода/вывода. На некоторых системах это выполнимо, но для IBM такое решение не всегда работает, так как для этих систем существует слишком большое количество нестандартного оборудования. Попытка передать данные в порт ввода/вывода, чтобы определить одно устройство, может активизировать другое устройство, которое захватит управление и заблокирует систему. Подобный метод усложняет начальный код для каждого из устройств и все равно работает не слишком хорошо. Поэтому операционные системы, опирающиеся на него, должны предоставлять возможность вручную корректировать результаты автоматического детектирования. Обычно это механизмы, подобные используемым в MINIX.

В третьем подходе, присущем MINIX, компилируются несколько драйверов, один из которых выбирается по умолчанию. Монитор начальной загрузки считывает различные *параметры загрузки*. Эти параметры могут быть введены

вручную либо храниться на диске. Если, например, при загрузке ищется параметр, имеющий вид

```
hd = xt
```

то используется драйвер диска для XT. Если подобного параметра не обнаружено, применяется драйвер по умолчанию.

В MINIX есть еще два средства, уменьшающие проблемы с различными драйверами жестких дисков. Прежде всего, это специальный драйвер, который в своей работе использует функции BIOS. Этот драйвер гарантированно заработает практически с любой системой. Чтобы выбрать его, необходимо задать следующий параметр загрузки:

```
hd = bios
```

Тем не менее этот драйвер стоит рассматривать лишь как последнюю инстанцию. На системах с процессором 80286 и старше MINIX работает в защищенном режиме, а код BIOS всегда выполняется в реальном режиме (режим 8086). Переключение режимов при каждом вызове функции BIOS требует очень много времени.

Кроме того, еще одна стратегия, которая используется в MINIX при работе с драйверами, сводится к тому, чтобы отложить инициализацию до самого последнего момента. Таким образом, если ни один из драйверов жестких дисков не работает с некоторой конфигурацией, мы все равно можем запустить MINIX с дискеты и выполнить некоторые полезные действия. Это не кажется большим прорывом в дружественности к пользователю, но примите во внимание: если бы все драйверы пытались инициализироваться непосредственно при запуске системы, то неправильная установка некоторых устройств, которые все равно лежат балластом, могла бы полностью парализовать ОС. Отложив инициализацию драйверов, система может работать с тем, что работает, давая пользователю тем самым возможность исправить ситуацию.

Отступая в сторону, заметим, что этот урок тяжело нам дался. Ранние версии MINIX пытались инициализировать жесткий диск при загрузке системы. Если жесткого диска не было, система зависала. Такое поведение было особенно неприятным потому, что MINIX рассчитана и на машины без жесткого диска, хотя это и уменьшает доступный объем памяти и производительность.

В этом и следующем разделах мы будем рассматривать драйвер жесткого диска в стиле AT, устанавливаемый в MINIX по умолчанию. Это многоцелевой драйвер, обеспечивающий поддержку широкого диапазона контроллеров, как ранних, применявшихся в 286-х системах, так и современных EIDE-контроллеров. В основном общие аспекты работы жесткого диска, которые мы включим в рассмотрение, относятся и к остальным драйверам.

Главным циклом задачи жесткого диска служит уже изученный нами единый код, поддерживающий шесть стандартных запросов. Запрос DEV_OPEN неприятен потенциальным немалым количеством работы, так как на жестком диске всегда есть разделы, и с высокой вероятностью подразделы. При открытии устройства (то есть при первом обращении к нему) эта информация должна быть прочитана.

Некоторые контроллеры жестких дисков также поддерживают приводы CD-ROM со сменным носителем. В этом случае при обработке DEV_OPEN должно проверяться наличие носителя в приводе. Для CD-ROM имеет значение и операция DEV_CLOSE: она приводит к отпиранию дверцы привода и выбросу диска. В случае сменного носителя есть и другие сложности, в большей степени имеющие отношение к гибким дискам, поэтому мы рассмотрим их позже. В данном случае, чтобы «выбросить» диск, используется операция DEV_IOCTL, устанавливающая значение флага, который будет анализироваться при работе DEV_CLOSE. Кроме того, DEV_IOCTL применяется для записи и чтения таблиц разделов.

Каждый из запросов DEV_READ, DEV_WRITE и SCATTERED_IO, как мы видели, выполняется в три действия: подготовка, планирование и завершение. В отличие от RAM-дисков у жесткого диска фазы планирования и завершения принципиально различны. Драйвер не задействует такие алгоритмы планирования перемещения головки, как SSF или элеваторный алгоритм, но поддерживает более ограниченный алгоритм, объединяющий запросы последовательных секторов. При нормальной работе запросы обычно отправляются файловой системой, которую интересуют блоки, кратные 1024 байт. Драйвер же обрабатывает запросы, кратные величине сектора (512 байт). Пока поступающие запросы адресуют цепочку последовательных секторов, каждый новый запрос помещается в конец очереди. Эта очередь организована в виде массива, и, когда массив заполняется или поступает запрос, не укладывающийся в цепочку, делается вызов для завершения обмена данными.

Обыкновенные вызовы DEV_READ и DEV_WRITE могут запрашивать более одного блока, но за каждым вызовом процедуры планирования непосредственно следует вызов процедуры, которая обеспечивает, что текущий список запросов завершен. В случае операции SCATTERED_IO вызову завершающей процедуры может предшествовать несколько вызовов процедуры планирования. Пока запрашиваются блоки последовательных данных, список расширяется до заполнения массива. Вспомните, что у вызова SCATTERED_IO есть флаг, который отмечает необязательные запросы. Драйвер жесткого диска, как и драйвер RAM-диска, игнорирует этот флаг и доставляет все запрошенные блоки.

Рудиментарное планирование, которое драйвер жесткого диска осуществляет, откладывая выполнение на время запроса последовательных блоков, нужно рассматривать как второй шаг предположительно трехэтапного алгоритма планирования. Сама файловая система при помощи рассеянного ввода/вывода могла бы реализовать что-либо наподобие элеватора в версии Теори[1], наподобие тому, как в запросе разрозненного ввода/вывода запросы сначала сортируются по номеру блока. Третий этап планирования происходит в контроллере современного диска, например, того, что описан в табл. 3.5. Подобные контроллеры достаточно «умные» и умеют хранить в буфере большой объем данных, тем самым обеспечивая максимальную эффективность передачи больших объемов данных при помощи внутренних алгоритмов, безотносительно к порядку поступления запросов.

[1] См. [82]. — *Примеч. ред.*

3.7.4. Реализация драйвера жесткого диска в MINIX

Жесткие диски часто называют «винчестерами». Существует несколько историй о том, как произошло это название. Такое кодовое название имел проект IBM, разрабатывавшей технологию, в которой магнитные головки «парят» над поверхностью вращающегося диска, опираясь на тонкую прослойку воздуха. Одно из толкований названия в том, что у первых моделей было два герметически закрытых корпуса — модуля данных, фиксированный (30 Мбайт) и сменный (тоже 30 Мбайт). Предположительно, это напомнило разработчикам винчестер «30-30», который фигурирует во многих вестернах[1]. Каково бы ни было происхождение названия, основа технологии остается той же самой, хотя типичные жесткие диски современных микрокомпьютеров гораздо миниатюрнее и хранят намного большие объемы данных, чем их 14-дюймовый прототип начала 1970-х.

Задача кода в файле wini.c в том, чтобы скрыть реальный драйвер жесткого диска от остального ядра. Это позволяет включать в ядро несколько драйверов, выбирая нужный при загрузке. Позже ядро можно перекомпилировать, оставив только тот драйвер, который необходим.

В файле wini.c имеется только одно описание данных, задающее массив hdmap. Этот массив связывает имена драйверов с адресами функций. Он инициализируется компилятором и содержит столько элементов, сколько необходимо для драйверов жестких дисков, подсоединенных при компиляции в файле include/minix/config.h. Используется он в функции winchester_task, имя которой помещается в массив task_tab, необходимый при запуске ядра. Когда вызывается функция winchester_task, она сначала пытается найти переменную окружения с именем hd при помощи системной функции, работающей так же, как и аналогичные механизмы в обычных программах на языке C, через считывание переменных, установленных монитором начальной загрузки. Если переменная hd не задана, используется первая запись из массива (то есть драйвер по умолчанию). Соответствующая выбранному драйверу функция вызывается косвенно, по адресу, хранящемуся в массиве. В последующем разделе мы будем рассматривать работу функции at_winchester_task, адрес которой в стандартной версии MINIX находится в первой ячейке массива hdmap.

Код драйвера жесткого диска стандарта AT содержится в файле at_wini.c. Это запутанный драйвер для такого же сложного устройства. Три страницы кода занимают макроопределения, необходимые для обращения к регистрам контроллера, битам состояния, структурам данных, а также прототипы функций. Как и в случае с драйверами других блочных устройств, в драйвере винчестера создается структура типа driver, хранящаяся в переменной w_dtab. Эта структура заполняется адресами функций, которые и выполняют все основные задачи драйвера. Код большинства функций находится в файле at_wini.c, но, так как для жесткого диска не нужны специальные действия для завершения работы, поле dr_cleanup

[1] Согласно *Microsoft Computer Dictionary*, описываемый диск имел емкость 30 Мбайт и время доступа 30 мс (аналогия с патронами «Winchecter .30–.30», калибром 0,3 дюйма — 7,62 мм и весом 30 гран — 1,9 г). — *Примеч. ред.*

структуры содержит ссылку на функцию nop_cleanup в файле driver.c. Последняя используется всеми драйверами, которые не нуждаются в особых действиях для завершения. Точка входа в драйвер — это функция at_winchester_task, которая выполняет аппаратно-зависимую инициализацию и передает управление в главный цикл (файл driver.c). Главный цикл исполняется бесконечно, обслуживая поступающие запросы, вызывая функции, адреса которых записаны в полях структуры w_dtab.

Итак, теперь мы имеем дело с реальным электромеханическим устройством хранения данных, поэтому для инициализации драйвера необходимо выполнить немало действий. Различные параметры, относящиеся к жесткому диску, хранятся в массиве wini. Как часть политики отложенной инициализации функция init_params, вызываемая при инициализации ядра, не выполняет ничего такого, что может привести к сбою, то есть не делает никаких обращений к самому дисковому устройству. Основная ее задача — поместить в массив wini некоторую информацию о логической конфигурации жесткого диска. Эти сведения извлекаются при помощи BIOS из CMOS-памяти, которая используется некоторыми компьютерами для хранения основных параметров конфигурации. Код BIOS выполняется в самом начале работы компьютера, до того как начнется первый этап процесса загрузки MINIX. Невозможность получить сохраненные в CMOS параметры необязательно означает сбой. У некоторых современных дисков эта информация может быть получена от самого диска.

После вызова главного цикла некоторое время ничего не происходит, до тех пор пока не будет сделана попытка обращения к жесткому диску. Затем драйвер получает запрос DEV_OPEN и для его обработки косвенно вызывает функцию w_do_open. В свою очередь, она вызывает w_prepare, чтобы определить, корректно ли указано устройство, а далее, чтобы выяснить тип устройства и инициализировать некоторые дополнительные параметры, вызывает w_identify. Счетчик в массиве wini нужен для того, чтобы определить, является ли текущий вызов первым после запуска MINIX. После каждого вызова значение счетчика увеличивается. Если обнаружено, что это первая операция — DEV_OPEN, вызывается функция partition.

Следующая функция, w_prepare, принимает один целочисленный аргумент, device, равный младшему номеру устройства, и возвращает указатель на структуру device, которая содержит информацию об адресе и размере устройства. (В языке C идентификаторы, означающие имя структуры, разрешено использовать в качестве имени переменной.) Тип устройства (является ли оно диском, разделом или подразделом) можно определить по его младшему номеру. После того как w_prepare завершает свою работу, нет необходимости вызывать никакие записывающие или считывающие данные функции, связанные с разбиением на разделы. Как вы могли увидеть, w_prepare вызывается в ответ на запрос DEV_OPEN. Это также является одной из частей трехэтапного цикла подготовка/планирование/завершение, применяемого во всех операциях обмена данных. В этом контексте важно, чтобы переменная w_count инициализировалась нулевым значением.

Программно совместимые со стандартом AT диски находятся в употреблении довольно давно, и функция w_identify должна определить, какой из появившихся

за годы вариантов жесткого диска используется. На первом шаге выясняется, доступны ли для чтения и записи порты ввода/вывода, которые должны существовать для всех дисковых контроллеров одного семейства. Если эта проверка завершилась успехом, адрес обработчика прерываний жесткого диска помещается в таблицу дескрипторов прерываний и контроллеру прерываний выдается инструкция отвечать на прерывания от устройства. Затем контроллеру диска дается команда ATA_IDENTIFY. Если она завершилась успешно, извлекается различная информация, включая строку-идентификатор модели диска и параметры физических секторов, головок и цилиндров. (Заметьте, что «физическая» конфигурация в действительности может не совпадать с реальной физической структурой, но у нас нет другой альтернативы, кроме как верить в то, что заявляет диск.) Кроме того, эти сведения содержат информацию о том, поддерживает ли диск *линейную адресацию блоков* (Linear Block Addressing, LBA). Если данная функция поддерживается, драйвер вправе игнорировать цилиндры, дорожки и секторы и адресовать секторы диска просто по их абсолютным номерам, что намного проще.

Как уже было упомянуто ранее, не исключена ситуация, когда init_params окажется не в состоянии получить информацию о конфигурации диска из BIOS. Если такое случилось, код на следующих строках пытается сформировать подходящий набор параметров, основанных на тех данных, которые удалось прочитать с самого диска. Основная идея состоит в том, что номера цилиндра, дорожки и сектора не должны превышать соответственно 1023, 255 и 63, то есть учитывается заложенное в структуры данных BIOS ограничение на количество битов, выделяемых под эти параметры.

Если команда ATA_IDENTIFY возвращает отрицательный результат, это может просто означать, что вы столкнулись со старой моделью диска, который не поддерживает саму команду. В таком случае все, что мы имеем, — это параметры логической конфигурации, ранее прочитанные функцией init_params. Если они корректны, то они и записываются в поля структуры wini, в противном случае сообщается об ошибке и система отказывается работать с диском.

Наконец, чтобы считать адреса в байтах, в MINIX используется переменная u32_t. Максимальный объем устройства (в секторах), с которым умеет работать драйвер, ограничен произведением количества цилиндров, головок и секторов. На момент написания этой книги диски объемом 4 Гбайт мало распространены, но опыт показывает, что программное обеспечение должно проверять подобные ограничения, независимо от того, что на сегодняшний день такие тесты могут показаться излишними. Затем базовый адрес и размер всего диска записываются в массив wini и вызывается (дважды при необходимости) функция w_specify, передающая обратно контроллеру его рабочие параметры. В завершение имя устройства и идентифицирующая его строка, определенная при помощи identify (для усовершенствованных устройств), или параметры цилиндров и головок, определенные BIOS (для старых устройств), выводятся на консоль.

Функция w_name возвращает указатель на строку, содержащую имя устройства: «at-hd0», «at-hd5», «at-hd10» или «at-hd15». Функция w_specify, в дополнение

к передаче параметров контроллеру, выполняет повторную калибровку устройства (для старых моделей), передавая команду поиска нулевого цилиндра.

Теперь мы готовы обсудить функции, вызываемые при обработке запроса на передачу данных. Сначала вызывается уже рассмотренная нами функция w_prepare. Здесь важно то, что она инициализирует нулем значение переменной w_count. Далее при передаче данных вызывается функция w_schedule, принимающая три параметра: откуда брать данные, куда их поместить и какое количество байтов передавать. Количество байтов должно быть кратно размеру сектора, что проверяется в коде функции. Бит необязательного выполнения, который может присутствовать в запросе SCATTERED_IO, сбрасывается при передаче кода операции контроллеру, но заметьте, что он сохраняется в поле io_request структуры iorequest_s. Жесткий диск будет пытаться выполнять все запросы, но, как мы увидим далее, драйвер вправе отказаться от некоторых из них, если произошли ошибки. Затем производится контроль выхода последнего байта запроса за пределы устройства, и при необходимости уменьшается запрошенное количество байтов. К этому моменту вычисляется первый сектор, который будет запрошен.

Начиная отсюда, процесс планирования приобретает серьезный характер. Если уже имеются текущие запросы (для этого проверяется, равно ли значение переменной w_count нулю) и если новый запрошенный сектор не следует за последним сектором в очереди, вызывается функция w_finish, которая выполняет отложенные запросы. В противном случае в переменную w_nextblock, хранящую номер следующего сектора в цепочке, записывается свежее значение, и новый запрошенный сектор добавляется в очередь. Когда достигается максимально допустимое количество запросов, определяемое переменной max_count, опять же вызывается w_finish. Как мы увидим, хранить предельное значение в переменной удобно тем, что его несложно потом подстроить.

Итак, в функции w_prepare есть два места, где делается вызов w_finish. Обычно w_prepare завершается, не обращаясь к w_finish, но этот вызов так или иначе производится из главного цикла в файле driver.c. Таким образом, вызов может произойти повторно, что грозит ошибкой. Чтобы этого не случилось, в коде w_finish прежде всего проверяется, не был ли вызов повторным. Если в массиве все еще есть запросы, управление переходит в основную часть кода w_finish.

Так как ожидаемо получение значительного количества запросов, главная часть w_finish представляет собой цикл. Перед входом в цикл в переменную r помещается значение, сигнализирующее об ошибке, с целью принудительно провести повторную инициализацию контроллера. Структура command используется для передачи всех требуемых параметров в функцию, которая занимается действительным взаимодействием с контроллером диска. Параметр cmd.precomp необходим для некоторых дисков, чтобы компенсировать различие в производительности магнитного носителя и скорости, с которой поверхность диска проходит под магнитными головками, когда они перемещаются от внешних цилиндров к внутренним. Этот параметр для одного диска всегда принимает одно и то же значение, а многие диски его просто игнорируют. Параметр cmd.count хранит количество передаваемых секторов, усеченное до одного байта, так как именно такова разрядность всех командных регистров и регистров состояния контроллера. По-

сле заполнения этих полей переменной cmd следует условный оператор, обрамляющий код, задающий первый сектор, который будет передан. Сектор задается либо 28-битным логическим адресом (первая ветвь условного оператора), либо номерами цилиндра, дорожки и сектора (вторая ветвь). В обоих случаях используются одни и те же поля структуры cmd.

Наконец, загружается сама команда (чтение или запись) и, чтобы инициировать обмен данными, вызывается подпрограмма com_out. Этот вызов может потерпеть неудачу, если контроллер не готов принимать команды или не придет в готовность за заданное время ожидания. В таком случае увеличивается значение счетчика ошибок и, если оно достигнет значения MAX_ERRORS, операция прерывается. Иначе оператор языка C continue вызывает переход на начало цикла, то есть начинает повторную попытку.

После того как контроллер принял переданную в подпрограмму com_out команду, требуется ждать некоторое время до тех пор, пока данные не будут готовы. Поэтому (если команда была DEV_READ) вызывается функция w_intr_wait. Подробнее мы рассмотрим ее позже, а сейчас скажем лишь, что она вызывает receive, соответственно, в данной точке задача жесткого диска приостанавливается. Некоторое время спустя (задержка зависит от того, пришлось ли диску делать поиск цилиндра) подпрограмма w_intr_read завершается.

Данный драйвер не использует DMA, хотя многие контроллеры поддерживают такую возможность. Вместо этого драйвер применяет программный ввод/вывод. Если функция ожидания w_intr_wait не сообщила об ошибке, ассемблерная подпрограмма port_transfer считывает SECTOR_SIZE байтов из порта данных контроллера в буфер назначения, который должен находиться в кэше блоков файловой системы. Затем изменяются значения различных указателей и счетчиков, что регистрирует успешную передачу данных. Наконец, если количество байтов в текущем запросе подошло к 0, указатель на текущий запрос перемещается на следующую запись в массиве запросов.

В том случае, когда обрабатывается команда DEV_READ, первая часть кода, устанавливающая параметры команды и передающая ее в контроллер, выглядит точно так же, отличаясь лишь кодом операции. Но последующие действия в случае чтения другие. Прежде всего, ожидается, когда контроллер подаст сигнал о своей готовности принимать данные. Команда waitfor представляет собой макроопределение и обычно выполняется очень быстро. Подробнее мы расскажем о ней позже, сейчас скажем только, что waitfor в конечном итоге завершится, но долгие задержки должны быть исключительно редкими. Затем при помощи процедуры port_write данные записываются в порт данных контроллера, после чего делается вызов w_intr_wait и задача жесткого диска приостанавливается. Через какое-то время, когда произойдет прерывание и задача «проснется», производятся некоторые результирующие подсчеты.

Наконец, нужно разобраться с ошибками чтения или записи, которые могли произойти. Если контроллер уведомил драйвер, что причина ошибки в сбойном секторе, повторять то же самое смысла нет. Но для других типов ошибок стоит сделать некоторое количество повторов. Это количество определяется путем подсчета ошибок до достижения счетчиком значения MAX_ERRORS. Когда дости-

гается значение MAX_ERRORS/2, вызывается функция w_need_reset, принудительно инициализирующая контроллер перед следующей попыткой. Но, если запрос был необязательным (такие запросы имеют место при SCATTERED_IO), повторных попыток не делается.

Когда w_finish завершается, успешно или с ошибкой, в переменную w_command всегда помещается значение CMD_IDLE. Это позволяет другим функциям определить, что причиной ошибки является не механическое или электрическое повреждение диска, вызывающее прерывание при попытке выполнить операцию.

Контроллер диска управляется посредством набора регистров, которые на некоторых системах отображаются в память, но на IBM PC-совместимых системах находятся в отдельном адресном пространстве портов ввода/вывода. Список регистров, используемых стандартным AT-совместимым контроллером, приведен в табл. 3.6. Структура регистра № 6 расписана в табл. 3.7, где буквенные обозначения раскрываются так:

- LBA — для режима адресации цилиндр/головка/сектор (CHS) — 0. Для режима логической адресации блоков (LBA) — 1;
- D — для главного диска (master) — 0. Для подчиненного диска (slave) — 1;
- HSn — для режима CHS: выбор головки. Для режима LBA: 24–27 биты выбора блока.

Таблица 3.6. Регистры управления IDE-контроллером жесткого диска. Номера в скобках означают биты логического адреса блока, соответствующие каждому регистру в режиме LBA

Регистр	Чтение	Запись
0	Данные	Данные
1	Ошибка	Предкомпенсация записи
2	Количество секторов	Количество секторов
3	Номер сектора (0–7)	Номер сектора (0–7)
4	Цилиндр (младш.) (8–15)	Цилиндр (младш.) (8–15)
5	Цилиндр (старш.) (16–23)	Цилиндр (старш.) (16–23)
6	Выбор привода/головки (24–27)	Выбор привода/головки (24–27)
7	Состояние	Команда

Таблица 3.7. Поля регистра «Выбор привода/головки» (пояснения в тексте)

7	6	5	4	3	2	1	0
1	LBA	1	D	HS3	HS2	HS1	HS0

Это наша первая встреча с аппаратным обеспечением ввода/вывода и полезно напомнить, что порты ввода/вывода могут вести себя совершенно иначе, чем адреса памяти. Вообще говоря, входные и выходные регистры, которым соответствует один и тот же порт ввода/вывода, не обязательно совпадают. Таким образом, данные, записанные в некоторый порт, нельзя считать из него последующей операцией чтения. Например, последний регистр в табл. 3.6 показывает состоя-

ние контроллера диска, если считывать из него данные, а при записи в него данных передает команду контроллеру. Кроме того, часто сам факт записи данных в порт ввода/вывода приводит к выполнению некоторых действий, независимо от характера передаваемых данных. Например, это так для регистра команд AT-совместимого контроллера. При работе с ним параметры заносятся в младшие регистры, а в регистр команд помещается код операции. Сигналом начала операции является факт записи данных в регистр команд.

В дополнение, этот случай является иллюстрацией ситуации, когда назначение регистров зависит от режима работы. В приведенном в таблице примере режим выбирается шестым битом шестого регистра (см. табл. 3.7). Данные, записываемые или считываемые в регистры 3–5, а также младшие четыре бита регистра 6 по-разному интерпретируются в зависимости от значения бита LBA.

Теперь рассмотрим, как при помощи com_out команда передается контроллеру. Прежде чем менять какие-либо регистры, программа узнает, занят ли контроллер, считывая бит STATUS_BSY. Здесь важна скорость, и так как контроллер обычно должен быть свободен или станет свободен скоро, здесь используется активное ожидание. Поэтому в начале кода функции вызывается команда waitfor, тестирующая значение бита STATUS_BSY. Для повышения скорости эта команда реализована в виде макроса, который один раз проверяет значение бита, и только если контроллер занят, вызывает функцию ожидания. Это позволяет избежать накладных расходов на вызов функции в большинстве случаев. В тех редких ситуациях, когда контроллер занят, вызывается функция w_waitfor, в цикле проверяющая значение бита занятости до тех пор, пока контроллер не освободится или не истечет заданный интервал времени. В результате, если контроллер свободен, будет возвращено значение «истина» с минимальной задержкой. Если контроллер занят, будет возвращено значение «истина», когда удалось дождаться его освобождения, и «ложь» в другом случае. Подробнее мы расскажем об этом, когда займемся самой функцией w_waitfor.

Контроллер способен обслуживать более одного диска, поэтому, когда он освободится, в регистры записывается байт, выбирающий привод, головку и режим работы, и waitfor вызывается снова. Иногда приводу не удается выполнить команду или правильно вернуть код ошибки, в конце концов, это механическое устройство, которое может заесть или просто сломаться, и для страховки отправляется сообщение задаче таймера, с целью запланировать вызов подпрограммы, которая разблокирует драйвер. Следом за этим команда передается контроллеру, для чего сначала все параметры записываются в разные регистры, а затем в регистр команд кладется код самой команды. Последний шаг является критической секцией, поэтому он обрамлен вызовами процедур lock и unlock, которые, соответственно, запрещают и разрешают прерывания.

Следующие несколько функций, которые мы рассмотрим, меньше по объему. Мы уже отмечали, что w_finish вызывает w_need_reset, когда счетчик сбоев превышает половину от максимального числа попыток (MAX_ERRORS). Кроме того, та же функция вызывается в том случае, когда приходится ожидать готовности или прерывания диска. Работа функции w_need_reset сводится к тому, чтобы устано-

вить переменную state для каждого диска в массиве wini, чтобы при следующем обращении была повторена инициализация.

Функция w_do_close в отношении обычного жесткого диска очень проста. Если необходима поддержка CD-ROM или других приводов со сменными носителями, подпрограмма должна быть усложнена, в нее следует добавить код, отпирающий дверцу привода или выбрасывающий носитель, в зависимости от того, какие действия поддерживает оборудование.

Функция com_simple отправляет команду контроллеру и немедленно завершается, не выполняя обмен данными. В эту категорию попадают команды, которые служат для идентификации диска, установки некоторых параметров и повторной калибровки привода.

Когда com_out подготавливает задачу таймера к возможному сбою, она передает адрес подпрограммы w_timeout, чтобы задача таймера вызвала ее, когда истечет заданный период времени. Обычно диск успевает выполнить задание раньше, после чего присваивает переменной w_command значение CMD_IDLE. Обнаружив это значение, w_timeout определяет, что команда выполнена, и завершается. Если команда чтения или записи не успела завершиться, имеет смысл уменьшить объем запросов на чтение или запись данных. Это делается в два этапа, для чего количество запрашиваемых секторов уменьшается сначала до 8, затем до 1. Во всех случаях истечения времени ожидания на экран выводится сообщение и для повторной инициализации всех приводов вызывается функция w_need_reset. Кроме того, вызывается функция interrupt, которая отправляет сообщение задаче жесткого диска, имитируя аппаратное прерывание, которое должно произойти по завершении дисковой операции.

Когда требуется сброс, вызывается w_reset. Эта подпрограмма использует одну из функций, входящих в драйвер часов, milli_delay. Сначала w_reset делает начальную задержку, чтобы дать приводу время вернуться в исходное состояние после предыдущих действий. Затем *стробируется* бит регистра управления контроллера диска, то есть сначала на заданное время бит устанавливается в 1, а затем возвращается в 0. Далее вызывается waitfor, чтобы позволить диску прийти в состояние готовности. Если сброс не завершился успехом, на экран выводится сообщение и возвращается код ошибки. Сделавшей вызов программе остается решить, как в этом случае поступить.

Команды, затрагивающие обмен данными с диском, обычно завершаются генерацией прерывания, которое отправляет сообщение обратно драйверу. Фактически прерывание генерируется каждый раз, когда считывается или записывается сектор. Таким образом, после того как дана команда, всегда вызывается w_intr_wait. В свою очередь, эта подпрограмма в цикле обращается к receive, игнорируя содержимое сообщений. Цикл завершается, когда приходит сообщение, устанавливающее w_status в значение «не занято». По получении подобного сообщения проверяется состояние запроса. Это еще одна критическая секция, поэтому, чтобы гарантировать, что во время ее обработки не произойдут новые прерывания, используются команды lock и unlock.

Мы встретили несколько мест с подставленным макросом waitfor, означающим активное ожидание определенного значения бита занятости регистра

состояния контроллера диска. Этот макрос сначала проверяет бит и, если контроллер оказался занят, вызывает функцию w_waitfor. В свою очередь, она устанавливает таймер при помощи milli_start и входит в цикл, поочередно проверяющий регистр состояния и таймер. Если интервал времени истекает, вызывается w_need_reset, чтобы обеспечить сброс контроллера при следующем обращении к диску.

Параметр TIMEOUT, используемый функцией w_waitfor, равен 32 с. Аналогичный параметр, WAKEUP, нужный при планировании «пробуждения» драйвера по сигналу задачи часов, равен 31 с. Это очень большие периоды времени для активного ожидания, если сравнить их с обычным процессом, которому на работу дается 100 мс. Такие большие задержки обусловлены существующими стандартами для AT-совместимых дисков. Согласно этим стандартам, диску может потребоваться до 31 с, чтобы разогнаться. Конечно, на практике такое время требуется только в самом худшем случае, а разгон диска в большинстве систем происходит при включении питания или после длительных периодов бездействия. MINIX — развивающаяся система. Возможно, когда будет добавлена поддержка CD-ROM (или других устройств, у которых раскрутка происходит часто), возникнет необходимость в новом способе обработки задержек.

Функция w_handler играет роль обработчика прерываний. Адрес этой функции записывается функцией w_identify в таблицу дескрипторов прерываний (IDT) при первом запуске драйвера. Когда генерируется прерывание, регистр состояния контроллера диска копируется в переменную w_status, после чего вызывается функция ядра interrupt, чтобы повторно запланировать задачу жесткого диска. Конечно, в тот момент, когда приходит прерывание от жесткого диска, его задача уже находится в приостановленном состоянии в результате предшествующего вызова receive, который сделала подпрограмма w_intr_wait после инициирования дисковой операции.

Последняя функция в файле at_wini.c это w_geometry. Она возвращает максимальное значение номеров цилиндров, дорожек и секторов для выбранного жесткого диска. В отличие от RAM-диска, где эти значения имитировались, здесь в эти числа вложен реальный смысл.

3.7.5. Работа с дисководом для гибких дисков

Драйвер флоппи-дисковода (для гибких дисков) сложнее, чем драйвер жесткого диска, а его код больше. Это на первый взгляд парадоксально, так как механизмы дисковода для гибких дисков должны быть проще, чем механизмы винчестера, но контроллер первого проще и требует к себе больше внимания от операционной системы. Помимо того, некоторые сложности вносит сменный носитель. В текущем разделе мы рассмотрим некоторые вопросы, которые могут оказаться полезными для программиста, имеющего дело с гибкими дисками. В основных деталях код драйвера сходен с кодом драйвера жесткого диска. Детально же разбирать код, ввиду его сложности, мы не будем.

Одна из проблем, с которыми мы не столкнемся, работая с флоппи-приводами, — это необходимость поддерживать много типов контроллеров, что было

обязательно для жестких дисков. Хотя использующиеся сейчас гибкие диски с высокой плотностью записи не поддерживались оригинальными IBM PC, все контроллеры гибких дисков обслуживаются одним драйвером. Такое отличие от жестких дисков произошло, видимо, потому, что на дисководы гибких дисков отсутствует давление, заставляющее разработчиков повышать их производительность. Дискеты редко используются как рабочее хранилище в компьютерной системе, их скорость и емкость слишком ограничены по сравнению с винчестерами. Дискеты продолжают использоваться, но только для переноса небольших файлов, поэтому такими дисководами все еще оснащено большинство компьютеров.

Драйвер дисковода для гибких дисков не включает в себя сложных алгоритмов планирования, таких как SSF или элеваторная схема. Запросы выполняются строго последовательно, причем следующий запрос не принимается, пока не выполнен текущий. Изначально при разработке MINIX предполагалось, что эта система предназначена для персональных компьютеров, где большую часть времени будет активен только один процесс, и вероятность появления одного запроса во время обработки другого мала. Таким образом, поддержка очередей запросов не дала бы заметного увеличения производительности, зато потребовала бы значительного усложнения кода. Сейчас это тем более не имеет смысла, так как дискеты остаются нужны по большей части для переноса отдельных файлов с одной системы на другую.

Таким образом, хотя драйвер флоппи-дисковода и не поддерживает упорядочивание запросов, он должен, как и все драйверы блочных устройств, обеспечивать разрозненный ввод/вывод, поэтому он накапливает подобные запросы в массиве до тех пор, пока запрашиваются последовательные секторы. Но здесь массив запросов меньшего объема, чем у драйвера жесткого диска. Количество запросов в нем ограничено числом секторов в одной дорожке. Кроме того, драйвер учитывает флаг OPTIONAL и игнорирует необязательный запрос, если для его выполнения требуется переход на другую дорожку.

Простота устройства аппаратной части дисковода гибких дисков приводит к усложнению его драйвера. Использовать в дешевых, медленных и обладающих малой емкостью дисководах сложные контроллеры, входящие в современные жесткие диски, неоправданно. Вследствие чего программному обеспечению приходится явно учитывать все аспекты взаимодействия с диском. В качестве примера, показывающего сложность работы с флоппи-дисководом, рассмотрим процесс позиционирования магнитной головки на нужную дорожку при выполнении операции SEEK. Для жесткого диска драйверу вообще никогда не требуется явно вызывать SEEK. У жестких дисков номера цилиндров, головок и секторов, видимые программисту, могут не совпадать с реальными. Фактически реальная геометрия может быть весьма запутанной, например, на внешних цилиндрах может быть больше секторов, чем на внутренних. Тем не менее пользователь этого не замечает. Жесткие диски способны поддерживать логическую адресацию блоков (LBA), когда сектор указывается его абсолютным номером на диске, как альтернативу традиционной адресации «цилиндр/головка/сектор». Но даже при традиционной адресации можно использовать любую геометрию, раз контроллер

сам вычисляет, куда переместить головку и при необходимости выполняет операцию поиска.

Но для дискет операция SEEK требует явного программирования. Если команда SEEK завершилась неудачей, требуется вызвать подпрограмму, выполняющую операцию RECALBRATE, которая принудительно перемещает головку на нулевой цилиндр. Это позволяет контроллеру заново найти нужную дорожку, пошагово перемещая головку на известное количество дорожек. Конечно же, подобные действия необходимы и для жесткого диска, но контроллер жесткого диска обходится и без детальных инструкций драйвера.

Вот некоторые из особенностей дисководов для гибких дисков, которые усложняют драйвер.

1. Сменный носитель.
2. Различные форматы дисков.
3. Необходимость управления мотором.

Некоторые контроллеры жестких дисков предусматривают работу со сменным носителем (например, у CD-ROM), и обычно контроллер справляется с большинством сложностей без помощи драйвера. В случае дискетного привода встроенной поддержки нет, и это при том, что нужна она еще больше. Дискеты используются для переноса файлов, и часто бывает, что нужно менять дискеты, извлекая одну и вставляя следующую. Если данные, которые должны быть записаны на один диск, оказались на другом, неприятностей не избежать. Драйвер обязан пойти на все во избежание таких проблем, хотя это не всегда возможно, так как не все приводы позволяют определить, открывалась ли дверца с момента последнего обращения. Другая проблема, к которой приводит использование сменного носителя, — это обращение к приводу, в котором в текущий момент нет дискеты, что опасно зависанием системы. Такой проблемы можно избежать, если существует возможность определить, открыта ли дверца привода, но так как это не всегда гарантируется, необходимо предусмотреть прерывание операции по истечении интервала времени.

Сменный носитель по определению заменяется другим, причем в случае дискет носители могут иметь множество различных форматов. MINIX поддерживает как 3,5-дюймовые, так и 5,25-дюймовые диски, отформатированные с различной плотностью записи, от 360 Кбайт до 1,2 Мбайт (для 5-дюймовых дискет) или 1,44 Мбайт (для 3-дюймовых). MINIX работает с семью различными форматами. Существует два способа решения проблемы размножения форматов, и в MINIX реализованы оба. При первом подходе каждому формату соответствует отдельное устройство, с собственным младшим номером. Например, в MINIX вы можете увидеть 14 различных устройств, начиная с /dev/pc0, которому сопоставлен 5-дюймовый диск объемом 360 Кбайт, и заканчивая /dev/PS1, то есть 3-дюймовой дискетой на 1,44 Мбайт. Помнить все четырнадцать вариантов неудобно, поэтому реализован альтернативный метод. Когда к дисководу обращаются через файл /dev/fd0 или /dev/fd1 (второй дисковод), драйвер тестирует дискету, чтобы определить ее формат. У разных форматов разное число цилиндров и секторов, и чтобы определить формат, делается попытка прочитать последние секторы

и дорожки. Нужный вариант определяется путем исключения. Конечно, для этого требуется некоторое время, кроме того, диск, имеющий сбойные секторы или защиту от копирования, может быть определен неправильно.

Последняя сложность при работе с флоппи-дисководом — это управление двигателем. Считывать или записывать данные на дискету невозможно, если она не вращается. Жесткие диски разрабатываются так, чтобы работать тысячи часов без износа, но если мотор дисковода все время оставлять в движении, дискета быстро придет в негодность. Если при обращении к диску мотор оказался выключен, то, прежде чем пытаться считывать данные, необходимо подать команду для его включения, после чего подождать примерно полсекунды. На включение и выключение двигателя уходит много времени, поэтому MINIX оставляет двигатель работающим в течение еще нескольких секунд после обращения. Если за это время к диску не будет новых обращений, мотор отключается.

3.8. Часы

Часы (также называемые *таймерами*) необходимы для нормального функционирования любой системы с разделением времени. В частности, часы нужны для определения текущего времени, а также для того, чтобы предотвратить монополизацию процессора одним процессом. Программное обеспечение часов может иметь форму драйвера, хотя часы не являются ни блочным, ни символьным устройством. Наше изучение часов будет следовать тому же шаблону, что и в предыдущих разделах: сначала мы сделаем общий обзор аппаратного и программного обеспечения, а затем рассмотрим, как это реализовано в MINIX.

3.8.1. Аппаратное обеспечение часов

В компьютерах используются таймеры двух типов, и оба типа заметно отличаются от тех, которые используются людьми. Простейший таймер подключается к линии питания на 110 или 220 В и генерирует прерывания в каждый период колебания переменного напряжения, с частотой 50 или 60 Гц.

Второй тип часов состоит из трех компонентов: кварцевого резонатора, счетчика и регистра. Условная схема приведена на рис. 3.15. Кварцевый резонатор представляет собой особым образом укрепленный кусочек кристалла кварца, который может генерировать колебания с очень высокой стабильностью. Обычно частота колебаний резонатора находится в пределах от 5 до 100 МГц, в зависимости от параметров кристалла. В компьютере практически всегда есть как минимум один такой элемент, который генерирует синхросигнал для различных схем компьютера. Сигнал от резонатора передается на счетчик, отсчитывающий время в обратном направлении. Когда значение счетчика становится равным нулю, генерируется прерывание.

У программируемых часов обычно есть несколько режимов работы. В *режиме одноразового срабатывания* при запуске часов в счетчик помещается значение, сохраненное в регистре, которое затем уменьшается с каждым импульсом от ре-

зонатора. Когда значение счетчика достигает нуля, вырабатывается прерывание и таймер останавливается до следующего явного запуска. В *режиме прямоугольных импульсов*, после того как счетчик доходит до нуля и генерируется прерывание, значение из регистра вновь копируется в счетчик, и процесс повторяется до бесконечности. Эти периодические прерывания иногда называются «тиками».

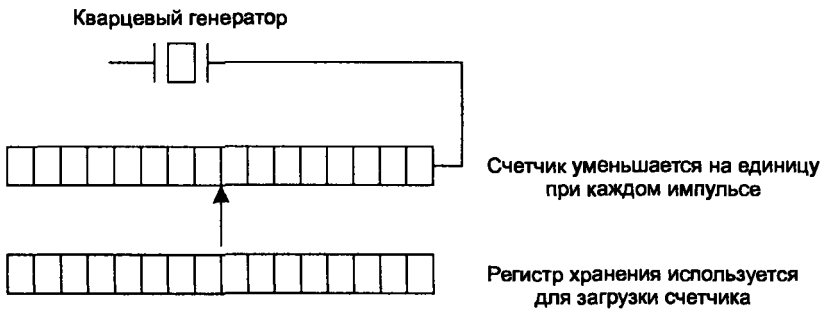


Рис. 3.15. Программируемые часы

Преимущество программируемых часов в том, что их частота может быть изменена программно. Если используется 1-мегагерцовый кристалл, импульсы от резонатора приходят каждую микросекунду. В случае 16-разрядного счетчика часы можно запрограммировать на генерацию прерываний с периодом от 1 мкс до 65 536 мс. Микросхемы программируемых часов обычно содержат две или три независимо программируемые схемы и имеют некоторые дополнительные настройки (например, отсчет вперед, а не назад, блокирование прерываний и пр.).

Чтобы компьютер не «забыл» текущее время при выключении питания, в большинство систем устанавливаются подпитываемые от батарейки запасные часы, на тех же низковольтных схемах, что и в наручных цифровых часах. Значение времени с этих вспомогательных часов может считываться при запуске системы. Если же таких часов нет, текущее время может запрашиваться у пользователя. Кроме того, существует стандартный протокол, который позволяет узнать текущее время с удаленной системы. Так или иначе, в MINIX, UNIX и некоторых других системах запрошенное время затем преобразуется в число тиков, прошедших с 12 часов дня 1 января 1970 года по всеобщему скоординированному времени (UTC, ранее говорилось — Гринвичское время). С каждым сигналом от часов значение времени увеличивается на единицу. Обычно имеются вспомогательные программы, которые позволяют вручную задать текущее время на системных и резервных часах, а также синхронизировать два таймера.

3.8.2. Программное обеспечение часов

Аппаратная часть часов только генерирует прерывания с заданным интервалом. Все остальное должно делаться программно, в драйвере часов. Функции драйвера часов во многом зависят от операционной системы, но чаще они включают в себя большую часть функций из следующего списка.

1. Поддержание текущего значения времени суток.
2. Предотвращение монополизации процессора одним процессом.
3. Подсчет полезной загрузки процессора.
4. Обработка системного вызова alarm, используемого пользовательскими процессами.
5. Выполнение функций сторожевого таймера для некоторых частей самой системы.
6. Профилирование, мониторинг и сбор статистики.

Первая из задач, поддержание текущего времени суток (также называемого *реальным временем*), не сложна. Для этого требуется только увеличивать значение счетчика времени по каждому сигналу от таймера, как говорилось ранее. Единственное, за чем нужно следить, — это число битов в счетчике времени. Если частота сигналов составляет 60 Гц, то 32-разрядный счетчик переполнится примерно за два года. Поэтому понятно, что в 32-битном счетчике нельзя хранить число тиков с 1.01.1970.

У этой проблемы три решения. Первое — использовать 64-битный счетчик, хотя это несколько усложнит поддержание времени, так как изменять его значение требуется много раз в секунду. Второй способ — хранить время в секундах, а не в тиках, задействуя для подсчета тиков в текущей секунде вспомогательный счетчик. Так как 2^{32} секунд составляет примерно 136 лет, этот метод вполне пригоден до XXII века.

Согласно третьему подходу время считается в тиках, но относительно момента загрузки системы, а не относительно какого-либо фиксированного момента в прошлом. Когда считывается значение вспомогательного таймера или пользователь сам вводит текущее время, вычисляется время загрузки системы и сохраняется в подходящей форме где-либо в памяти. Все три случая демонстрируются на рис. 3.16.

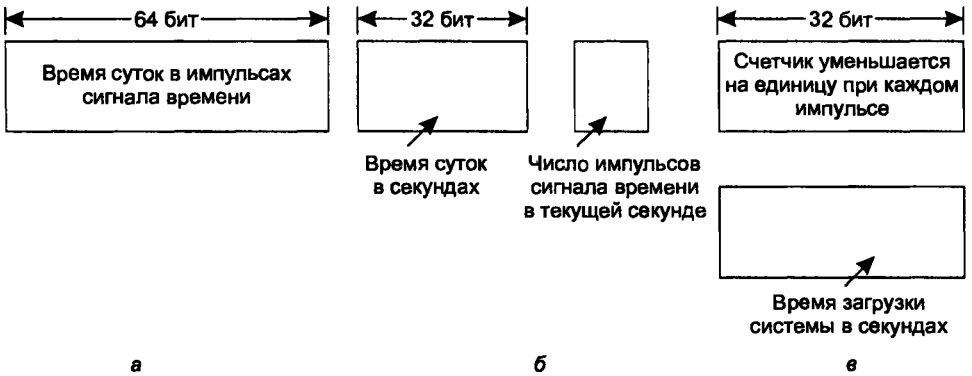


Рис. 3.16. Три способа хранения текущего времени: *а* — высокоемкий 64-битный счетчик; *б* — вспомогательный счетчик; *в* — подсчет от момента загрузки системы

Второе назначение таймера в том, чтобы не позволять процессу занимать процессор слишком долго. Когда процесс получает управление, планировщик инициализирует счетчик величиной кванта времени, выраженной в тиках. По каждому прерыванию от часов значение счетчика уменьшается на единицу. Когда счетчик достигает нуля, драйвер часов передает планировщику команду запланировать на выполнение следующий процесс.

Третья функция часов — это учет утилизации процессора. Самый точный способ — запускать при старте процесса отдельный счетчик. Когда процесс останавливается, значение этого счетчика скажет, сколько времени процесс работал. Чтобы время учитывалось правильно, значение второго счетчика необходимо сохранять в начале прерывания и восстанавливать при выходе из обработчика.

Менее точный, но гораздо более простой способ — использовать указатель на текущий процесс в таблице процессов, хранящийся в глобальной переменной. По каждому сигналу таймера увеличивается значение счетчика в ячейке текущего процесса. Таким образом, каждый такт таймера считается занятым текущим процессом в таблице процессов. Небольшая неувязка такой методики проявляется в той ситуации, когда процесс генерирует множество прерываний. В этом случае все равно будет считаться, что процесс получил полный «тик» времени, хотя реально он работал гораздо меньше. Но точный учет процессорного времени — слишком дорогостоящая операция, и практически никогда он не делается.

В MINIX и во многих других операционных системах процесс может попросить систему передать ему предупредительный сигнал с некоторой заданной задержкой. Обычно в качестве такового используется сигнал ОС, сообщение или что-либо подобное. Примером приложения, которому такое может потребоваться, служит сетевой интерфейс. Здесь необходимо повторно передавать пакет данных, если за некоторое время не поступило подтверждение его получения. Еще один пример — программа-тренажер, где обучаемому дается ограниченное время на ответ.

Если драйвер часов поддерживает достаточное количество таймеров, он может для каждого запроса запускать отдельный таймер. Обычно это не так, и драйвер должен имитировать несколько виртуальных таймеров, имея за душой только один физический. Один из способов опирается на таблицу, в которой хранится время подачи сигнала для всех текущих таймеров, а также на переменную, задающую время наступления следующего срабатывания. Каждый раз, когда обновляется текущее время, драйвер проверяет ближайший по времени сигнал. Если его время пришло, сигнал отправляется и драйвер переходит к следующему.

Когда ожидается много сигналов, более эффективно имитировать несколько таймеров при помощи единой очереди запросов, отсортированной по времени. Очередь может быть организована в виде списка, как показано на рис. 3.17. Каждая запись в этой очереди хранит число тиков между сигналом в текущей записи и предыдущим. В частности, показанная на рисунке ситуация соответствует сигналам, ожидаемым в моменты времени 4203, 4207, 4213, 4215 и 4216.

Здесь первое прерывание произойдет через 3 такта таймера. С каждым тиком таймера значение переменной «следующий сигнал» уменьшается на единицу. Когда оно достигает нуля, генерируется первый сигнал из очереди и первая за-

пись исключается, после чего в переменную помещается время из следующей записи в очереди (в данном случае это время равно 4).

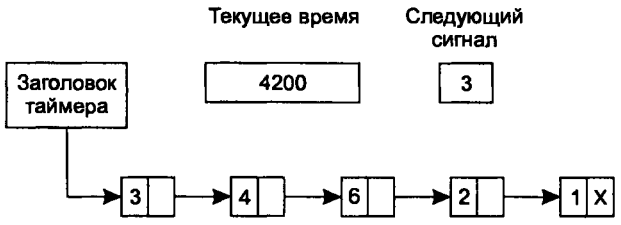


Рис. 3.17. Имитация нескольких таймеров на одних часах

Обратите внимание, что при возникновении каждого прерывания часов драйвер должен выполнить несколько действий: увеличить значение реального времени, уменьшить счетчик времени кванта и проверить, равен ли он 0, учесть время работы процессора и уменьшить значение времени сигнального таймера. Все эти действия должны производиться много раз в секунду, поэтому они тщательно программируются в расчете на максимальное быстродействие.

Некоторым частям операционной системы также необходимо устанавливать таймеры. Это так называемые *сторожевые таймеры*. Изучая драйвер жесткого диска, мы видели, что каждый раз, когда контроллеру отдается команда, запускается таймер как средство выполнения некоторых действий, если команда не будет выполнена. Кроме того, мы упоминали, что драйвер флоппи-дисковода должен дождаться, пока раскрутится двигатель, а также должен выключить его, если в течение некоторого времени не было сделано ни одного обращения. Существуют принтеры, которые умеют печатать 120 символов/с (8,3 мс/символ), но даже они не в силах выполнить возврат каретки за 8,3 мс, поэтому драйвер принтера должен приостановиться после передачи символа возврата каретки.

Для создания сторожевых таймеров драйвером часов используется тот же механизм, что и для работы с пользовательскими сигналами. Единственное различие в том, что вместо подачи сигнала драйвер часов вызывает подпрограмму обратной связи, адрес которой ему передается при вызове. Эта процедура является частью кода, сделавшего вызов, но благодаря тому, что все драйверы разделяют общее адресное пространство, драйвер часов все равно может обратиться к указанной подпрограмме. Вызываемая подпрограмма затем может выполнить любые необходимые действия, в том числе и возбудить прерывание, хотя на уровне ядра работа с прерываниями неудобна, а сигналы не существуют. Именно поэтому и реализован отдельный механизм сторожевых таймеров.

Последний пункт в списке функций драйвера часов — это *профилирование*. Некоторые операционные системы предоставляют механизм, благодаря которому пользовательские программы могут получить от системы гистограмму, показывающую, где процессор провел больше времени. Если профилирование поддерживается, то по каждому сигналу таймера проверяется, является ли текущий процесс профилируемым. Если это так, определяется, в каком из диапазонов

адресов находится счетчик команд, и значение времени для этого диапазона увеличивается на единицу. Этот же механизм пригоден и для профилирования самой системы.

3.8.3. Обзор драйвера часов в MINIX

Код драйвера часов в MINIX находится в файле clock.c. Задача часов может получать шесть типов сообщений с указанными параметрами.

1. HARD_INT
2. GET_UPTIME
3. GET_TIME
4. SET_TIME (новое значение времени в секундах)
5. SET_ALARM (номер процесса, вызываемая подпрограмма, величина задержки)
6. SET_SYN_AL (номер процесса, величина задержки)

Сообщение HARD_INT отправляется драйверу тогда, когда возбуждается аппаратное прерывание и необходимо выполнить некоторую работу, например нужно послать предупредительный сигнал процессу или процесс работает чересчур долго.

Сообщение GET_UPTIME используется для определения времени, прошедшего с момента загрузки системы. GET_TIME возвращает количество секунд, отсчитанных с 12 часов дня 1.01.1970, а SET_TIME устанавливает текущее время. Последнее сообщение вправе отправлять только суперпользователь.

Внутреннее представление времени в драйвере часов соответствует схеме на рис. 3.16, *в*. Когда задается текущее время, драйвер вычисляет, когда была загружена система. Это вычисление возможно благодаря тому, что драйвер знает текущее реальное время, а также знает продолжительность работы системы (в тиках). Реальное время загрузки системы сохраняется в отдельной переменной. Позже, при вызове GET_TIME, текущее время работы системы переводится из тиков в секунды и суммируется со значением времени, сохраненным в переменной при загрузке.

Вызов SET_ALARM позволяет процессу запустить таймер, который сработает через указанное время (в тиках). Когда пользовательский процесс делает системный вызов alarm, то системный вызов отправляет сообщение менеджеру памяти, который, в свою очередь, отправляет сообщение с информацией о запросе драйверу часов. По исчерпании указанного промежутка времени драйвер часов посылает сообщение обратно менеджеру памяти, который уже заботится о том, чтобы передать сигнал процессу.

Сообщение SET_ALARM используется также теми задачами, которым необходимо установить сторожевой таймер. В этом случае, когда истекает указанное время, просто вызывается заранее указанная процедура. Сам драйвер часов не имеет представления о том, что делает эта процедура.

Вызов SET_SYN_AL подобен вызову SET_ALARM, за исключением того, что он устанавливает *синхронный сигнальный таймер*. Этот вариант отличается тем, что

процессу отправляется сообщение вместо передачи ему сигнала или вызова подпрограммы. Задача синхронного сигнального таймера отвечает за диспетчеризацию сообщений процессам-потребителям. Подробно синхронные таймеры будут обсуждаться позже.

В задаче часов нет никаких значительных структур данных, но она поддерживает несколько локальных переменных, помогающих отслеживать время. Только одна из этих переменных, lost_ticks, — глобальная, она объявлена в файле glo.h. Сейчас она не используется. Возможно, в будущем, если в систему будет добавлен драйвер, который может надолго заблокировать прерывания и потерять несколько тактов таймера, она и появится. Если такой драйвер действительно будет написан, программист может увеличивать значение lost_ticks, чтобы компенсировать время, в течение которого прерывания были запрещены.

Очевидно, прерывания часов происходят очень часто и необходимо обрабатывать их как можно быстрее. Поэтому в MINIX обработчик прерываний по большей части выполняет минимальное количество действий. Получив управление, обработчик помещает в локальную переменную ticks значение lost_ticks + 1, после чего опирается на полученное значение при подсчете времени занятости процессора, обновляет значение переменной pending_ticks, а затем сбрасывает переменную lost_ticks в 0. Переменная pending_ticks объявлена как PRIVATE, ее объявление не лежит внутри какой-либо функции, но она видима только в пределах файла clock.c. Еще одна переменная, sched_ticks, также объявлена как PRIVATE. Значение этой переменной уменьшается на единицу с каждый сигналом часов, с целью отслеживать время выполнения процесса. Обработчик отправляет сообщение менеджеру памяти только тогда, когда сработал таймер или истек отведенный процессу квант времени. Благодаря такой схеме в большинстве случаев обработчик прерываний практически сразу завершается.

Когда задача часов получает какое-либо сообщение, она прибавляет значение pending_ticks к переменной realtime, после чего обнуляет pending_ticks. Переменная realtime, вместе с boot_time, позволяет вычислить текущее системное время. Обе эти переменные объявлены как PRIVATE, поэтому остальные части системы в состоянии определить время только отправляя сообщения задаче часов. Несмотря на то что в любой отдельный момент значение переменной realtime может быть неточным, механизм вычисления времени обеспечивает, что при каждом обращении вычисляется точное значение. Другими словами, каждый раз, когда вы глядите на часы, вы видите правильное время, хотя, когда вы их не видите, время может быть неточным.

Для поддержки таймеров в переменную next_alarm записывается время, когда должен сработать ближайший обычный или сторожевой таймер. Обслуживая таймеры, необходимо соблюдать осторожность, так как в момент срабатывания запросивший услугу процесс может быть уже завершен. Поэтому, когда наступает пора отправить сигнал, сначала делается проверка, действительно ли сигнал все еще нужен. Если нужда в нем отпала, никаких действий не выполняется.

Каждому пользовательскому процессу разрешается устанавливать только один сигнальный таймер. Если процесс делает системный вызов alarm уже после запуска им другого таймера, предыдущий таймер отменяется. Таким образом, информа-

цию о таймере удобно хранить в таблице процессов, выделяя под нее одно слово в каждой из записей. Для задач при срабатывании таймера должна вызываться некоторая функция, адрес которой должен где-то храниться. Для этой цели предназначен массив watch_dog. Аналогичный массив, syn_table, хранит флаги, отмечающие, установлен ли для процесса синхронный сигнальный таймер.

Общая логика драйвера часов следует той же модели, что и для драйвера диска. Главная программа драйвера представляет собой бесконечный цикл, в котором сообщения принимаются, обрабатываются и отправляются ответные сообщения (кроме случая CLOCK_TICK). Для обработки каждого типа сообщений предусмотрена отдельная процедура. Все эти процедуры, в соответствие со стандартным соглашением об именовании, имеют названия вида do_xxx, где вместо xxx подставляется название действия. К сожалению, некоторые компоновщики усекают имена подпрограмм до семи или восьми символов, поэтому имена do_set_alarm и do_set_time потенциально конфликтны. В силу чего выбрано название do_setalarm. Подобная проблема иногда встречается в MINIX и решается обычно путем корректирования одного из имен.

Задача синхронного сигнального таймера

В этом разделе мы будем изучать еще одну задачу — задачу *синхронного сигнального таймера*. Такой таймер аналогичен обычному, за исключением того, что он в заданное время отправляет процессу сообщение, а не вызывает сигнал или выполняет указанную подпрограмму, как это делает обычный таймер. Сигнал или вызов подпрограммы от обычного таймера могут произойти в любой момент работы процесса-инициатора вызова, в то время как сообщение может быть получено только тогда, когда процесс сделает вызов receive. Именно поэтому таймер, работающий с сообщениями, назван синхронным.

Синхронный таймер был добавлен в MINIX для поддержки сетевого сервера, который, как и менеджер памяти или файловая система, работает как отдельный процесс. Часто возникает потребность ограничить время, в течение которого процесс может находиться в заблокированном состоянии, ожидая ввода. Например, при работе с сетью отсутствие подтверждения получения пакета в течение определенного отрезка времени означает сбой при передаче данных. Перед тем как пытаться получить сообщение (это блокирует процесс), сетевой сервер устанавливает синхронный таймер. Так как при срабатывании таймер отправляет процессу сообщение, процесс в некоторый момент обязательно будет разблокирован. Если процесс получает сообщение, то он прежде всего сбрасывает таймер. После этого, проанализировав тип или отправителя полученного сообщения, сервер может определить, прибыл ли пакет или же истекло время ожидания. В последнем случае сервер может сделать попытку исправить ошибку, обычно — повторно отправив пакет.

Отправка сообщения синхронным таймером обходится дешевле, чем подача сигнала обычным, так как последнему требуется передавать несколько сообщений и производится значительное количество дополнительных действий. Вызов функции сторожевым таймером происходит быстро, но пригоден только для задач, которые находятся в общем адресном пространстве с ядром. Когда процесс

ожидает сообщения, использовать синхронный таймер проще и удобнее, нежели чем сигналы или вызов функции сторожевым таймером. С такими сообщениями легко работать, и они не требуют большого количества вспомогательных вычислений.

Обработчик прерываний часов

Как уже описывалось ранее, значение переменной realtime обновляется не сразу после возбуждения прерывания часов. Подпрограмма, обслуживающая прерывание, обновляет значение переменой pending_ticks, а также выполняет некоторые дополнительные простые действия, такие как учет процессорного времени или же уменьшение значения счетчика кванта времени. Сообщение отправляется задаче таймера только в том случае, если требуются более сложные операции. Такое поведение является компромиссом между идеей о том, что задачи в MINIX должны взаимодействовать исключительно при помощи сообщений, и пониманием того, что на практике обработка каждого сигнала таймера потребляет процессорное время. Практика показала, что на медленных машинах переход от реализации, где сообщение отправляется задаче таймера по каждому прерыванию, к описанной выше реализации может привести к повышению производительности до 15 %.

Миллисекундные задержки

Как еще одна уступка реальности, в файле clock.c имеется несколько подпрограмм, которые позволяют задавать задержки с миллисекундным разрешением. Такие небольшие задержки необходимы для работы с некоторыми устройствами ввода/вывода. Реализовать их при помощи таймеров и механизма передачи сигналов практически невозможно. Находящиеся в этом файле функции должны вызываться задачей напрямую. В их основе старейший и простейший механизм: непрерывный опрос. Непрерывно, с максимально возможной скоростью считывается значение счетчика, используемого для генерации прерываний, и считанное значение преобразуется в миллисекунды. Это делается до тех пор, пока не истечет требуемое время.

Резюме

Таблица 3.8 резюмирует различные сервисы, предоставляемые кодом в файле clock.c. Существует несколько путей доступа к таймеру и несколько вариантов выполнения запроса. Некоторые из сервисов доступны для любого процесса, эти сервисы возвращают результат выполнения запроса при помощи сообщения.

Код ядра может запросить текущее время работы системы посредством вызова функции, избегая тем самым лишних затрат на передачу сообщения. Пользовательский процесс может установить таймер, в результате срабатывания которого процесс получит сигнал. Задача также может запросить сторожевой таймер, который вызовет некоторую функцию. Ни один из этих механизмов не может быть использован сервером, но сервер может запросить синхронный сигнальный таймер. Кроме того, код задач или ядра может запросить задержку миллисекундной длительности при помощи функции milli_delay или же встроить в код проце-

дуры ожидания вызовы milli_elapsed, например, во время ожидания при считывании данных из порта.

Таблица 3.8. Сервисы, предоставляемые программным обеспечением таймера

Сервис	Способ доступа	Ответ	Клиенты
Gettime	Системный вызов	Сообщение	Любой процесс
Uptime	Системный вызов	Сообщение	Любой процесс
Uptime	Вызов функции	Значение, возвращенное функцией	Ядро или задача
Alarm	Системный вызов	Сигнал	Любой процесс
Alarm	Системный вызов	Активизация сторожевой функции	Задача
Synchronous alarm	Системный вызов	Сообщение	Сервер
Milli_delay	Вызов функции	Активное ожидание	Ядро или задача
Milli_elapsed	Вызов функции	Значение, возвращенное функцией	Ядро или задача

3.8.4. Реализация драйвера часов в MINIX

При запуске MINIX вызываются все имеющиеся в системе драйверы. Большинство из них просто делают попытку получить сообщение и блокируются. Драйвер часов, clock_task, также делает такой вызов, но перед этим он вызывает подпрограмму init_clock, настраивая программируемый таймер на работу с частотой 60 Гц. Когда драйвер получает какое бы то ни было сообщение, он прежде всего добавляет значение pending_ticks к переменной realtime и сбрасывает переменную pending_ticks. Это действие потенциально конфликтно с прерыванием таймера, поэтому оно обрамлено вызовами lock и unlock, предотвращающими возможную ситуацию состязания. Главный цикл у драйвера часов точно тот же, что и остальных драйверов: принимается сообщение, вызывается обслуживающая его функция и генерируется ответное сообщение.

Функция do_clocktick в действительности не вызывается при каждом сигнале таймера, как можно было бы подумать, глядя на ее название. Она вызывается тогда, когда обработчик прерываний определил, что необходимо выполнить некоторые важные действия. Сначала эта функция проверяет, не сработал ли таймер (сигнальный или сторожевой). Если это так, в таблице процессов просматриваются поля с информацией о запущенном таймере. За один проход по таблице может потребоваться обслужить несколько таймеров. Когда обнаруживается процесс, время срабатывания таймера у которого меньше текущего времени или равно ему, но не равно нулю, проверяется соответствующий этому процессу элемент массива watch_dog. Если эта ячейка содержит адрес функции, значит, процесс затребовал сторожевой таймер, и функция вызывается. Для проверки значения указателя привлекается тот факт, что в языке С числовое значение одновременно может интерпретироваться и как логическое. Если в таблице watch_dog обнаружен пустой указатель (в С ему соответствует нулевое значение,

интерпретируемое в логическом контексте как ложь), значит, процесс затребовал обычный таймер, поэтому вызывается функция cause_sig, передающая процессу сигнал SIGALARM. Запись в массиве watch_dog используется и тогда, когда сработал синхронный сигнальный таймер. В этом случае в массив вместо адреса сторожевой функции помещается адрес функции cause_alarm. Такой же подход неплох и для обычного таймера, только уже с адресом функции cause_sig, но тогда нам бы пришлось переписать функцию cause_sig так, чтобы она принимала аргументы, а номер процесса определяла бы из глобальной переменной. Или же переписать остальные функции так, чтобы они получали ненужные им аргументы.

Мы обсудим функцию cause_sig, когда станем изучать системную задачу в следующем разделе. Назначение этой функции — посылать сообщение менеджеру памяти. Здесь сначала нужно определить, ожидает ли менеджер памяти сообщение. Если да, то сообщение о срабатывании таймера отправляется, в противном случае делается отметка передать сообщение при первой оказии.

При переборе записей в таблице процессов обновляется значение переменной next_alarm. Перед началом цикла в эту переменную помещается очень большое значение, после чего для каждого процесса, у которого время срабатывания таймера (поле p_alarm) не равно нулю, сравнивается значение p_alarm и next_alarm. При этом в next_alarm помещается меньшее из двух значений.

Закончив облуживание таймеров, do_clocktick приступает к проверке, не подошло ли время запланировать на выполнение следующий процесс. Время кванта выполнения хранится в переменой sched_time, объявленной как PRIVATE. Обычно значение этой переменной декрементируется с каждым тиком таймера. В тех случаях, когда активизируется do_clocktick, обработчик прерывания сам не уменьшает значение переменной, позволяя сделать это функции do_clocktick, которая также проверяет, стало ли значение равно нулю. Значение sched_ticks не сбрасывается при запуске нового процесса (поскольку менеджер памяти и файловая система должны продолжать работать), вместо этого значение сбрасывается через каждые SCHED_RATE тиков. Перед тем как отобрать у текущего процесса процессор, проверяется, что он выполнялся в течение хотя бы одного полного такта планировщика.

Следующая подпрограмма, do_getuptime, занимает всего одну строчку. Она копирует текущее значение переменной realtime (количество тактов таймера с момента загрузки системы) в соответствующее поле сообщения. Этот метод позволяет любому процессу получить информацию о времени работы системы, но для задач издержки на передачу сообщения чересчур велики, поэтому имеется соответствующая функция, get_uptime, к которой задачи могут обращаться напрямую. Так как эта функция вызывается не через сообщение для задачи часов, ей приходится самостоятельно добавлять к значению realtime количество отложенных тиков. Чтобы в то время, когда делается обращение к переменной pending_ticks, не произошло прерываний таймера, оно обрамлено вызовами lock и unlock.

Функция do_get_time вычисляет текущее значение времени исходя из значений realtime и boot_time (момент загрузки системы в секундах). Функция do_set_time имеет обратное назначение. Она вычисляет новое значение boot_time, основываясь на переданном ей значении реального времени и времени работы системы.

Следующие две подпрограммы, do_setalarm и do_setsyn_alarm, настолько похожи, что мы будем рассматривать их вместе. Обе они сначала извлекают из сообщения информацию о процессе, которому будет передан сигнал, а также время задержки. Затем do_setalarm извлекает адрес функции, которую необходимо вызвать, а несколькими строками далее этот адрес заменяется нулевым указателем, если целевой процесс является пользовательским процессом, а не задачей. Как мы уже видели ранее, подпрограмма do_clocktick сравнивает значение данного указателя с нулем и тем самым определяет, нужно ли послать целевому процессу сигнал или вызвать сторожевую функцию у задачи. Далее обе функции вычисляют оставшееся до срабатывания таймера время (в секундах) и записывают его в ответное сообщение. В завершение обе функции вызывают common_setalarm. При этом do_setsyn_alarm всегда передает common_setalarm значение cause_alarm.

Функция common_setalarm завершает работу, начатую одной из двух только что описанных функций. Затем она записывает время срабатывания таймера в таблицу процессов, а в массив watch_dog заносит адрес сторожевой функции (этот адрес также может быть равен нулю или быть ссылкой на функцию cause_alarm). Далее функция сверяет все записи в таблице процессов, чтобы определить, какой таймер сработает следующим, точно так же, как это делается в do_clocktick.

Код cause_alarm несложен. В элемент массива syn_table, соответствующий целевому процессу, помещается значение TRUE. Кроме того, если задача синхронного сигнального таймера не активна, ей отправляется сообщение, чтобы она «проснулась».

Реализация задачи синхронного сигнального таймера

Структура задачи синхронного сигнального таймера, syn_alarm_task, следует общей структуре всех задач. После инициализации она переходит в бесконечный цикл, принимающий и отправляющий сообщения. Инициализация сводится к тому, что задача заявляет о своей активности, записывая в переменную syn_al_alive значение TRUE, а после сообщает о том, что ей нечем заняться, обнуляя (FALSE) все элементы массива syn_table. В этой таблице имеются ячейки для всех процессов.

Внешний цикл программы начинается с того, что функция декларирует, что ее работа завершена, после чего управление передается во внутренний цикл. В этом цикле проверяются все записи в массиве syn_table. Если обнаруживается процесс, ожидающий сигнала синхронного таймера, соответствующая ему запись сбрасывается, соответствующему процессу отправляется сигнал CLOCK_INT и объявляется, что еще есть поле деятельности. Завершив внутренний цикл, программа не останавливается, ожидая новых сообщений, если не установлен флаг work_done (работа завершена). Если для задачи имеется работа, передавать ей сообщения не нужно, так как cause_alarm обращается к таблице syn_alarm напрямую. В результате, пока имеются сигналы, ожидающие доставки, внешний цикл программы очень быстро проворачивается несколько раз.

Фактически эта задача не используется в обычном установочном пакете MINIX. Она будет использоваться сетевым сервером только в том случае, если перекомпилировать ядро, включив поддержку сети. Сетевой сервер нуждается именно в этом

механизме потому, чтобы выжидать небольшое время, если ожидаемый пакет не принимается. Кроме того, серверу нельзя послать сигнал, так как сервер должен работать все время, а большинство сигналов по умолчанию «убивают» процесс.

Реализация обработчика прерываний часов

Код обработчика прерываний часов — пример компромисса между потребностью делать как можно меньше (ради минимизации времени обработки) и необходимостью изредка выполнять достаточно сложные действия. Обработчик изменяет значения небольшого количества переменных и проверяет значения некоторых других. Код clock_handler начинается с того, что делается учет процессорного времени. В MINIX отслеживается как процессорное время пользовательских процессов, так и время системы. Текущее время отдается пользовательскому процессу, если он исполнялся в момент возникновения прерывания. Системное время соответствует случаям, когда исполнялась файловая система или менеджер памяти. Ссылка на последний запланированный пользовательский процесс (два сервера не учитываются) всегда хранится в переменной bill_ptr. После того как завершены действия по учету процессорного времени, увеличивается значение одной из основной переменных, pending_ticks. Чтобы узнать, нужно ли пробудить терминал или послать сообщение задаче часов, необходимо знать реальное время, но каждый раз обновлять значение переменной realtime — дорогостоящая операция, так как обращение к этой переменной производится через блокировки. Чтобы избежать накладных расходов, обработчик вычисляет собственную версию реального времени в локальной переменной now. Существует небольшая вероятность погрешности в значении этой переменной, но у подобной ошибки не будет серьезных последствий.

Оставшаяся часть кода обработчика зависит от проверок различных условий. Как терминал, так и принтер нуждаются в том, чтобы время от времени получать управление. Для терминала предусмотрена глобальная переменная tty_timeout, поддерживаемая задачей терминала. Она хранит время, соответствующее моменту получения управления задачей терминала в следующий раз. Для принтера нужно проверить значения нескольких переменных, объявленных в коде модуля принтера как PRIVATE, для этого служит функция pr_restart, которая быстро завершается даже в худшем случае, если принтер не отвечает. Далее делается проверка, не имеется ли сработавших таймеров и не завершился ли квант времени, при выполнении этого условия активизируется задача часов. Последнее условие — сложное, оно является логической конъюнкцией трех более простых условий. Следующий далее код

```
interrupt(CLOCK);
```

приводит к тому, что задаче часов отправляется сообщение HARD_INT.

Обсуждая работу do_clocktick, мы упомянули, что функция уменьшает значение переменной sched_ticks и проверяет, не обнулилась ли она, то есть истек ли квант времени. В упомянутом выше сложном составном условии проверяется равенство sched_ticks единице. Если задача часов еще не активизирована, необходимо уменьшить на единицу значение sched_ticks, и если оно стало нулевым,

завершить квант. Если это произошло, следует сохранить информацию о том, что текущий процесс был активен в прошедшем кванте. Для чего в переменную prev_ptr копируется текущее значение bill_ptr.

Утилиты для работы со временем

Помимо прочего, в файле clock.c есть несколько вспомогательных функций. Многие из этих функций аппаратно-зависимые, и при переносе системы на другую аппаратную платформу их необходимо переписать. Мы рассмотрим только назначение этих функций, не вдаваясь в детали их работы.

Функция init_clock вызывается в начале работы задачи часов clock_task. Она устанавливает режим работы и период программируемых часов так, чтобы они «тикали» 60 раз в секунду. Несмотря на то что «скорость процессора» увеличилась с 4,77 МГц у оригинальных IBM PC до сотен мегагерц на современных системах, для программирования таймера используется одно и то же значение константы TIMER_COUNT, вне зависимости от модели компьютера, на котором работает MINIX. У каждого IBM-совместимого компьютера, каким бы быстрым ни был его процессор, для управления различными устройствами, оперирующими временем, имеется сигнал с частотой 14,3 МГц. Примерами устройств, которым необходим таймер, являются последовательный интерфейс и экран.

Функция clock_stop дополняет init_clock. Эта функция не очень нужна, она используется потому, что пользователям MINIX иногда может потребоваться запускать другие операционные системы. Эта функция просто сбрасывает параметр микросхемы таймера, переводя ее в режим работы по умолчанию, как этого могут ожидать MS-DOS и некоторые другие операционные системы.

Функция milli_delay необходима для задач, которым требуются задержки миллисекундной величины. Она написана на языке C и не содержит аппаратно-специфичного кода, но в ее коде вы увидите технику, которую могли бы ожидать только в низкоуровневых ассемблерных процедурах. Функция инициализирует переменную-счетчик нулем, после чего быстро, в непрерывном цикле, проверяет эту переменную до тех пор, пока она не достигнет нужного значения. Во второй главе мы говорили о том, что такая методика, называемая активным ожиданием, по возможности должна избегаться, но правила не бывают без исключений, это жизнь. За инициализацию счетчика отвечает следующая функция, milli_start, которая просто обнуляет значения двух переменных. Опрос значения счетчика производится при помощи последней функции в файле, milli_elapsed. Она обращается к аппаратному обеспечению часов. Проверяется значение того же самого счетчика, который используется для обратного отсчета тиков таймера, то есть тот, что может дойти до нуля или принять максимальное значение до того, как закончится требуемый временной интервал. Функция milli_elapsed корректирует это поведение.

3.9. Терминалы

Каждый многоцелевой компьютер оснащен одним или несколькими терминалами, при помощи которых он взаимодействует с пользователями. Существует

чрезвычайно большое количество различных форм терминалов. Поэтому на плечи драйвера терминала ложится задача скрыть эти различия, чтобы не переписывать аппаратно-независимую часть системы и пользовательские программы для каждого нового типа терминала. В подразделах ниже мы последуем нашему стандартному пути и сначала обсудим аппаратное и программное обеспечение терминалов в общем и целом, а затем перейдем к подробностям их реализации в MINIX.

3.9.1. Аппаратное обеспечение терминала

С точки зрения операционной системы, терминалы можно разбить на три широкие категории, в зависимости от того, как ОС взаимодействует с ними. Первую категорию представляют отображаемые в память терминалы, состоящие из клавиатуры и экрана, подключенных к компьютеру. Во вторую группу входят терминалы, подключаемые через последовательную линию передачи данных стандарта RS-232, чаще всего через модем. Третью группу образуют терминалы, подключаемые через сеть. Таксономия терминалов иллюстрируется рис. 3.18.

Отображаемые в память терминалы

Первая широкая категория терминалов на рис. 3.18 — это отображаемые в память терминалы. Такие терминалы представляют собой составную часть самого компьютера. Доступ к такому терминалу происходит посредством специальной области памяти, называемой *видеопамятью*, которая входит в общее адресное пространство компьютера и адресуется так же, как и остальная память (рис. 3.19).

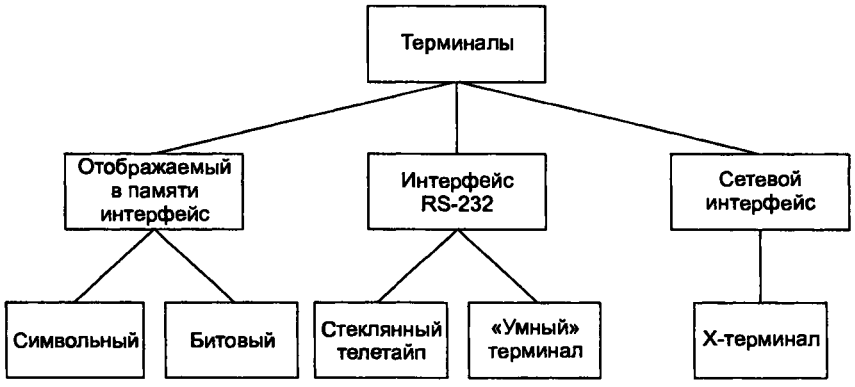


Рис. 3.18. Типы терминалов

На видеокарте имеется также микросхема, называемая *видеоконтроллером*. На ее вход подаются коды символов, а на выходе она генерирует видеосигнал, который воспринимается монитором. Электронно-лучевая трубка монитора генерирует электронный луч, который горизонтально сканирует экран, рисуя на нем линии. Обычно экран имеет разрешение от 480 до 1024 линий по вертикали

и от 640 до 1200 точек в линии. Эти точки называют *пикселами*. Сигнал от видеоконтроллера модулирует электронный луч и определяет, будет ли пиксел светлым или темным. У цветных мониторов есть три независимо модулируемых луча, отдельно для красного, зеленого и синего цветов.

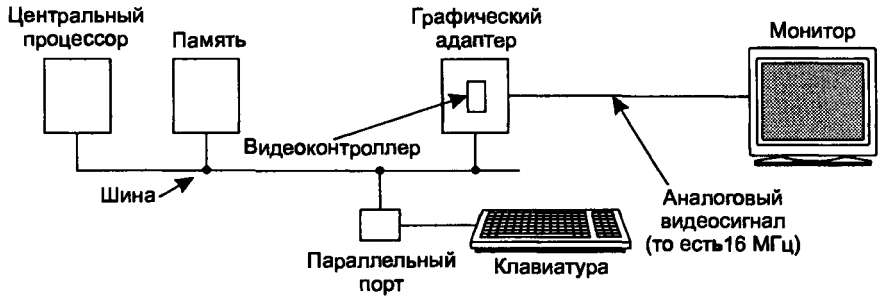


Рис. 3.19. Отображаемые в память терминалы, доступ через видеопамять

У простого монохромного монитора каждый символ (включая межсимвольный промежуток) помещается в прямоугольник 9 пикселов в ширину и 14 в высоту, всего на экране получается 25 строк по 80 символов в каждой. При этом дисплей должен иметь разрешение 350 растровых строк по 720 пикселов в каждой. Каждый кадр обновляется от 45 до 70 раз в секунду. Видеоконтроллер при этом мог бы считывать из видеопамяти первые 80 символов, генерировать для них 14 строк растра, затем считывать следующую строку символов, генерировать растр для них и т. д. Фактически же, большинство контроллеров считывают символ из видеопамяти один раз на строку растра, чтобы не было необходимости в буферизации. Рисунки символов 9 × 14 хранятся в ПЗУ видеоконтроллера (или ОЗУ, чтобы отображать пользовательские шрифты). Эта область памяти адресуется 12-битным адресом, в котором 8 бит определяют код символа, а четыре бита задают номер строки. Восемь битов каждого байта памяти определяют 8 пикселов, а 9-й пиксел всегда пустой. Таким образом, для отображения на экране одной строки текста требуется 14 × 8 = 1120 обращений к видеопамяти. То же количество обращений делается к ПЗУ генератора символов.

У IBM PC есть несколько режимов работы экрана. Простейший из них представляет собой символьный дисплей для консоли. На рис. 3.20, *а* мы видим участок видеопамяти. Каждый символ на рис. 3.20, *б* представлен в видеопамяти двумя байтами. Младший байт — это ASCII-код символа, а старший — байт атрибутов, который может задавать цвет, инверсию, мигание и прочие атрибуты символа. В этом режиме весь экран 25 × 80 символов занимает 4000 байт видеопамяти.

Графические терминалы основаны на том же принципе, за исключением того, что позволяют управлять цветом каждого пиксела на экране по отдельности. В простейшем варианте, для монохромного дисплея, каждому пикселу соответствует один бит видеопамяти. В другом предельном случае цвет пиксела описывается 24-разрядным словом, в котором по 8 бит выделяется на красный, зеленый

и синий компоненты. Чтобы просто хранить содержимое экрана разрешением 768 × 1024 пикселов и глубиной цветности 24 бит/пиксел, требуется 2 Мбайт видеопамяти.

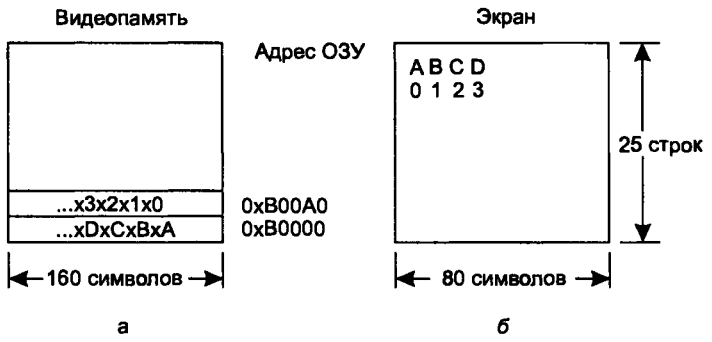


Рис. 3.20. а — содержимое видеопамяти для монохромного дисплея IBM; *б* — соответствующее изображение на экране (x — байты атрибутов)

У отображаемого в память терминала клавиатура полностью отделена от экрана. Она может подключаться через последовательный или параллельный порт. При каждом срабатывании клавиши генерируется прерывание, и драйвер клавиатуры считывает из порта ввода/вывода код нажатого символа.

У IBM PC клавиатура содержит встроенный микропроцессор, который через специальный последовательный порт взаимодействует с чипом контроллера на материнской плате. Прерывание генерируется каждый раз, когда клавиша нажимается или отпускается. Все, что сообщает клавиатура, — это номер нажатой клавиши, а не ее ASCII-код. Например, когда нажимается клавиша A, в регистр ввода/вывода помещается код 30. А означает ли это символ верхнего регистра, нижнего регистра, CTRL+A, ALT+A, CTRL+ALT+A или другую комбинацию, должен знать драйвер клавиатуры. У него имеется достаточно информации для выполнения этой работы, так как он в курсе, какие клавиши были нажаты, но не были отпущены (например, SHIFT). Так как клавиатурный интерфейс перекладывает все сложности на программное обеспечение, он очень гибок. Например, программе может потребоваться информация о том, была ли цифра набрана на цифровой клавиатуре или на основной. В принципе, драйвер клавиатуры может предоставить эту информацию.

Терминалы RS-232

Терминалы RS-232 представляют собой устройства, состоящие из клавиатуры и экрана, взаимодействующие через последовательный интерфейс. Такой интерфейс может передавать по одному биту за раз (рис. 3.21). Эти терминалы подключаются через 9- или 25-контактный разъем, в котором один контакт служит для передачи данных, один для приема и один — заземление. Остальные контакты отвечают за разного рода управляющие функции, и большинство из них не

используется. Чтобы передать символ, терминал RS-232 должен передавать его по одному биту, предварив передачу стартовым битом и завершив ее одним или двумя стоповыми битами. Также может быть добавлен бит контроля четности, который позволяет делать простейшую проверку правильности передачи. Этот бит может находиться перед стоп-битами, хотя обычно он требуется только для взаимодействия с мэйнфреймами. Стандартно применяются скорости передачи данных 9600, 19200 и 38400 бит/с. Терминалы RS-232 обычно подключаются к удаленной системе при помощи модема или телефонной линии.

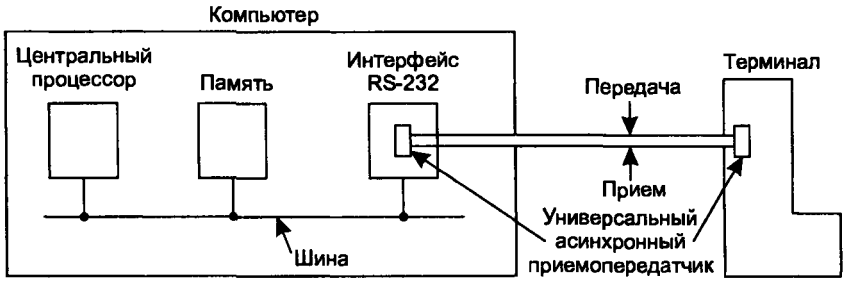


Рис. 3.21. Терминал RS-232 взаимодействует с компьютером через последовательную линию передачи данных, по одному биту за раз. Терминал и компьютер полностью независимы

Так как на внутреннем уровне как терминал, так и компьютер представляют информацию в виде символов, а передают по последовательной линии, были разработаны микросхемы, осуществляющие преобразование символа в последовательное представление и обратное преобразование. Эти чипы называются *UART* (Universal Asynchronous Receiver Transmitter, универсальный асинхронный приемопередатчик). UART подключаются к компьютеру при помощи интерфейсных карт RS-232, соединенных с шиной. Терминалы RS-232 постепенно вымирают, вытесняемые персональными компьютерами и X-терминалами, но они все еще встречаются в старых мэйнфреймах, особенно в банковских системах и им подобных.

Чтобы напечатать символ, драйвер терминала записывает его в интерфейсную карту. Там он буферизуется и при помощи UART последовательно сдвигается, а его биты передаются через последовательную линию. Даже на скорости 38400 бит/с для передачи одного символа требуется 250 мс. Как следствие малой скорости передачи, драйвер терминала обычно передает символ карте RS-232 и блокируется.

Чтобы вывести символ на экран, драйвер терминала записывает этот символ в интерфейсную карту, в которой она буферизируется, после чего поразрядно выдвигается в последовательную линию универсальным асинхронным приемопередатчиком. Например, для аналогового модема, работающего со скоростью 56000 бит/с, для передачи одного символа требуется немного более 179 мкс. Поскольку такая скорость передачи низка, драйвер обычно передает один символ в интерфейсную карту RS-232. Затем драйвер блокируется и ждет прерывания,

которое инициирует интерфейс, передав символ и перейдя в состояние готовности к приему следующего символа. Микросхема UART способна одновременно передавать и принимать символы. Прерывание также генерируется при получении символа, и обычно несколько принятых символов могут сохраняться в буфере. Получив прерывание, драйвер терминала должен проверить регистр, чтобы определить причину прерывания. Некоторые интерфейсные карты имеют собственный процессор и память и способны одновременно поддерживать несколько линий, разгружая тем самым центральный процессор.

Терминалы с интерфейсом RS-232 подразделяются на три категории. Наиболее простыми являются печатающие терминалы или телетайпы. Символы, набираемые на клавиатуре, посылаются компьютеру. Символы, посланные компьютером, печатаются на бумаге. Такие терминалы уже давно считаются устаревшими и почти не встречаются, разве только в качестве примитивных принтеров.

Простейшие электронно-лучевые терминалы работают похоже, но вместо бумаги они выводят символы на экран. Их также называют «стеклянными телетайпами» (glass ttys), поскольку функционально они аналогичны печатающим телетайпам. Термин «tty» является сокращением слова Teletype®, означающего имя компании, бывшей пионером в области компьютерных терминалов. Теперь сокращение «tty» используется для обозначения любого терминала. «Стеклянные» телетайпы также устарели.

Умные электронно-лучевые терминалы на самом деле представляют собой небольшие специализированные компьютеры. У них есть процессор и память. Они также содержат программное обеспечение, хранящееся, как правило, в ПЗУ. С точки зрения операционной системы, основное различие между «стеклянным» телетайпом и «умным» терминалом состоит в том, что последний понимает управляющие последовательности символов, называемые ESC-последовательностями. При помощи передачи такому терминалу ASCII-символа ESC (0x1B), за которым передается еще несколько других символов, можно управлять выводом на экран терминала. Например, с помощью ESC-последовательности можно переместить курсор на новую позицию, вывести текст в любое заданное место экрана, очистить экран и т. д. Именно такие терминалы используются в системах мэйнфреймов и эмулируются другими операционными системами. Ниже мы обсудим программное обеспечение «умных» терминалов.

X-терминалы

Наиболее интеллектуально развиты терминалы, содержащие центральный процессор, такой же мощный, как и у основного компьютера, с мегабайтами памяти, клавиатурой и мышью. Терминалом такого типа является *X-терминал*, на котором работает система *X Window System* (часто называемая просто X), разработанная в Массачусетсском технологическом институте. Обычно X-терминалы соединяются с компьютером через Ethernet.

X-терминал представляет собой компьютер, на котором работают X-программы и который взаимодействует с программами, работающими на удаленном компьютере. Некоторые специализированные устройства предназначены исключительно для работы X, на других, многоцелевых компьютерах, X-программа запускается

просто как одна из пользовательских программ. В любом случае, у X-терминала должен быть большой графический экран, с разрешением от 960 × 1200 и выше, монохромный, черно-белый или цветной, с расширенной клавиатурой и мышью, обычно с тремя кнопками.

Программа, работающая на X-терминале, собирающая ввод с клавиатуры или мыши и принимающая команды от удаленного компьютера, называется *X-сервером*. X-сервер общается по сети с *X-клиентами*, работающими на удаленном хосте. Он посылает им данные с клавиатуры и мыши, а также принимает от них команды отображения. Может показаться странным наличие X-сервера на терминале и клиентов на удаленном хосте, но работа X-сервера состоит в отображении битов, поэтому он должен находиться близко к пользователю. С точки зрения программы, клиент велит серверу выполнить те или иные действия, например вывести текст или отобразить геометрическую фигуру. Сервер (в терминале), как и все серверы, просто делает то, что ему велят. Схема взаимодействия клиента и сервера показана на рис. 3.22.

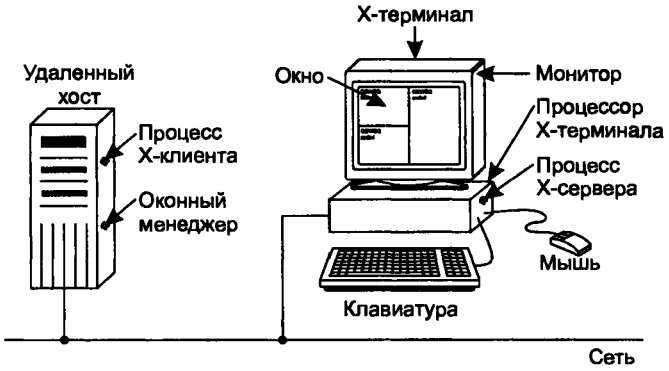


Рис. 3.22. Клиенты и серверы в системе X Windows

Поверх экрана X-терминала выводятся окна, каждое из которых представляет собой прямоугольник из пикселов. У каждого окна обычно есть строка заголовка в верхней части, полоса прокрутки слева и прямоугольник в нижней левой части для изменения размера окна. Одним из X-клиентов является программа, называемая *менеджером окон* (window manager). Ее назначение — контролировать процесс создания, уничтожения и перемещения окон на экране. Управляя окнами, она посылает X-серверу команды, говорящие ему, что делать. Среди этих команд есть, например, такие, как рисование точки, линии, прямоугольника, многоугольника, заполненных прямоугольника и многоугольника и др.

Назначение X-сервера в том, чтобы координировать данные от клавиатуры, мыши и клиентов и соответствующим образом обновлять изображение на экране. Она должна следить за тем, которое из окон выбрано в данный момент (то, над которым находится курсор мыши). Таким образом X-сервер узнает, которому клиенту направлять ввод с клавиатуры.

3.9.2. Программное обеспечение терминала

Клавиатура и экран являются почти независимыми устройствами, поэтому мы будем рассматривать их здесь по отдельности. Однако они не совсем независимы, так как вводимый с клавиатуры символ обычно эхом выводится на экран. В MINIX клавиатура и экран являются частью одной задачи, в других системах за них могут отвечать разные драйверы.

Программное обеспечение ввода

Основная работа клавиатурного драйвера состоит в сборе вводимых символов с клавиатуры и передаче их программам, читающим с терминала. Существует две концепции, описывающие работу драйвера. Согласно первой, задача драйвера заключается в сборе ввода и передаче его программам без всяких изменений. Программа, читающая с терминала, получает необработанные последовательности ASCII-символов. (Передавать программам пользователя коды клавиш неприемлемо, так как они в большой степени зависят от конкретной машины.)

Эта философия хорошо удовлетворяет потребности таких сложных текстовых редакторов, как `emacs`, который позволяет пользователю связать любое действие с любым символом или последовательностью символов. Однако это означает, что если пользователь вместо `date` наберет на клавиатуре `dste`, а затем исправит ошибку, удалив три последние символа и допечатав символы `ate`, за которыми нажмет `Enter`, программа пользователя получит одиннадцать ASCII-символов.

Не всем программам нужны эти подробности. Чаще всего им нужна уже исправленная строка, а не вся последовательность введенных символов. Таким образом, формируется вторая философская идея: драйвер выполняет все редактирование внутри строки, а программе пользователя передает уже только результат. Первая концепция является символьно-ориентированной, вторая — строчно-ориентированной. Изначально эти режимы работы драйвера назывались *режимом без обработки* (или «сырым» режимом) и *режимом с обработкой*. В стандарте POSIX режим с обработкой называется *каноническим режимом*. *Неканонический режим* соответствует режиму без обработки, хотя многие детали поведения терминала могут различаться. Совместимые со стандартом POSIX системы предоставляют несколько библиотечных функций, поддерживающих выбор любого из этих двух режимов, а также изменение многих аспектов конфигурации терминала.

Итак, основная задача клавиатурного драйвера состоит в сборе символов. Если каждое нажатие на клавишу вызывает прерывание, драйвер может получать введенный символ во время обработки прерывания. Если прерывания преобразуются низкоуровневым программным обеспечением в сообщения, каждый полученный символ может включаться в сообщение. В качестве альтернативы символ может помещаться в небольшой буфер в памяти, а сообщение использоваться только для извещения драйвера о том, что что-то прибыло. Второй подход более надежен, особенно если сообщение посылается только ожидающему его процессу, а драйвер клавиатуры занят обработкой предыдущего символа.

Получив символ, драйвер должен начать его обработку. Если клавиатура передает информацию о номере нажатой клавиши, а не о коде символа, который нужен для прикладных программ, драйверу требуется преобразовать номер в код символа при помощи таблицы. Не все IBM-совместимые клавиатуры имеют одинаковую нумерацию клавиш, поэтому драйвер, чтобы обеспечить поддержку различных клавиатур, должен иметь разные таблицы перекодировки для разных клавиатур. Простейший подход — скомпилировать драйвер с таблицей, предназначенной для преобразования кодов клавиш в кодировку ASCII (American Standard Code for Information Interchange, американский стандартный код обмена информацией). К сожалению, такое решение неудовлетворительно для неанглоязычных пользователей. В разных странах приняты разные раскладки клавиатур, и стандартного набора символов ASCII недостаточно для большинства людей, населяющих восточное полушарие. Например, тем, кто разговаривает на французском, португальском и испанском языках, требуются буквы с надстрочными знаками и знаки препинания, которых нет в английском. Чтобы обеспечить гибкую работу с клавиатурными раскладками для различных языков, во многих операционных системах имеется возможность загружать различные *кодовые страницы*. Они позволяют выбирать (при загрузке или позже), как клавиатурные коды будут преобразовываться в коды символов.

Если терминал находится в каноническом режиме (режиме с обработкой), введенные символы должны храниться в буфере до тех пор, пока не будет завершена вся строка. Даже если терминал переведен в «сырой» режим, может оказаться, что программа еще не запрашивала входные данные, поэтому введенные символы все равно должны буферизироваться, чтобы позволить пользователю производить упреждающий ввод. (Разработчиков систем, не позволяющих пользователям вводить символы с клавиатуры без возможности исправления, следует обмазывать дегтем и вываливать в перьях, ибо заставлять их пользоваться собственной системой было бы слишком жестоким наказанием.)

Для буферизации символов обычно применяются два метода. В первом случае в драйвере содержится центральный пул буферов, в каждом из которых хранится около 10 символов. С каждым терминалом связана структура данных, содержащая, среди прочего, указатель на цепочку буферов, в которых находятся символы, введенные с данного терминала. Чем больше символов введено, тем больше выделяется буферов, соединенных в цепь. Когда символ передается программе пользователя, буферы удаляются и память возвращается центральному пулу.

Другой подход заключается в том, что буферизация производится прямо в структуре данных терминала, без центрального пула буферов. Поскольку пользователи часто печатают команду, обработка которой требует некоторого времени (например, на перекомпиляцию и сборку большой программы), а затем печатают еще несколько строк, буфер драйвера должен вмещать не меньше 200 символов для каждого терминала. В крупной системе разделения времени с сотней терминалов постоянное выделение 20 Кбайт на буфер ввода с клавиатур кажется чрезмерным, поэтому центральный пул буферов размера около 5 Кбайт будет, видимо, достаточным. С другой стороны, при выделенном буфере для от-

дельного терминала драйвер становится проще (не требуется управления списком). Такой подход является предпочтительным на персональном компьютере с единственной клавиатурой. Рисунок 3.23 иллюстрирует разницу между этими двумя методами.

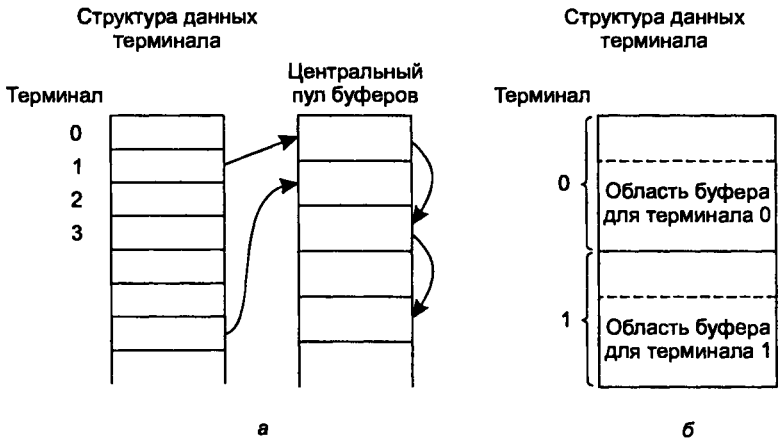


Рис. 3.23. *а* — центральный пул буферов; *б* — выделенный буфер для каждого терминала

Хотя клавиатура и экран являются логически разделенными устройствами, многие пользователи привыкли видеть только что введенные с клавиатуры символы отображаемыми на экране. Некоторые (старые) терминалы должны были автоматически (аппаратно) отображать все, что вводилось с клавиатуры, что не только крайне неудобно при вводе паролей, но также значительно ограничивает гибкость сложных редакторов и других программ. К счастью, на большинстве терминалов при нажатии клавиши ничего автоматически не отображается. Отображением символов на экране занимается исключительно программное обеспечение. Этот процесс называется *печатью эха* или *эхо-отображением.*

Эхо-отображение несет в себе ту сложность, что во время нажатия пользователем клавиши программа может осуществлять правильный вывод на экран. По меньшей мере, драйвер должен решить, где поместить «эхо» так, чтобы оно не кануло под выводным потоком программы.

Кроме того, если пользователь ввел более 80 символов в одной строке, вывод «эха» на 80-символьном экране может выглядеть по-разному. В зависимости от приложения переход на следующую строку может оказаться приемлемым либо неприемлемым. Некоторые драйверы просто усекают все введенные строки до 80 символов, игнорируя все после 80-й колонки.

Еще одна проблема заключается в обработке символов табуляции. Обычно драйвер вычисляет текущую позицию курсора, учитывая как вывод программы, так и «эхо», после чего вычисляет число отображаемых до позиции табулятора пробелов.

Наконец, существует проблема эквивалентности устройств. Логически в конце строки текста требуется символ возврата каретки, чтобы переместить курсор

обратно к колонке 1, и символ перевода строки для перемещения курсора на следующую строку. Требовать от пользователя вводить оба символа — вряд ли удачная мысль, хотя на некоторых терминалах имеется специальная клавиша, посылающая оба символа с 50%-й вероятностью в том порядке, в котором их ожидает программа. Преобразование всего, что поступает с клавиатуры в стандартный внутренний формат, используемый операционной системой, является одной из задач драйвера.

Если стандартом регламентируется хранение только символов перевода строки (соглашение UNIX), тогда символы возврата каретки должны преобразовываться в символы перевода строки. Если внутренний формат предусматривает хранение обоих символов (соглашение Windows), тогда драйвер должен формировать символ перевода строки при получении символа возврата каретки, и наоборот. Независимо от внутренних правил, терминал может требовать наличия обоих символов для корректного управления выводом на экран. Поскольку к большому компьютеру могут оказаться подключенными терминалы различных типов, драйвер клавиатуры должен заниматься преобразованием всех различных комбинаций символа возврата каретки и символа перевода строки во внутренний стандарт, а также следить за правильностью эхопечати.

И это еще не все. Другая проблема, связанная с перемещением на новую строку. Некоторым терминалам, чтобы отобразить такие символы, требуется больше времени, чем на отображение нормальных символов, букв или цифр. К примеру, если микропроцессору в терминале, чтобы выполнить прокрутку экрана, нужно скопировать большой блок текста, то перевод строки может выполняться медленно. Если механической печатающей головке надо вернуться к левому полю документа, на вывод символа возврата каретки потребуется больше времени. В обоих случаях драйвер терминала может вставлять в выходной поток *символы заполнения* (пустые символы) или же просто приостанавливать вывод на некоторое время, чтобы терминал успел за потоком символов. Необходимая задержка часто связана со скоростью терминала. Например, на скорости 4800 бит/с и ниже задержка вряд ли нужна, а на 9600 бит/с и выше может понадобиться один заполняющий символ. Терминалам с аппаратной поддержкой табуляции, особенно тем, которые выдают печатный документ, дополнительная задержка может потребоваться после символа табуляции.

При работе в каноническом режиме некоторые вводимые символы имеют особое значение. В табл. 3.9 показаны все специальные символы, определенные стандартом POSIX. По умолчанию все они являются управляющими символами, которые не должны конфликтовать с вводимым текстом или кодами, используемыми программами. Однако все символы, кроме последних двух, допустимо программно изменять.

Символ ERASE позволяет пользователю удалить один только что введенный символ, что обычно делается клавишей «забой» (backspace) или комбинацией клавиш CTRL+H (обоим вариантам соответствует код 0x08). Этот символ не добавляется к очереди символов, а, наоборот, удаляет предыдущий символ из очереди. Печать эха для такого символа должна выглядеть как последовательность трех символов: перемещение курсора на позицию влево, пробел и еще раз

возврат на позицию, чтобы удалить с экрана предыдущий символ. Если же предыдущим символом был символ табуляции, его удаление зависит от его же интерпретации при печати. Если он был преобразован в пробелы, необходима дополнительная информация о том, насколько далеко следует смещать курсор. Если же сам символ табуляции хранится в очереди ввода, он может быть удален, а вся строка напечатана еще раз. В большинстве систем символ ERASE удаляет символы текущей строки. Символы из предыдущей строки и разделяющие строки символы возврата каретки или перевода строки не удаляются.

Таблица 3.9. Специальные символы канонического режима

Символ	Имя в POSIX	Комментарий
CTRL+H	ERASE	Удалить один символ слева
CTRL+U	KILL	Удалить всю введенную строку
CTRL+V	LNEXT	Интерпретировать следующий символ буквально
CTRL+S	STOP	Остановить вывод
CTRL+Q	START	Начать вывод
DEL	INTR	Прервать процесс (SIGINT)
CTRL+\	QUIT	Принудительный дамп памяти (SIGQUIT)
CTRL+D	EOF	Конец файла
CTRL+M	CR	Возврат каретки (неизменный символ)
CTRL+J	NL	Перевод строки (неизменный символ)

Если пользователь обнаруживал ошибку в начале введенной строки, единственным способом ее исправления во многих старых системах, не позволяющих перемещать курсор влево-вправо, являлось удаление всей строки. В этом случае бывало удобнее воспользоваться специальным символом KILL. В некоторых системах эта строка полностью исчезала с экрана, но в других она оставалась видимой, включая возврат каретки и перевод строки, в соответствии с предпочтениями иных пользователей. Как и ERASE, символ KILL работает только с текущей строкой. При удалении блока символов драйвер вправе вернуть освободившиеся буферы в пул.

Иногда символы ERASE или KILL должны быть введены в строку как обычные данные. Для этого служит символ LNEXT, действующий в качестве *префиксного символа*. В системе UNIX ему по умолчанию соответствует сочетание клавиш CTRL+V (код 0x16). В более старых версиях UNIX в качестве символа KILL часто используется символ @, но впоследствии этот символ стал составной частью адресов электронной почты сети Интернет, как, например, linda@cs.washington.edu. Те, кому привычнее старые соглашения, могут переопределить символ KILL как @, но тогда им придется вводить символ @ буквально при наборе адреса электронной почты. Это можно сделать, нажав на клавиатуре последовательно клавиши CTRL+V и @. Сам символ LNEXT может быть введен, если дважды нажать клавиши CTRL+V. Встретив символ LNEXT, драйвер установит флаг, означающий, что следующий символ не следует подвергать специальной обработке. Сам символ LNEXT не помещается в очередь символов.

Чтобы приостановить и продолжить вывод на экран, также предоставляются специальные управляющие коды. В UNIX это символы STOP (CTRL+S) и START (CTRL+Q). Они не хранятся в буфере, но используются для установки и сброса флага в структуре данных терминала. При каждой операции вывода на экран проверяется значение этого флага. Если флаг установлен, вывод не производится. Эхо при этом, как правило, также подавляется.

Часто возникает необходимость прервать выполнение отлаживаемой программы. Для этой цели могут использоваться символы INTR (DEL) и QUIT (CTRL+\). В системе UNIX клавиша DEL посылает сигнал прерывания SIGINT всем процессам, запущенным с этого терминала. Реализация может быть непростой. Наиболее сложным делом является передача информации от драйвера в ту часть системы, которая занимается обработкой сигналов, поскольку она не ожидает получения подобной информации. Результат нажатия клавиш CTRL+\ (код 0x1C) аналогичен нажатию клавиши DEL, с той разницей, что процессам посылается сигнал SIGQUIT, вызывающий прекращение работы процесса с сохранением дампа памяти, если этот сигнал специально не перехватывается процессом. При нажатии любой из означенных клавиш драйвер должен вывести эхо в виде перевода строки и возврата каретки, а также очистить свой буфер с накопленными символами, чтобы позволить начать новый ввод. Часто вместо клавиши DEL для символа INTR по умолчанию используется сочетание клавиш CTRL+C (код 0x03), так как с появлением электронно-лучевых дисплеев стало привычным нажимать клавишу DEL для удаления символа справа от курсора при редактировании.

Специальный символ EOF (CTRL+D), означающий конец файла в UNIX, сообщает ожидающей ввода программе, что информации на входе больше не будет. Программа действует так, как если бы при чтении из файла уперлась в его конец.

Некоторые драйверы терминала предоставляют возможность более сложного редактирования строки, чем было описано здесь. Они имеют специальные управляющие символы, позволяющие удалять целиком слова, перемещать курсор вперед и назад по символам и по словам, вставлять текст в середину уже набранной строки и т. д. Добавление подобных функций к драйверу значительно увеличивает его. Кроме того, эти функции чаще всего оказываются неиспользуемыми экранными редакторами, предпочитающими работать с драйверами клавиатуры в «сыром» режиме.

Стандарт POSIX требует того, чтобы в стандартной библиотеке были доступны несколько функций, позволяющих управлять параметрами терминала. Самые важные из них tcgetattr и tcsetattr. Первая получает от системы копию структуры termios, показанной в листинге 3.3. Эта структура содержит всю информацию, необходимую для задания режимов работы, настройки специальных символов и управления характеристиками терминала. Программа может узнать текущие значения установок и изменить их на свой вкус. Функция tcsetattr позволяет передать эту структуру обратно драйверу терминала.

Стандарт POSIX не раскрывает, как реализовывать свои требования при помощи системных вызовов или библиотечных подпрограмм. В MINIX имеется системный вызов ioctl, записываемый следующим образом:

```
ioctl(file_descriptor, request, argp);
```

Листинг 3.3. Структура termios. В MINIX тип c_flag_t эквивалентен short, speed_t — int и cc_t равен char

```
struct termios {
   tcflag_t c_iflag;                    /* Режимы ввода */
   tcflag_t c_oflag;                    /* Режимы вывода */
   tcflag_t c_cflag;                    /* Режимы управления */
   tcflag_t c_iflag;                    /* Локальные режимы */
   speed_t c_ispeed;                    /* Скорость ввода */
   speed_t c_ospeed;                    /* Скорость вывода */
   cc_t c_cc[NCCS];                     /* Управляющие символы */
};
```

Этот вызов позволяет определять и изменять конфигурацию многих устройств ввода/вывода. Функции tcgetattr и tcgetattr реализованы на его основе. Здесь переменная request указывает, считывать или записывать структуру termios (для записи также указывается, нужно ли выполнять действие немедленно или отложить до завершения текущей очереди запросов). Переменная argp содержит указатель на структуру termios. Такой подход был выбран по большей мере для совместимости с UNIX, чем из соображений элегантности.

Следует сделать несколько замечаний о структуре termios. Четыре управляющих слова в ней обеспечивают большую гибкость. Отдельные биты поля c_iflag отвечают за то, как обрабатываются входные данные. Например, бит ICRNL управляет преобразованием символов CR в NL при вводе. В MINIX этот флаг установлен по умолчанию. Поле c_oflags содержит флаги, управляющие выводом. Например, флаг OPOST разрешает обработку выводимых данных. Этот бит, а также бит ONLCR, который руководит преобразованием символов NL в последовательность CR NL, в MINIX устанавливаются по умолчанию. Поле c_cflag содержит флаги управления. Установки MINIX по умолчанию разрешают передачу 8-битных символов, кроме того, модем должен «класть трубку» при отключении пользователя. Поле c_lflag — это флаги *локального режима*. Бит ECHO управляет выводом эха (при входе пользователя в систему эту функцию можно отключить, чтобы обезопасить набор пароля). Один из самых важных битов — бит ICANON, разрешающий канонический режим. Когда ICANON сброшен, существует несколько возможностей. Если сохранить значения по умолчанию для всех остальных настроек, терминал переходит в режим, идентичный традиционному *режиму cbreak*. В этом режиме символы передаются программе, не дожидаясь ввода всей строки, но управляющие коды INTR, QUIT, START и STOP сохраняют свой эффект. Это можно отменить, сбросив значения других флагов, и получить эквивалент обычного режима без обработки.

Различные специальные символы, значения которых можно изменить (включая расширения MINIX), хранятся в массиве c_cc. Кроме того, в нем хранятся два параметра, используемые в неканоническом режиме. Значение MIN, помещаемое в c_cc[VMIN], задает минимальное количество символов, достаточное для вызова read. Величина TIME, хранящаяся в c_cc[VTIME], задает лимит времени для этого вызова. К какому результату приводят разные комбинации значений, показано в табл. 3.10, иллюстрирующей обработку вызова, запрашивающего N байтов. Если TIME = 0 и MIN = 1, поведение аналогично режиму без обработки.

Таблица 3.10. Параметры MIN и TIME определяют, как выполняется чтение в неканоническом режиме. N — число запрошенных байтов

TIME = 0		TIME > 0
MIN = 0	Вызов завершается немедленно, возвращая имеющееся число байтов, от 0 до N	Сразу же запускается таймер. Возвращается первый полученный байт, или ни одного, если истекло время
MIN > 0	Вызов возвращает от MIN до N байтов. Возможна бесконечная блокировка	После первого байта запускается межбайтовый таймер. Возвращается N байтов, если они уложились во временной интервал, но не меньше 1 байта. Возможна бесконечная блокировка

Программное обеспечение вывода

Терминальный вывод несколько проще ввода. По большей части компьютер посылает символы терминалу, который их отображает. Обычно блок символов, например строка, записывается на терминал за один системный вызов. Как правило, метод, используемый для терминалов с интерфейсом RS-232, состоит в том, что для каждого терминала выделяется выходной буфер. Эти буферы могут входить в тот же пул буферов, что и входные буферы, или представлять собой выделенные буферы. Вывод эха на терминал также копируется в буфер. После того как символы помещены в выходной буфер, первый символ выводится на терминал, а затем драйвер блокируется. Когда происходит прерывание, извещающее драйвер о готовности терминала принять следующий символ, посылается следующий символ и т. д.

Отображаемые на память терминалы позволяют применять еще более простую схему. Символы, которые должны быть напечатаны, один за одним извлекаются из пространства пользователя и записывают непосредственно в видеопамять. В случае с терминалами RS-232 символ просто передается в последовательную линию передачи данных. При отображении в память некоторые символы нуждаются в дополнительной обработке. Среди них символы забоя, возврата каретки, перевода строки и звукового сигнала (CTRL+G). Драйвер отображаемого в память терминала должен программно отслеживать текущее положение курсора, обновляя его после вывода печатного символа. При выводе специальных символов он должен соответствующим образом менять положение курсора.

В частности, переводя строку внизу экрана, необходимо выполнять прокрутку его содержимого. Как работает прокрутка, иллюстрирует рис. 3.20. Если видеоконтроллер всегда располагает начало видеопамяти по адресу 0xB0000, единственный способ сдвинуть содержимое — скопировать 24 × 80 символов (каждый символ представляется двумя байтами) из области, начинающейся по адресу 0xB00A0, в область 0xB0000. На это требуется время.

К счастью, обычно аппаратное обеспечение несколько упрощает эту операцию. У большинства видеоконтроллеров имеется регистр, задающий, какому адресу в видеопамяти соответствует первая строка экрана. Благодаря этому, чтобы прокрутить экран вверх на одну строку, можно изменить значение этого регистра так, чтобы видимая область начиналась по адресу 0xB00A0, а не 0xB0000.

Тогда единственное, что должен делать драйвер, — просто записать новые символы в следующую строку в видеопамяти. Когда видеоконтроллер достигает верхней границы видеопамяти, он переходит в ее конец, начиная вновь с нижнего адреса.

Еще одна вещь, о которой должен заботиться драйвер отображаемого в память терминала, — это позиционирование курсора. Опять же, упрощает дело наличие аппаратного регистра, задающего, куда должен переместиться курсор. Наконец, остается проблема обработки звукового сигнала, который получается путем подачи прямоугольной или синусоидальной волны на внутренний динамик, имеющий мало общего с видеопамятью.

Очевидно, что большинство проблем, с которыми сталкиваются драйверы отображаемых в память терминалов, переносятся и на микропроцессоры внутри терминалов RS-232. С точки зрения самого микропроцессора он является центральным, в системе с отображаемым в память терминалом.

Экранным редакторам и другим сложным программам бывает нужно перерисовать экран, заменив определенные его участки, не затрагивая остального текста. Для этого многие терминалы поддерживают наборы управляющих команд, позволяющие перемещать курсор, удалять строки и т. д. Эти команды часто реализуются в виде *ESC-последовательностей*, то есть последовательностей символов, начинающихся с символа ESC (0x1B). Во времена расцвета терминалов с интерфейсом RS-232 существовали сотни разновидностей терминалов, у каждого из которых был свой набор ESC-последовательностей. В результате было довольно сложно написать программное обеспечение, работающее более чем на одном типе устройств.

В конце концов производители компьютеров и программного обеспечения осознали необходимость стандартизации ESC-последовательностей, в результате чего был разработан стандарт ANSI. Некоторые примеры ESC-последовательностей этого стандарта приведены в табл. 3.11.

Таблица 3.11. Некоторые ESC-последовательности стандарта ANSI

ESC-последовательность	Значение
ESC [nA	Переместить курсор вверх на n строк
ESC [nB	Переместить курсор вниз на n строк
ESC [nC	Переместить курсор вправо на n позиций
ESC [nD	Переместить курсор влево на n позиций
ESC [m;nH	Переместить курсор в позицию (m, n)
ESC [sJ	Очистить экран от позиции курсора (0 — до конца, 1 — до начала, 2 — весь)
ESC [sK	Очистить строку от позиции курсора (0 — до конца, 1 — до начала, 2 — всю)
ESC [nL	Вставить n строк у курсора
ESC [nM	Удалить n строк у курсора
ESC [nP	Удалить n символов у курсора
ESC [n@	Вставить n символов у курсора

ESC-последовательность	Значение
ESC [nm	Разрешить выделение текста (0 — нормальный, 4 — полужирный, 5 — мерцающий, 7 — инверсный)
ESC M	Прокрутка экрана в обратную сторону, если курсор находится в верхней строке

3.9.3. Обзор драйвера терминала в MINIX

Код драйвера терминала в MINIX находится в четырех С-файлах (если включена поддержка псевдотерминалов RS-232, то в шести). Вместе они образуют самый большой и сложный драйвер в MINIX. Сложность частично объясняется тем, что драйвер должен обслуживать одновременно клавиатуру и экран, которые сами по себе являются сложными устройствами, а также два других опциональных типа терминалов. Но многих все равно удивляет, что обслуживание терминального ввода/вывода требует в тридцать раз больше кода, чем занимает планировщик. (Это удивление подкрепляется многочисленными книгами, в которых обсуждению работы планировщика отводится в тридцать раз больше места, чем обсуждению ввода/вывода.)

Задача терминала принимает следующие семь типов сообщений.

1. Чтение с терминала (от файловой системы по поручению пользовательского процесса).
2. Вывод на терминал (от файловой системы по поручению пользовательского процесса).
3. Установка параметров терминала для ioctl (от файловой системы по поручению пользовательского процесса).
4. Ввод/вывод, произошедший за последний тик таймера (от прерывания часов).
5. Отмена предыдущего запроса (от файловой системы, когда возникает сигнал).
6. Открыть устройство.
7. Закрыть устройство.

Сообщения с командами на запись и чтение имеют тот же формат, что и сообщения, показанные в табл. 3.4, за тем небольшим исключением, что не требуется поле POSITION. Работая с диском, программа должна указывать, какой блок необходимо считать. При работе с терминалом такого выбора нет: всегда считывается следующий введенный пользователем символ. Терминалы не поддерживают переход на произвольную запись (позиционирование).

Функции стандарта POSIX, tcsetattr и tcgetattr, необходимые для определения и изменения параметров (атрибутов) терминала, поддерживаются при помощи системного вызова ioctl. Хороший стиль программирования требует, чтобы для управления терминалом использовались эти и другие функции из файла include/termios.h, а преобразование этих вызовов в системный вызов ioctl было бы оставлено на стандартную библиотеку языка С. Тем не менее существуют не-

которые операции, которые не охватываются стандартом POSIX, например загрузка альтернативной раскладки клавиатуры. Для таких операций не остается ничего другого, как явно использовать вызов ioctl.

Сообщение, которое ioctl отправляет драйверу терминала, содержит код запрашиваемой функции и указатель. Библиотечной функции tcsetattr соответствуют коды функций TCSETS, TCSETSW и TCSETSF, а указатель должен ссылаться на структуру termios (см. листинг 3.3). В данном случае вызов замещает текущие значения атрибутов новыми, различие между тремя функциями в том, что запрос TCSETS приводит к немедленному изменению атрибутов, TCSETW выполняется только после того, как завершается вывод, а TCSETSF сначала дожидается окончания вывода, а затем очищает все еще не считанные входные данные. Функция tcgetattr транслируется в вызов ioctl с кодом операции TCGETS и возвращает сделавшей вызов программе заполненную атрибутами структуру termios, позволяя определить параметры устройства. Те вызовы ioctl, у которых нет аналогов в стандарте POSIX, могут требовать ссылки на другие структуры. Например, для операции KIOCSMAP, загружающей новую раскладку клавиатуры, требуется ссылка на 1536- байтовую структуру keymap_t (16-битные коды для 128 клавиш × 6 модификаторов). Таблица 3.15 резюмирует, как вызовы стандарта POSIX преобразуются в системные вызовы ioctl.

У драйвера терминала имеется одна центральная структура данных, tty_table, представляющая собой массив структур tty, по одной на терминал. У стандартного персонального компьютера есть только одна клавиатура и экран, но MINIX поддерживает до восьми виртуальных терминалов, в зависимости от объема памяти у контроллера. Это позволяет с одной консоли входить в систему несколько раз, переключая клавиатуру между разными «пользователями». Если есть две виртуальные консоли, то, нажав ALT+F2, можно переключиться на вторую, а нажатие ALT+F1 активизирует первую консоль. Для переключения можно также использовать комбинацию ALT и клавиши управления курсором. Кроме того, последовательные линии обеспечивают поддержку двух удаленных пользователей, подключающихся через модем или кабель RS-232, а *псевдотерминалы* позволяют пользователям подключаться через сеть. Драйвер написан так, чтобы можно было легко добавлять новые терминалы.

Каждая из структур tty в массиве tty_table отслеживает как ввод, так и вывод. При вводе она поддерживает очередь символов, которые были введены пользователем и еще не были считаны, информацию о запросе на чтение символов и об интервалах таймера, чтобы задача не блокировалась, если не нажато ни одного символа. При выводе она хранит параметры запросов на вывод, которые еще не были выполнены. Также есть другие поля с различной общей информацией, например с описанной выше структурой termios, которая оказывает влияние на многие особенности ввода и вывода. Кроме того, в структуре tty есть указатель, который ссылается на информацию, необходимую для конкретного класса устройств, но не требуемую для всех терминалов. Так, завязанной на аппаратное обеспечение части кода драйвера терминала требуется информация о текущем положении вывода в видеопамяти, но при работе с линией RS-232 эта информация не нужна. Дополнительно для каждого типа устройств имеются собственные

структуры данных, содержащие информацию о расположении буферов, в которые помещают свои данные обработчики прерываний. Медленным устройствам, таким как клавиатуры, не нужны такие большие буферы, как быстрым.

Терминальный ввод

Чтобы лучше понять, как работает драйвер терминала, сначала рассмотрим, как напечатанный пользователем символ прокладывает себе путь к нуждающейся в нем программе.

Когда пользователь входит с системной консоли, для него создается оболочка, которая для ввода, вывода и вывода ошибок использует /dev/console. Запустившись, оболочка пытается считать данные из стандартного ввода при помощи библиотечного вызова read. Эта процедура отправляет файловой системе сообщение, содержащее дескриптор файла, адрес буфера и количество считываемых байтов. На рис. 3.24 это сообщение обозначено как 1. Отправив сообщение, оболочка блокируется, ожидая ответа. (Пользовательские процессы исполняют только примитив send_rec, который комбинирует вызов send с вызовом receive от того процесса, которому было послано сообщение.)

Файловая система получает сообщение и определяет местонахождение *i*-узла, соответствующего указанному дескриптору файла. В данном случае это *i*-узел для специального символьного файла /dev/console, содержащий младший и старший номера устройства для терминала. Старший номер для терминалов равен 4, младший для консоли равен 0.

Файловая система при помощи карты устройств, dmap, определяет номер задачи терминала. Затем она отправляет выбранной задаче сообщение, которое на рис. 3.24 показано как 2. Обычно к этому моменту пользователь еще ничего не напечатал, и драйвер терминала не в состоянии выполнить запрос. Поэтому драйвер немедленно отправляет файловой системе ответ, чтобы разблокировать ее и сообщить, что ни одного символа пока не введено. Это показано на рисунке как 3. Файловая система фиксирует в таблице tty_table тот факт, что пользовательская задача ожидает ввода с терминала и переходит к следующему запросу. Пользовательская оболочка при этом остается заблокированной, пока не прибудут введенные символы.

Когда, наконец, на клавиатуре набран символ, возбуждаются два прерывания, одно — когда клавиша нажимается, и одно — когда отпускается. Это правило относится и к клавишам-модификаторам, таким как SHIFT или CTRL, которые сами по себе не передают символов, но все равно приводят к двум прерываниям. Прерыванию клавиатуры соответствует IRQ1, и ассемблерная функция _hwint01 в файле mpx386.s вызывает функцию kbd_hw_int, которая, в свою очередь, вызывает scan_keyboards, чтобы получить от аппаратного обеспечения клавиатуры коды клавиш. Если код соответствует обычному символу, он помещается во входную очередь клавиатуры, ibuf, тогда, когда прерывание было сгенерировано нажатием клавиши. Если прерывание инициировано отпусканием клавиши, оно в данном случае игнорируется. Коды клавиш-модификаторов, таких как CTRL или SHIFT, помещаются в очередь в любом случае, и при нажатии, и при отпускании. Обратите внимание: в очереди хранятся не ASCII-коды, а просто коды опросы кла-

виатуры, выдаваемые клавиатурами IBM. Затем kbd_hw_int устанавливает флаг tty_events, являющийся частью клавиатурной секции tty_table, вызывает force_timeouts и завершается.

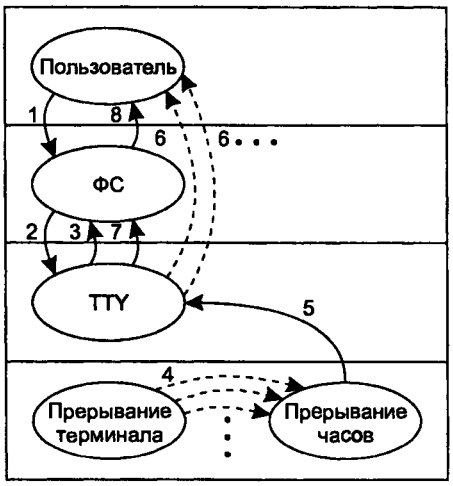


Рис. 3.24. Запрос на чтение с терминала в отсутствие введенных символов. ФС — файловая система. TTY — задача терминала. Обработчик прерываний помещает в очередь введенные символы сразу же, но TTY пробуждается только обработчиком прерываний часов

В отличие от других обработчиков прерываний, kbd_hw_int не отправляет сообщений задаче терминала. Вызов force_timeouts обозначен на рис. 3.24 пунктирными линиями (4). Это не сообщения. Здесь просто устанавливается значение переменной tty_timeout, регистрирующее, что настало время вызвать функцию tty_wakeup, которая и отправляет сообщение 5 задаче терминала. Обратите внимание, что, хотя код tty_wakeup находится в файле tty.c, эта функция вызывается в ответ на прерывание часов. Таким образом, сообщение задаче терминала отправляет обработчик прерываний часов. Если данные вводятся быстро, в очереди может оказаться несколько символов, поэтому на рисунке изображены многократные вызовы force_timeout.

Получив сообщение и пробудившись, задача терминала для каждого устройства проверяет значение флага tty_events и для каждого устройства, у которого флаг установлен, вызывает handle_events. Этот флаг может обозначать различные типы действий (хотя чаще всего — ввод), поэтому handle_events как для ввода, так и для вывода всегда вызывает специфичные для данного устройства функции. При вводе с клавиатуры вызывается kb_read, которая отслеживает наличие кодов управляющих клавиш CTRL, SHIFT и ALT и преобразует клавиатурные коды в ASCII, учитывая специальные символы и значения различных флагов, включая флаги, обозначающие канонический режим. В результате символ по большинству просто добавляется во входную очередь клавиатуры в tty_table, хотя некоторые коды, например BACKSPACE, могут приводить к другому эффекту.

Кроме того, обычно in_proccess осуществляет эхо-отображение вводимых ASCII-символов.

Когда введено достаточное количество символов, задача терминала вызывает ассемблерную подпрограмму phys_copy, которая копирует введенные данные по адресу, указанному оболочкой. Эта операция также не является передачей сообщения, поэтому обозначена на рис. 3.24 пунктирной линией. На рисунке изображено несколько линий потому, что до того как пользовательский запрос будет полностью выполнен, данные могут быть переданы несколько раз. Когда операция полностью завершится, драйвер терминала отправляет файловой системе сообщение о том, что работа выполнена (7), а файловая система в результате отправляет сообщение оболочке и выводит ее из состояния блокировки (8).

Определение того, какого количества символов достаточно, зависит от режима терминала. В каноническом режиме запрос считается выполненным, когда поступает символ перевода строки, конца строки или конца файла. Также длина строки не может превышать величину входной очереди, чтобы входные данные могли правильно обрабатываться. В неканоническом режиме может запрашиваться гораздо большее количество символов, и функции in_process может потребоваться передавать данные несколько раз перед тем, как файловой системе будет возвращено сообщение, индицирующее завершение работы.

Обратите внимание на то, что драйвер терминала копирует данные напрямую из своего адресного пространства в адресное пространство оболочки. Данные не передаются через файловую систему. При блочных операциях ввода/вывода информация сначала поступает к файловой системе, чтобы та могла поддерживать кэш недавно запрошенных блоков. Если запрошенный блок оказывается в кэше, файловая система может самостоятельно выполнить пользовательский запрос, без реального обращения к диску.

При работе с терминалом кэширование не имеет смысла. Кроме того, запрос файловой системы к драйверу диска всегда может быть обслужен максимум за несколько сотен миллисекунд, поэтому нет большой проблемы в том, что файловая система немного подождет. Операции с терминалом могут длиться часами или вообще могут не завершиться (в каноническом режиме драйвер ждет ввода полной строки, а в неканоническом ждет достаточно длинной строки, в зависимости от параметров MIN и TIME). Поэтому файловая система не должна блокироваться, ожидая выполнения запросов терминального ввода/вывода.

Вполне ожидаемо, что пользователь ввел какой-либо текст заранее, и символы благодаря предыдущим событиям 4 и 5 оказались доступны до того, как они были запрошены. В этом случае все события 1, 2, 6, 7 и 8 происходят сразу же после запроса, а 3 вообще не происходит.

Если случилось так, что задача терминала работает в момент возникновения прерывания часов, сообщение нельзя отправить, так как его никто не ждет. Поэтому флаг tty_events проверяется несколько раз в других местах кода, чтобы ввод и вывод происходили равномерно при занятости задачи терминала. Например, проверка делается непосредственно после обработки и ответа на сообщение. Таким образом, вводимые символы могут попадать во входную очередь без помощи сообщений от задачи часов. Если до того как терминал закончит выполне-

ние текущей задачи, произойдет два или больше прерывания часов, все символы сохраняются в ibuf, и повторно устанавливается флаг tty_events. В конечном счете задача терминала получит одно сообщение, все остальные будут потеряны. Но так как все переданные символы были сохранены в буфере, ни один из напечатанных символов не затеряется. Возможно даже, что к тому времени, когда задачей терминала будет получено сообщение, ввод уже был завершен, и пользовательскому процессу был отправлен ответ.

Системе обмена сообщений без буферизации (принцип рандеву) свойственна та проблема, когда обработчик прерываний отправляет сообщение процессу, который в текущий момент занят. Для большинства устройств, таких как жесткие диски, прерывания генерируются только в ответ на запросы, сделанные драйвером, поэтому в каждый момент времени может потребоваться обработка не более одного прерывания. Единственные устройства, которые генерируют прерывания сами по себе — это часы и терминал (а также сеть, если ее поддержка включена). Обслуживание часов сводится к подсчету тиков, соответственно, если прерывание не удалось обработать сразу, это некритично и компенсируется позже. Для обслуживания терминалов вводимые символы помещаются в буфер и устанавливается флаг, говорящий о том, что введены символы. Когда задача терминала работает, перед тем как уснуть, она проверяет значение этого флага, и если есть какая-то работа, откладывает сон.

Задача терминала не пробуждается прерываниями терминала напрямую, так как это привело бы к значительным накладным расходам. Часы отправляют сообщение задаче терминала на следующий тик после того, как произошло прерывание терминала. Печатая со скоростью 100 слов/мин, машинистка вводит за секунду менее десяти символов. Поэтому даже если машинистка работает очень быстро, прерывание будет возникать для каждого символа. Если буфер переполнится, вводимые символы могут быть потеряны, но практика показывает, что буфер в 32 символа вполне удовлетворителен. В случае других устройств ввода скорость поступления может быть в тысячи раз выше, чем скорость работы машинистки, пример тому — последовательный порт, к которому подключен модем со скоростью 28 000 бит/с. На такой скорости между тиками может быть получено около 48 символов, а если используется сжатие данных, скорость передачи данных может возрасти как минимум вдвое. Поэтому для последовательных линий в MINIX предусмотрен буфер объемом 1024 символа.

Мы несколько сожалеем о том, что задачу терминала нельзя реализовать без некоторых уступок нашим общим принципам построения драйверов, но использованный нами компромиссный метод позволяет достичь цели, не слишком усложняя программное обеспечение и без потери производительности. Очевидная альтернатива — отбросить принцип рандеву и заставить систему буферизовать все передаваемые сообщения, но это намного сложнее и медленнее.

Разработчики систем часто сталкиваются с противоречием между тем, чтобы следовать общим правилам, что всегда приводит к более элегантным, но иногда медленным решениям, и тем, чтобы применять более простые методики. Эти методики обычно дают более быстрый код, но зачастую, чтобы заставить его работать, требуются некоторые хитрости. Опыт — единственный критерий, позво-

ляющий выбрать при данных условиях лучший вариант. Немалый опыт разработки операционных систем собран у Lampson (1984) и Brooks (1975). Эти книги, хотя и не новы, остаются классикой.

В завершение нашего обсуждения терминального ввода мы подведем итог и рассмотрим все события, которые происходят, когда задача терминала впервые активизируется по запросу на чтение или когда она повторно получает управление после клавиатурного ввода (рис. 3.25). В первом случае, когда задача терминала получает сообщение, запрашивающее вводимые с клавиатуры символы, главная процедура, tty_task, вызывает do_read, которая и выполняет запрос. Если имеющейся в буфере информации недостаточно для выполнения запроса, процедура сохраняет параметры запроса в tty_table.

После этого, чтобы получить уже введенные данные, вызывается in_transfer, а затем — handle_events, которая, в свою очередь, вызывает kb_read и еще раз in_transfer, чтобы вытянуть из входного потока еще несколько символов. Функция kb_read несколько раз вызывает другие подпрограммы, не показанные на рис. 3.25. Если нет доступных данных, ничего не копируется. Когда вызовы in_transfer или handle_events полностью выполнили запрос, файловой системе отправляется сообщение, чтобы она могла разблокировать запросивший данные процесс. Если чтение не завершено (не введено ни одного символа или введено недостаточное количество), функция do_report информирует об этом файловую систему, чтобы та могла либо заблокировать вызвавший процесс, либо, если было запрошено неблокирующее чтение, отменить запрос.

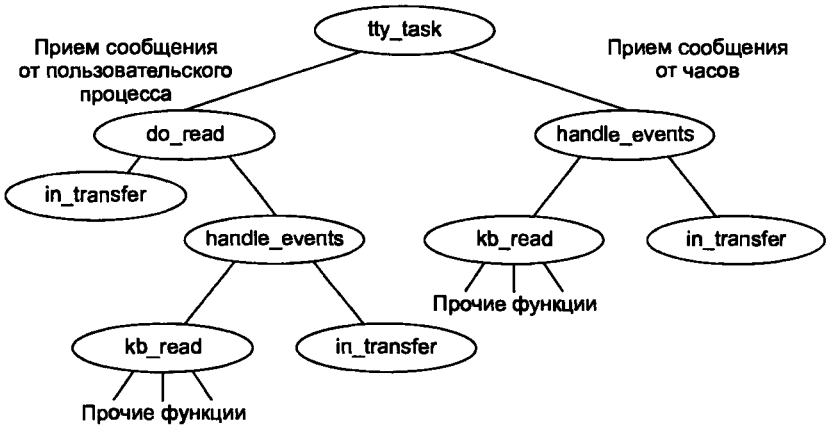


Рис. 3.25. Обработка ввода драйвером терминала. Правая часть соответствует процессу, запросившему символ. Левая часть выбирается, когда драйвер получил сообщение о введенном символе

Правая часть рис. 3.25 подытоживает все события, возникающие, когда задача терминала просыпается вследствие прерывания от клавиатуры. Когда напечатан символ, обработчик прерываний kb_hw_int помещает его код в клавиатурный буфер, устанавливает флаг, позволяющий определить, какому из устройств на-

правлен ввод, и завершает работу. Обработка символа продолжится через небольшой временной интервал, на следующем тике таймера. При этом задача часов отправляет задаче терминала сообщение, говорящее ей о том, что что-то произошло. Получив такое сообщение, tty_task проверяет наличие флага события у всех терминалов. Для каждого устройства, у которого флаг установлен, она вызывает handle_event. Последняя для клавиатуры вызывает функции kb_read и in_transfer, как это делается и при получении исходного запроса на чтение. Отраженные в правой части рисунка события могут происходить несколько раз, до тех пор пока не поступит достаточное для выполнения запроса количество символов. Этот запрос поступает к do_read после первого сообщения от файловой системы. Если файловая система пытается инициировать новый запрос до того, как устройство обслужило предыдущий, возвращается ошибка. Естественно, все устройства независимы; запрос на чтение с удаленного терминала обрабатывается отдельно от запроса чтения с клавиатуры.

Не показанные на рисунке функции, вызываемые kb_read, это: map_key — преобразующая коды клавиш (коды опроса клавиатуры), генерируемые аппаратным обеспечением в ASCII-коды, make_break — отслеживающая состояние клавиш-модификаторов (таких как SHIFT), и in_process — ответственная за такие возможности, как удаление неправильно введенного символа, обработка различных специальных символов и управление параметрами терминала. Кроме того, in_process вызывает функцию echo, чтобы вводимые символы отображались на терминале.

Терминальный вывод

Вообще говоря, консольный вывод проще ввода, так как здесь действиями управляет операционная система, и не нужно оглядываться на то, что события могут произойти в неподходящее время. Если же консолью является отображаемый на память экран, процесс еще больше упрощается. Прерывания не нужны, вывод производится путем копирования данных из одной области памяти в другую. С другой стороны, в этом случае все задачи управления экраном, включая обработку ESC-последовательностей, ложатся на программное обеспечение драйвера. Как и при изучении ввода в предыдущем разделе, мы проследим, какие действия происходят при выводе символа на консольный дисплей. В этом примере мы предполагаем, что вывод производится на активный дисплей. Небольшое усложнение, вносимое виртуальными консолями, мы обсудим позже.

Обычно, когда процесс хочет что-нибудь напечатать, он вызывает printf. Эта функция делает системный вызов write, который отправляет файловой системе сообщение с указателем на символы, подлежащие выводу (но не сами символы). Затем файловая система посылает сообщение драйверу терминала, а тот копирует переданные ему символы в видеопамять. Основные подпрограммы, из которых складывается терминальный вывод, показаны на рис. 3.26.

Когда задача терминала получает сообщение, требующее от нее вывести что-либо на экран, вызывается подпрограмма do_write, которая сохраняет параметры в соответствующей выбранной консоли структуре tty в массиве tty_table. Затем вызывается handle_event (та же функция, которая отрабатывает при обнаружении установленного флага tty_events).

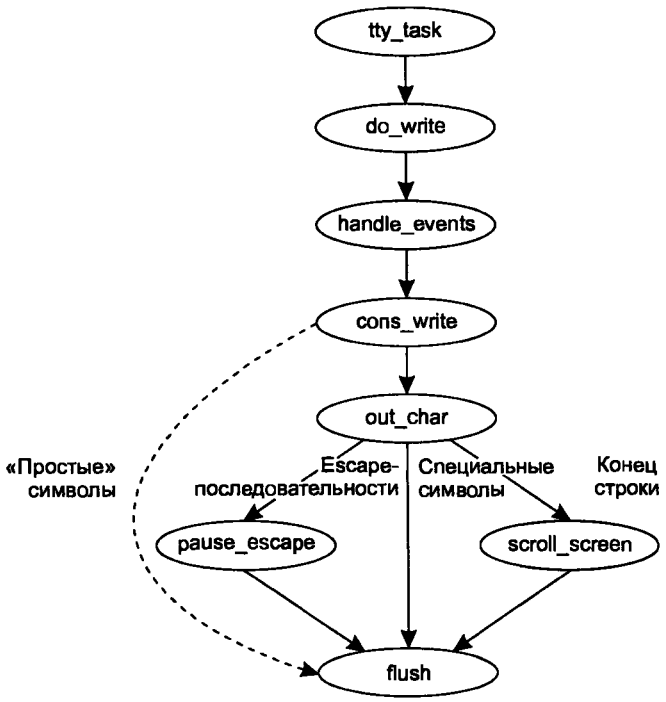


Рис. 3.26. Основные подпрограммы, используемые при выводе на консоль. Штриховой линией обозначено прямое копирование символов в буфер ramqueue подпрограммой cons_write

Эта подпрограмма при каждом вызове выполняет для указанного устройства как запись, так и чтение. В случае с консолью это означает, что сначала будут обработаны еще не принятые клавиатурные данные. Если необработанного ввода нет, переданные символы добавляются к другим символам, ожидающим вывода. Затем вызывается процедура cons_write, выполняющая вывод для отображаемых в память экранов. Последняя использует подпрограмму phys_copy, чтобы скопировать блок символов от пользовательского процесса в локальный буфер, причем, возможно, это действие будет повторено несколько раз, так как локальный буфер вмещает только 64 байта. Когда буфер укомплектовывается, каждый 8-битный байт переносится в другой буфер, ramqueue, представляющий собой массив 16-битных слов. Дополнительный байт заполняется текущим значением байта атрибутов, который определяет цвет символов, цвет фона и некоторые другие параметры текста. Когда это возможно, символы копируются напрямую в ramqueue, но некоторые символы, например управляющие символы, образующие ESC-последовательности, необходимо обрабатывать особо. Кроме того, специальная обработка требуется тогда, когда позиция символа превышает ширину экрана или когда ramqueue заполняется. В этом случае, чтобы передать символы и выполнить необходимые дополнительные действия, вызывается функция out_char. Так, каждый раз когда на последней строке экрана принимается

символ перевода строки, делается вызов подпрограммы scroll_screen, а parse_escape принимает символы, образующие ESC-последовательность. Обычно out_char вызывает подпрограмму flush, которая копирует ramqueue в видеопамять при помощи ассемблерной подпрограммы mem_vid_copy. Функция flush также вызывается после того, как в ramqueue передается последний символ, чтобы гарантировать, что отображены все символы. Конечный результат работы flush — команда, предаваемая чипу видеоконтроллера 6845 с целью отобразить в нужном месте курсор.

Байты, поступающие от пользовательского процесса, было бы логично выводить по одному, в цикле. Тем не менее, для систем класса Pentium с защитой памяти более эффективно предварительно накапливать их в ramqueue, а затем копировать весь блок при помощи вызова mem_vid_copy. Интересно, что в MINIX эта техника появилась в то время, когда система работала на более старых процессорах, не имевших защищенного режима. Предшественник mem_vid_copy решал другую проблему, проблему синхронизации. Дело в том, что у старых экранов запись в видеопамять необходимо было производить при очистке экрана, во время вертикального обратного хода луча, чтобы избежать появления мусора на экране. Сейчас такое устаревшее оборудование не поддерживается, так как это приводит к слишком большим накладным расходам. Тем не менее сама техника копирования целого блока из ramqueue применяется и поныне, хотя и для другой цели.

Доступная консоли область видеопамяти задается полями c_start и c_limit структуры console. Текущее положение курсора хранится в полях c_column и c_row. Координатам (0,0) соответствует верхний левый угол экрана, с этого места аппаратное обеспечение начинает заполнять экран. Изображение начинается с адреса, задаваемого значением c_org, и занимает 80 × 25 символов (4000 байт). Другими словами, чип 6845 извлекает из видеопамяти два байта по адресу c_org и отображает в левом верхнем углу символ, используя второй байт для задания цветов, мигания и прочих атрибутов. Затем чип извлекает из памяти следующее двухбайтовое слово и отображает символ в позиции (1, 0). Этот процесс продолжается до тех пор, пока не будет достигнута позиция (79, 0), после чего начинается рисование второй строки экрана, начиная с координат (0, 1).

При включении компьютера экран очищается, и данные записываются в видеопамять, начиная с положения c_start, а c_orig присваивается то же значение, что и c_start. Таким образом, первая строка текста выводится в верхней строке экрана. Когда вывод переходит на следующую строку, под управлением ли символа перевода строки, либо в результате заполнения верхней строки экрана, новые символы помещаются по адресу, равному c_start плюс 80. Со временем все 25 строк экрана окажутся заполнены и потребуется *прокрутка* экрана. Некоторым программам, например текстовым редакторам, требуется возможность прокручивать экран и в обратном направлении, когда курсор находится в верхней строке экрана и требуется перейти выше по тексту.

Существует два подхода к решению задачи прокрутки экрана. Когда применяется *программная прокрутка*, символу с координатами (0, 0) всегда соответствует один и тот же адрес видеопамяти, с нулевым смещением относительно c_start. Видеоконтроллер всегда выводит этот символ на одном месте, благодаря тому,

что в c_org хранится один и тот же адрес. Когда необходимо выполнить прокрутку, область видеопамяти, начинающаяся со второй строки экрана (смещение 80 относительно c_start), копируется в начало занимаемой консолью области (смещение 0). Положение самой занимаемой экраном области памяти не меняется. В результате содержимое экрана перемещается на одну строку вверх. Эта операция занимает время, необходимое для сдвига 80 × 2000 слов памяти. При *аппаратной прокрутке* данные никуда не перемещаются. Вместо этого контроллер получает инструкцию начинать выводить данные с другого адреса, например с 80-го слова. Для этого новое значение записывается в c_org, чтобы его можно было использовать в дальнейшем, а также копируется в нужный регистр видеоконтроллера. Такая схема будет работать, если контроллер достаточно умен и при достижении конца видеопамяти (адрес в c_limit) переходит на ее начало (адрес в c_start), или же при такой емкости видеопамяти, когда в ней помещается более одного экрана, то есть 80 × 2000 слов. У старых терминальных устройств обычно была небольшая память, зато они умели переходить на начало памяти, достигнув конца, и благодаря этому поддерживали аппаратную прокрутку. У современных контроллеров памяти обычно намного больше, чем это необходимо для хранения одного экрана, но перескакивать на начало они не научены. Так, контроллер с видеопамятью 32 768 байт может хранить в памяти 204 полные строки по 160 байт каждая и может 179 раз выполнить аппаратную прокрутку, прежде чем невозможность прыжка станет проблемой. Затем потребуется копирование области памяти с целью переместить данные с последних 24 строк в начало видеобуфера. Когда это делается, нижняя строка экрана заполняется пробелами для гарантии, что она пуста.

При конфигурировании виртуальных консолей доступная видеопамять поровну делится между всеми консолями, для чего соответствующим образом инициализируются поля c_start и c_limit каждой консоли. Это оказывает влияние на прокрутку. Любому видеоконтроллеру, обладающему достаточно большой для поддержки нескольких виртуальных консолей памятью, время от времени необходима программная прокрутка, хотя номинально поддерживается аппаратная реализация сдвига экрана. Чем меньший объем памяти доступен каждому виртуальному дисплею, тем чаще требуется программная прокрутка. Предел достигается, когда создается максимально возможное количество виртуальных консолей. При этом каждая прокрутка будет программной.

Положение курсора относительно начала видеопамяти можно вычислить, исходя из c_column и c_row, но быстрее хранить его в явном виде (c_cur). Когда на экран выводится символ, он записывается в память по адресу c_cur, который после вывода обновляется, как и значение в c_column. Все поля структуры console, оказывающие влияние на текущее положение вывода и начало экрана в памяти, перечислены в табл. 3.12.

При обработке символов, изменяющих текущую позицию (то есть символов перевода строки, забоя и т. д.), соответственным образом изменяются значения полей c_column, c_row и c_cur. Это делается в конце кода процедуры flush, где находится вызов функции set_6845, которая устанавливает регистры видеоконтроллера.

Таблица 3.12. Поля структуры console, связанные с текущей позицией на экране

Поле	Значение
c_start	Начало видеопамяти для консоли
c_limit	Предел видеопамяти для консоли
c_column	Номер текущего столбца (0–79), считая слева
c_row	Номер текущей строки (0–24), считая сверху
c_cur	Смещение курсора в видеопамяти
c_org	Область памяти, отводимая под экран. В чипе 6845 адресуется регистром базы

Драйвер терминала поддерживает ESC-последовательности, то есть гибкие возможности управления экраном для редакторов и прочих интерактивных программ. Поддерживаемые последовательности являются подмножеством стандарта ANSI и достаточны для простого переноса на MINIX большинства программ, написанных для другого аппаратного обеспечения и других операционных систем. Есть две категории ESC-последовательностей: те, которые никогда не имеют параметров, и те, которые могут содержать варьируемые данные. Единственный представитель первой категории, реализованный в MINIX, это ESC M — последовательность, выполняющая обратный перевод строки. При этом курсор поднимается на одну строку вверх, а если курсор уже находится на верхней строке, экран прокручивается на строку вниз. У последовательностей из второй категории могут быть один или два параметра. Все такие цепочки символов начинаются с ESC [. Символ [является преамбулой ESC-последовательности. Список последовательностей, описываемых стандартом ANSI и поддерживаемых MINIX, приведен в табл. 3.11.

Разбор ESC-последовательностей не так прост. В MINIX корректные последовательности могут содержать от двух символов, как ESC M, до 8, как последовательности с двумя двухразрядными параметрами. Например, последовательность ESC [20;60H перемещает курсор на 20-ю строку, в 60-й столбец. Если параметр один, его можно опускать, а при наличии двух параметров позволено опустить оба. Когда параметр не указывается или указывается значение, превышающее граничное, используется значение по умолчанию, которое равно наименьшему допустимому значению.

Рассмотрим способы того, как при помощи ESC-последовательностей переместить курсор в верхний левый угол экрана.

1. Последовательность ESC [H. Так как оба параметра пропущены, берутся наименьшие значения.
2. Последовательность ESC [1;1H переместит курсор в строку 1, столбец 1 (в ANSI нумерация столбцов и строк начинается с единицы).
3. Опустив только один из параметров, можно сконструировать последовательности ESC [1;H и ESC [;1H. Отсутствующий параметр по умолчанию будет считаться равным 1.

4. К тому же результату приведет и ESC [0;0H, так как в ней оба параметра меньше минимально допустимой величины. Они будут заменены значениями по умолчанию.

Эти примеры не нужно рассматривать как рекомендацию специально использовать некорректные значения параметров, а следует расценивать как иллюстрацию того, что интерпретирующий эти последовательности код нетривиален.

Для разбора ESC-последовательностей в MINIX реализован конечный автомат. Переменная состояния этого автомата c_esc_state в структуре console обычно имеет значение 0. Когда функция out_char обнаруживает символ ESC, она устанавливает переменную состояния в 1, и последующие символы будут интерпретироваться функцией parse_escape. Если следующий полученный символ является преамбулой ESC-последовательности, выполняется переход в состояние 2. В этом состоянии вычисляется значение числовых параметров, пока на вход поступают цифровые символы. А именно: предыдущее значение (которое изначально равно 0) умножается на 10, и к нему добавляется значение полученного числового символа. Значения параметров хранятся в массиве, и когда поступает символ точки с запятой, осуществляется переход к следующей ячейке массива. В MINIX этот массив содержит всего два элемента, но принцип тот же. Когда встречается нечисловой символ помимо точки с запятой, последовательность считается завершенной и снова вызывается do_escape. Полученный символ используется для того, чтобы определить, какое именно действие необходимо выполнить и как интерпретировать параметры, будь то значения по умолчанию или считанные из потока символов данные. Это иллюстрируется рис. 3.28.

Подгружаемые раскладки клавиатуры

Клавиатура IBM PC не генерирует ASCII-коды напрямую. Клавиши идентифицируются по их номерам, начиная с клавиш в верхнем ряду оригинальной клавиатуры IBM PC. Соответственно, клавиша ESC имеет номер 1, «1» имеет номер 2 и т. д. Номера присвоены всем клавишам, включая клавиши-модификаторы, такие как левый и правый SHIFT, которые соответственно имеют номера 42 и 54. Когда нажимается клавиша, MINIX извлекает ее номер из кода опроса клавиатуры. Код опроса генерируется и в том случае, когда клавиша отпускается, но здесь устанавливается старший значащий бит кода (что эквивалентно добавлению числа 128 к номеру). Благодаря этому операционная система может различать, была клавиша нажата или отпущена. Поскольку система отслеживает состояние клавиш-модификаторов, возможно большое число комбинаций. Для большинства целей достаточно двухклавишных сочетаний, таких как SHIFT+A или CTRL+D, удобных для печатающих двумя руками, но в некоторых специальных случаях могут применяться комбинации из трех (и более) клавиш, например CTRL+SHIFT+A или CTRL+ALT+DEL, последняя на PC используется для сброса системы.

Сложность клавиатуры PC имеет следствием большую гибкость ее использования. В стандартном наборе присутствуют 47 клавиш с типовыми символами (26 алфавитных символов, 10 цифр и 11 знаков препинания). Если мы будем поддерживать комбинации из трех клавиш-модификаторов, таких как CTRL+ALT+SHIFT, получается набор из 376 (8 × 47) вариантов. Это — ни коим образом не предел

возможного, но сейчас мы не будем акцентировать различия между левыми и правыми клавишами-модификаторами, а также учитывать функциональные клавиши или клавиши цифровой клавиатуры. Конечно, никто не заставляет нас использовать в качестве модификаторов только клавиши CTRL, SHIFT и ALT. Мы вправе исключить какую-либо клавишу из стандартного набора и использовать ее в качестве модификатора, для чего нужно написать драйвер, который будет поддерживать такую систему.

Чтобы определить, какой код передать программе при нажатии определенной клавиши, операционные системы, работающие с подобными клавиатурами, применяют *клавиатурные раскладки*, иначе *карты клавиш*. В MINIX типичная карта клавиш логически является массивом, имеющим 128 строк для каждого кода опроса (такой размер был выбран ради соответствия японским клавиатурам, у европейских и американских панелей нет такого числа клавиш) и 6 столбцов. Столбцы сопоставлены следующим состояниям: нет модификаторов, нажата клавиша SHIFT, нажата клавиша CTRL, нажата левая клавиша ALT, нажата правая клавиша ALT, нажата комбинация ALT+SHIFT. Такая схема при соответствующей клавиатуре, позволяет генерировать (128 – 6) × 6 = 720 различных кодов символов. Конечно, каждая запись в таблице должна содержать 16-битное значение. У американских клавиатур столбцы ALT и ALT2 идентичны. Для других языков клавиша ALT2 названа ALTGR, и на многих из подобных клавиатур нарисован третий символ на клавише, для ввода которого и используется этот модификатор.

Стандартная карта клавиш встроена в ядро MINIX и определяется строкой

```
#include "keymaps/us-std.src"
```

в файле keyboard.c. Для загрузки альтернативной раскладки по адресу keymap используется системный вызов ioctl следующего вида:

```
ioctl(0, KIOCSMAP, keymap)
```

Полная раскладка занимает 1536 байт (128 × 6 × 2). Дополнительные раскладки хранятся в сжатой форме. Чтобы создать новую сжатую карту, запускается программа genmap. При компиляции в genmap включается код keymap.src для определенной раскладки. Обычно genmap запускается сразу же после того, как скомпилирована, и записывает уплотненную версию в файл, после чего исполняемый файл программы удаляется. Команда loadkeys загружает указанную карту клавиш, выполняет декомпрессию и делает системный вызов ioctl, передавая раскладку в память ядра. MINIX может автоматически вызывать loadkeys при старте. Кроме того, пользователь может в любой момент вызвать эту команду.

Исходный код раскладки описывает большой инициализированный массив. В табл. 3.13 показано содержимое нескольких строк файла src/kernel/keymaps/us-std.src. На клавиатурах IBM PC нет клавиши, генерирующей код опроса 0. Коду 1 соответствует клавиша ESC. Из первой записи в таблице видно, что возвращаемый код для этой клавиши не зависит от состояния модификаторов SHIFT или CTRL, но если удерживается клавиша ALT, при нажатии ESC генерируется другой код.

Таблица 3.13. Несколько записей из файла оригинальной клавиатурной раскладки MINIX

Код опроса	Символ	Обычный	SHIFT	ALT1	ALT2	ALT+SHIFT	CTRL
00	Нет	0	0	0	0	0	0
01	ESC	C('[')	C('[')	CA('[')	CA('[')	CA('[')	C ('[')
02	'1'	'1'	'!'	A('1')	A('1')	A('!')	C('A')
13	'='	'='	'+'	A('=')	A('=')	A('+')	C('@')
16	'q'	L('q')	'Q'	A('q')	A('q')	A('Q')	C('Q')
28	CR/LF	C('M')	C('M')	CA('M')	CA('M')	CA('M')	C('J')
29	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL
59	F1	F1	SF1	AF1	AF1	ASF1	CF1
127	???	0	0	0	0	0	0

Значения, содержащиеся в ячейках таблицы, задаются при помощи макросов из файла include/minix/keymap.h:

```
#define C(c)   ((c)) & 0x1F)          /* Преобразование в управляющий код */
#define A(c)   ((c)) | 0x80)          /* Устанавливается восьмой бит (ALT) */
#define CA(c)  A(C(c))                /* CTRL+ALT */
#define L(c)   ((c) | HASCAPS)        /* Добавляется атрибут «Caps-Lock
                                         оказывает влияние» */
```

Первые три макроса манипулируют битами переданного символа, вырабатывая нужный код. Последний устанавливает бит HASCAPS в старшем байте 16-разрядного кода. Этот флаг указывает на необходимость проверки, фиксируется ли верхний регистр букв (клавиша Caps Lock), и возможную модификацию кода. В табл. 3.13 записи для кодов опроса 2, 13 и 16 иллюстрируют, как обрабатываются типичные клавиши с цифровыми, алфавитными символами и знаками препинания. У кода 28 можно видеть одну особенность. Обычно клавиша ENTER порождает код CR (0x0D), который записывается здесь как C('M'). Так как в UNIX символом новой строки является код LF (0x0A) и иногда его необходимо вводить напрямую, комбинация CTRL+ENTER обрабатывается особым образом и производит код C('J').

Код опроса 29 — это один из кодов модификаторов, который должен распознаваться всегда, поэтому, независимо от состояния прочих модификаторов, возвращается значение CTRL. Функциональные клавиши не генерируют обычных ASCII-символов, и строка таблицы для кода опроса 59 содержит символьную запись значений, возвращаемых при нажатии клавиши F1 в комбинации с различными модификаторами. Это следующие коды: F1: 0x0110, SF1: 0x1010, AF1: 0x0810, ASF1: 0x0C10, CF1: 0x0210. Последняя запись в таблице соответствует коду опроса 127 — типичному для конца массива маркеру. У большинства клавиатур, используемых в Европе и Америке, недостаточно клавиш, чтобы генерировать такие коды, поэтому соответствующие записи в таблице заполняются нулями.

Загружаемые шрифты

В ранних персональных компьютерах шаблоны, по которым генерировались изображения символов на экране, хранились только в ПЗУ. Но у контроллеров, которыми оснащаются современные системы, образы символов можно загружать в оперативную память. В MINIX эта возможность поддерживается с помощью системного вызова ioctl следующего вида:

```
ioctl(0, TIOCFDN, font)
```

MINIX поддерживает видеорежим 80 столбцов на 25 строк, и файлы со шрифтами содержат 4096 байт. Каждый байт такого файла описывает значения восьми пикселов, при этом подсвеченному пикселу соответствует 1, погашенному — 0. Тем не менее контроллер дисплея для хранения изображения каждого символа отводит 32 байта, чтобы работать в более высоких разрешениях, не поддерживаемых сейчас MINIX. Для того чтобы преобразовать такой файл в 8192-байтовую структуру font, адрес которой передается в системный вызов ioctl, предназначена команда loadfont. Как и клавиатурные карты, шрифты могут загружаться при запуске системы или же позднее, в любой момент нормальной работы. Тем не менее загружать шрифт не обязательно, у каждого контроллера в ПЗУ имеется встроенный шрифт, доступный по умолчанию. Встраивать шрифт в саму систему нет необходимости, поэтому единственное, что присутствует в ядре для поддержки загружаемых шрифтов, — это код, выполняющий операцию TIOCSFON системного вызова ioctl.

3.9.4. Реализация аппаратно-независимого драйвера терминала

В этом разделе мы начнем подробное изучение кода драйвера терминала. Рассматривая блочные устройства, мы видели, что драйверы, обслуживающие несколько различных устройств, могут разделять общий базовый код. Драйвер терминала представляет собой похожий случай, с той разницей, что одна задача терминала поддерживает несколько типов терминальных устройств. Мы начнем изучение с кода, не зависящего от конкретных устройств. Позже мы перейдем к коду, работающему с клавиатурой и отображаемым в память экраном.

Структуры данных задачи терминала

В файле tty.h содержатся определения, используемые в коде драйверов терминала. Большая часть объявленных в этом файле переменных можно идентифицировать по префиксу tty_. Кроме того, еще одна такая переменная объявлена в файле glo.h как EXTERN. Это переменная tty_timeout, она используется в обработчиках прерываний как терминала, так и часов.

Определения макросов O_NOCTTY и O_NONBLOCK (значения необязательных флагов для системного вызова open) повторяют определения, имеющиеся в файле include/fcntl.h. Это сделано с той целью, чтобы не присоединять другие заголовочные файлы. Типы devfun_t и devfunarg_t применяются для передачи указа-

телей на функции, которые используются в механизмах косвенного вызова для главного цикла дисковых драйверов, аналогичных рассмотренным нами ранее.

Самое главное описание в файле tty.h — это структура tty. Для каждого терминального устройства, рассматриваемого в целом (консольный дисплей и клавиатура образуют один терминал), существует одна такая структура. Первое ее поле, tty_events, это флаг, который устанавливается в результате прерывания по факту изменений, требующих от задачи терминала уделить внимание устройству. Когда устанавливается этот флаг, также изменяется значение глобальной переменной tty_timeout с целью скомандовать задаче часов пробудить задачу терминала при следующем сигнале часов.

Остальные поля структуры tty объединены в группы, связанные с операциями ввода и вывода, состоянием устройства и информацией о незавершенных операциях. В секции операций ввода первая пара переменных, tty_inhead и tty_intail, задают очередь, в которой буферизуются принимаемые символы. Поле tty_incount является счетчиком количества символов в этой очереди, а в tty_eotct подсчитываются строки символов, как будет описано ниже. Все вызовы, зависящие от конкретного устройства, являются косвенными, за исключением процедур инициализации терминала, которые и устанавливают указатели для косвенных вызовов. Указатели на специфичный для данного терминала код хранятся в полях tty_devread и tty_icancel. Это соответственно указатели на функцию чтения данных и функцию отмены чтения. Значение поля tty_min сравнивается с tty_eotct. Когда значение tty_eotct становится больше или равно значению tty_min, операция чтения считается завершенной. При каноническом вводе tty_min равно 1, и tty_eotct подсчитывает введенные строки. При неканоническом вводе tty_eotct подсчитывает символы, а tty_min присваивается значение из поля MIN структуры termios. Таким образом, сравнение этих двух полей позволяет определить, когда завершен ввод строки или получено минимальное необходимое количество символов (в зависимости от режима).

В поле tty_time хранится значение времени, когда обработчик прерываний таймера должен пробудить задачу часов, а поле tty_timenext используется для того, чтобы объединить в список активные поля tty_time. Каждый раз, когда устанавливается таймер, этот список сортируется по времени, чтобы обработчик прерываний часов мог проверять только первую запись. MINIX поддерживает большое количество удаленных терминалов, из них в любой момент времени только несколько терминалов устанавливают таймер. Поэтому использование списка активных таймеров значительно упрощает работу обработчика прерываний таймера по сравнению с тем, как если бы ему приходилось проверять каждую запись в tty_table.

Поскольку постановка данных в очередь вывода выполняется по-разному для разных устройств, секция вывода структуры tty не содержит переменных, а состоит исключительно из указателей на специфичные для данного устройства функции, служащие для вывода, эхо-отображения, отправки сигнала прерывания и отмены вывода. В секции состояния поля tty_reprint, tty_escaped и tty_inhibited служат флагами, индицирующими, что последний полученный символ несет специальную нагрузку. Так, когда получен символ CTRL+V (LNEXT),

флаг tty_escaped устанавливается в 1, чтобы обозначить, что любое особое значение следующего полученного символа необходимо игнорировать.

Следующая часть структуры хранит информацию о текущих операциях DEV_READ, DEV_WRITE и DEV_IOCTL. Каждая из этих операций затрагивает два процесса. Обслуживающий системный вызов сервер (обычно это файловая система) идентифицируется полем tty_incaller. Сервер вызывает задачу терминала от лица другого процесса, заинтересованного в операции ввода/вывода, и этот процесс идентифицируется полем tty_inproc. Как показано на рис. 3.24, при выполнении read символы копируются напрямую в приемный буфер в адресном пространстве сделавшего вызов процесса. Положение буфера задается полями tty_inproc и tty_in_vir. Две следующие переменные, tty_inleft и tty_incum, фиксируют, сколько еще нужно получить символов и сколько уже получено. Аналогичный набор переменных предусмотрен и для системного вызова write. При выполнении вызова ioctl может потребоваться немедленная передача данных между запрашивающим процессом и задачей, для чего нужен виртуальный адрес, но информацию о состоянии этой операции сохранять не надо. Наконец, структура tty содержит еще несколько переменных, которые не попадают ни в какую другую категорию. Это указатели на функции, выполняющие операции DEV_IOCTL и DEV_CLOSE на уровне устройства, структура termios, описанная в POSIX, и структура winsize, обеспечивающая поддержку экранов с оконным интерфейсом. В последней части структуры зарезервировано место для самой входной очереди в массиве tty_inbuf. Обратите внимание, что это массив значений типа u16_t, а не 8-битных символов char. Хотя приложения и устройства используют для представления символов 8-битные коды, язык C ужесточает требования, считая, что функция getchar должна возвращать данные более длинного типа, чтобы, помимо 256 допустимых значений байта, она могла возвращать код EOF.

Массив tty_table, состоящий из структур tty, объявлен как EXTERN. Для каждого терминала в этом массиве резервируется по одному элементу, а количество элементов определяется константами NR_CONS, NR_RS_LINES и NR_PTYS, заданными в файле include/minix/config.h. В обсуждаемую в книге конфигурацию включены две консоли, но MINIX можно перекомпилировать и включить поддержку до двух последовательных интерфейсов и до 64 псевдотерминалов.

В файле tty.h есть еще одно EXTERN-определение. Переменная tty_timelist представляет собой указатель, который используется таймером для поддержания списка полей tty_time. Файл tty.h включается во многие другие файлы, и область памяти для tty_table и tty_timelist резервируется при компиляции, так же как для EXTERN-переменных, объявленных в файле glo.h.

В конце файла находится текст двух макросов, buflen и bufend. Эти макроопределения многократно используются в коде задачи терминала, делающей много операций копирования в буферы и из них.

Независимый от устройств драйвер терминала

Основные функции задачи терминала и аппаратно-независимые вспомогательные функции все сконцентрированы в tty.c. Так как задача поддерживает большое число различных устройств, локализацию устройства для конкретного

вызова требуется проводить по младшему номеру устройства. Эти номера определяются макросами в начале файла. Далее следуют еще несколько макроопределений. Если устройство не инициализировано, указатели на специфические для данного устройства функции содержат нулевые значения, которые записываются компилятором. Тем самым можно создать макрос tty_active, который возвращает FALSE в случае, когда обнаруживается нулевой указатель. Естественно, код инициализации устройства нельзя вызвать косвенно, собственно, одна из задач такого кода — это инициализация указателей, которые своим присутствием делают возможными косвенные вызовы. Далее по файлу в директивах условной компиляции следуют макроопределения, заменяющие адреса функций инициализации RS-232 или псевдотерминала нулевыми адресами, если соответствующие устройства не включены в конфигурацию. Это позволяет полностью пропустить код для означенных устройств, когда в действительности он не нужен.

Так как терминал настраивается множеством параметров, а в сетевой системе терминалов несколько, декларируется и заполняется значениями по умолчанию структура termios_defaults. Сами значения задаются в файле /include/termios.h. Эта структура копируется в запись tty_table при инициализации или повторной инициализации терминала. Значения по умолчанию для специальных символов показаны в табл. 3.9. В табл. 3.14 представлены значения по умолчанию для различных флагов. Далее в коде следуют строки, где аналогичным образом объявляется структура winsize_defaults. Ее поля инициализируются нулями компилятором C, что подразумевает: «размер окна неизвестен, используйте /etc/termcap».

Таблица 3.14. Значения по умолчанию для флагов структуры termios

Поле	Значение по умолчанию
c_iflag	BRKINT ICRNL IXON IXANY
c_oflag	OPOST ONLCR
c_cflag	CREAD CS8 HUPCL
c_lflag	ISIG IEXTERN ICANON ECHO ECHOE

Точкой входа для задачи терминала является функция tty_task. Перед тем как начать свой главный цикл, она вызывает для каждого имеющегося в конфигурации терминала функцию инициализации tty_init (это делается в цикле в начале функции), а затем отображается сообщение о запуске MINIX. В коде для вывода сообщения используется функция printf, но при компиляции благодаря макроопределению вызовы printf подменяются на вызовы printk. Функция printk использует для вывода функцию putk, входящую в драйвер консоли, поэтому файловая система при выполнении вывода не затрагивается. Это сообщение отправляется напрямую на экран и не может быть перенаправлено.

Главный цикл задачи аналогичен главному циклу любой другой задачи. Принимается сообщение, при помощи оператора switch для каждого типа сообщения вызывается соответствующая функция обработки, после чего генерируется ответное сообщение. Тем не менее есть и некоторые отличия. Прежде всего, много работы делается низкоуровневыми процедурами отработки прерываний, особенно при обработке ввода с терминала. В предыдущем разделе мы видели, что от-

дельные символы, полученные с клавиатуры, принимаются и накапливаются в буфере без отправки сообщения задаче каждый раз, когда поступает символ. Поэтому, прежде чем пытаться получить сообщение, драйвер терминала «пробегает» по всей таблице tty_table, проверяя флаги tp->tty_events. Тогда, если необходимо позаботиться о незавершенных делах, вызывается функция handle_events. И только когда не было найдено ни одного требующего внимания терминала, делается вызов receive. Если сообщение пришло от аппаратного обеспечения, оператор continue «накоротко» замыкает цикл и повторяется проверка произошедших событий.

Далее, эта задача обслуживает несколько устройств. Если получено сообщение от аппаратуры, то требующее обслуживания устройство или устройства определяются путем проверки флагов tp->tty_events. Если прерывание не аппаратное, то, чтобы определить, какое устройство должно ответить на сообщение, используется поле сообщения TTY_LINE. Младший номер устройства декодируется при помощи серии сравнений, и в tp помещается адрес корректной записи в таблице терминалов tty_table. Если устройство является псевдотерминалом, вызывается do_pty (из файла pty.c), после чего главный цикл перезапускается. В этом случае ответное сообщение генерируется функцией do_pty. Конечно, если в конфигурации нет псевдотерминалов, данный вызов при помощи макроса заменяется пустым, описанным ранее. Можно было бы понадеяться на отсутствие попыток обращения к несуществующим устройствам, но всегда проще добавить еще одну упреждающую проверку, чем при ошибке разбираться со всей остальной системой. В ситуации, когда устройство не существует или не сконфигурировано, генерируется ответное сообщение с кодом ENXIO и управление опять передается в начало цикла.

Оставшуюся часть кода составляет то, что мы уже видели в главной функции других задач, оператор switch, проверяющий значение типа сообщения. Для запроса каждого типа вызывается соответствующая функция, do_read, do_write и т. д. В каждом из случаев функции сами создают ответное сообщение, вместо того чтобы возвращать необходимую информацию обратно в главный цикл. Ответное сообщение генерируется в конце цикла только при условии, что было получено сообщение неправильного типа. В этом случае оно содержит код ошибки EINVAL. Так как отправлять сообщения требуется во многих местах кода, реализована общая подпрограмма tty_reply, которая отвечает за детали создания ответного послания.

Если полученное задачей tty_task сообщение имеет корректный тип, а также не является результатом прерывания и не пришло от псевдотерминала, оператор switch в конце главного цикла перенаправляет его одной из функций do_read, do_write, do_ioctl, do_open, do_close или do_cancel. У этих функций есть два аргумента: tp, указатель на структуру tty, и адрес сообщения. Прежде чем рассматривать их по отдельности, мы упомянем некоторые общие положения. Так как задача терминала обслуживает несколько устройств, эти функции должны срабатывать быстро, чтобы главный цикл мог продолжить работу. Тем не менее do_read, do_write и do_ioctl не всегда могут сразу обеспечить выполнение всех запрошенных действий за раз. Чтобы позволить файловой системе обслуживать

другие вызовы, от функций требуется немедленный ответ. Поэтому, если запрос не может быть удовлетворен немедленно, в ответном сообщении в поле состояния доставляется код SUSPEND. На рис. 3.24 это соответствует сообщению, отмеченному (3), и приводит к приостановке процесса-инициатора вызова в то время, как файловая система выходит из блокировки. Сообщения под номерами (7) и (8) будут отправлены позже, после завершения операции. Если операция может быть выполнена полностью или произошла ошибка, то в поле состояния ответного сообщения возвращается либо количество переданных байтов, либо код ошибки. В последнем случае файловая система немедленно отправит сделавшему вызов процессу ответное сообщение, пробуждая его.

Чтение с терминала фундаментально отличается от чтения с диска. Драйвер диска отдает команды аппаратному обеспечению, и в итоге тоже возвращаются данные, если не произошел механический или электрический сбой. В случае терминала компьютер может высветить на экране приглашение, но не существует способа принудить сидящего за клавиатурой человека начать печатать. Хотя бы потому, что нет никакой гарантии, что за клавиатурой кто-то сидит. Чтобы обеспечить быстрый возврат, функция do_read начинает с того, что сохраняет информацию о запросе. Благодаря этому он может выполнен позже, когда прибудет недостающая информация. Сначала должен быть проведен контроль ошибок. Ошибка возникает в случае, если устройство все еще ожидает ввод, чтобы выполнить предыдущий запрос, или если переданные в сообщении параметры неправильны. После того как проверка пройдена, информация готова к записи в поля записи tp->tty_table, соответствующей устройству. Следующим шагом в переменную tp->inleft записывается количество полученных символов. Это важный шаг, так как переменная tp->inleft необходима для определения момента завершения чтения. В каноническом режиме значение tp->inleft декрементируется с каждым возвращенным символом до тех пор, пока не будет получен символ новой строки, после чего эта переменная сразу обнуляется. В неканоническом режиме она изменяется по-другому, но так или иначе, когда запрос выполнен, по тайм-ауту или после получения минимального необходимого количества символов, в эту переменную записывается нуль. Когда значение tp->tty_inleft достигает нуля, отправляется ответное сообщение. Как мы увидим, ответное сообщение может генерироваться в нескольких местах кода. Кроме того, иногда необходимо удостовериться, что выполняющий чтение процесс ожидает ответа. Здесь индикатором служит ненулевое значение tp->tty_inleft.

В каноническом режиме терминал ожидает ввода до тех пор, пока не будет введено запрошенное количество символов или не будет получен символ конца строки или конца файла. Чтобы проверить, находится ли терминал в каноническом режиме, тестируется значение бита ICANON структуры termios. Если бит не установлен, проверяется значение переменных MIN и TIME, с целью определиться в дальнейших действиях. В табл. 3.10 мы видели, как различные комбинации параметров MIN и TIME определяют поведение при чтении с терминала. Сначала проверяется значение параметра TIME. Нулевое значение соответствует левому столбцу таблицы, и в этом случае других проверок в текущий момент не нужно. Если значение TIME не равно нулю, проверяется параметр MIN. Если это нуль, де-

лается вызов settimer для запуска таймера, который по истечении заданного интервала времени прервет запрос DEV_READ, даже если не будет прочитано ни одного байта. В переменную tp->tty_min в этом случае записывается 1, чтобы вызов завершился немедленно после того, как до истечения тайм-аута будет получен хотя бы один символ. В этой точке не проверяется, имеются ли уже введенные символы, поэтому может оказаться, что запроса уже ожидают несколько символов. Как только обнаруживаются уже введенные символы, они возвращаются процессу, но не больше, чем задано при вызове read. В том случае, если и MIN и TIME не равны нулю, таймер ведет себя по-другому — отсчитывает время между символами. Он запускается только после того, как введен первый символ, и перезапускается с каждым новым. В переменной tp->tty_eotct в неканоническом режиме подсчитываются введенные символы. Ее значение проверяется, и при равенстве нулю (ни одного символа еще не получено) таймер останавливается. Оба вызова settimer обрамлены вызовами lock и unlock, чтобы предотвратить прерывания от часов в то время, когда работает settimer.

В любом случае после этого блока условных операторов вызывается функция in_transfer, чтобы напрямую передать уже имеющиеся во входной очереди символы на чтение процессу. Затем следует вызов handle_events, который, в свою очередь, может поместить новые данные во входную очередь и который снова вызывает in_transfer. Такое кажущееся дублирование вызовов требует дополнительных пояснений. Хотя до сих пор обсуждение велось в терминах клавиатурного ввода, функция do_read не зависит от конкретных устройств и обслуживает в том числе удаленные терминалы, подключаемые по последовательной линии. Возможна ситуация, когда предыдущий поток ввода заполнил буфер, и ввод был приостановлен. Первый вызов in_transfer не запускает поток, но вызов handle_events может это сделать. То, что он при этом еще раз вызывает in_transfer, это всего лишь небольшой довесок. Нам важно, что удаленный терминал снова получит разрешение отправлять данные. Оба вызова способны привести к тому, что запрос будет выполнен и файловой системе будет отправлено ответное сообщение. В качестве флага-индикатора отправки ответного сообщения используется переменная tp->tty_inleft. Если ее значение не равно нулю, do_read самостоятельно генерирует и отправляет ответ. За это отвечают последние девять строк кода функции. Если в исходном запросе было указано неблокирующее чтение, файловой системе дается указание вернуть код ошибки EAGAIN сделавшему вызов процессу. Если в запросе было затребовано обыкновенное блокирующее чтение, файловая система получает код SUSPEND, который разблокирует ее, но дает указание оставить процесс-инициатор вызова заблокированным. В данном случае в поле tp->tty_inrepcode записывается значение REVIVE. Когда позднее вызов read будет выполнен (если вообще будет выполнен), этот код помещается в ответное сообщение файловой системе, с целью информировать ее о том, что запросивший чтение процесс был приостановлен и должен быть разбужен.

Функция do_write подобна do_read, но проще, так как при обработке системного вызова read нужно учитывать меньшее количество разнообразных настроек. Сначала делается ряд проверок, подобным проверкам в do_read, из которых мы узнаем, что предыдущий запрос уже обработан и переданные параметры кор-

ректны. Затем параметры сообщения копируются в структуру tty. После этого вызывается функция handle_events и проверяется значение переменной tp->tty_outleft, чтобы узнать, завершена ли работа. Если да, значит, ответное сообщение уже было отправлено handle_events и делать больше нечего. Если нет, генерируется ответ, параметры которого зависят от того, был ли исходный вызов write блокирующим или нет.

Таблица 3.15. Вызовы POSIX и операции ioctl

Функция POSIX	Операция POSIX	Тип ioctl	Параметр ioctl
tcdrain	(нет)	TCDRAIN	(нет)
tcflow	TCOOFF	TCFLOW	int=TCOOFF
tcflow	TCOON	TCFLOW	int=TCOON
tcflow	TCIOFF	TCFLOW	int=TCIOFF
tcflow	TCION	TCFLOW	int=TCION
tcflush	TCIFLUSH	TCFLSH	int=TCIFLUSH
tcflush	TCOFLUSH	TCFLSH	int=TCOFLUSH
tcflush	TCIOFLUSH	TCFLSH	int=TCIOFLUSH
tcgetattr	(нет)	TCGETS	termios
tcsetattr	TCSANOW	TCSETS	termios
tcsetattr	TCSADRAIN	TCSETSW	termios
tcsetattr	TCSAFLUSH	TCSETSF	termios
tcsendbreak	(нет)	TCSBRK	int=duration

Далее следует довольно большая, но не сложная для понимания функция, do_ioctl. Тело этой функции состоит из двух операторов switch. Первый определяет размер параметра, на который ссылается указатель в сообщении. Если размер параметра не равен нулю, проверяется его корректность. В этом месте нельзя контролировать содержимое, но можно удостовериться, что структура имеет требуемый размер, начинается по указанному адресу и умещается в том сегменте, где она должна находиться. Остаток функции составляет еще один оператор switch, проверяющий тип запрошенной операции ioctl. К несчастью, поддержка требуемых стандартом POSIX операций подразумевает, что для операций ioctl необходимо изобретать имена, которые напоминают, но не дублируют POSIX-имена. Соответствие между именами запросов POSIX и параметрами вызова ioctl в MINIX показано в табл. 3.15. Операция TCGETS обслуживает пользовательский вызов tcgetattr и просто возвращает копию структуры tp->tty_termios для терминального устройства. Следующие четыре типа запросов разделяют общий код. Типы запросов TCSETSW, TCSETSF и TCSETS соответствуют вызовам функции POSIX tcsetattr, и основное действие в этом случае общее: структура termios копируется в структуру терминала tty. Для вызова TCSETS копирование происходит немедленно, а для TCSETSW и TCSETSF выполняется после того, как завершен вывод. Копирование производится вызовом phys_copy, следующим за вызовом setattr. Если вызов tcsetattr был сделан с модификатором, требующим отложить выполнение до момента, пока не будет завершен текущий вывод, то, если проверка перемен-

ной tp->tty_outleft показывает, что вывод действительно еще не завершен, параметры запроса помещаются в структуру tty терминала для последующей обработки. Вызов tcdrain приостанавливает программу до тех пор, пока не будет завершен вывод. Он транслируется в вызов ioctl типа TCDRAIN. Если вывод уже завершен, дальнейшей обработки не требуется. Иначе информация также записывается в структуру tty.

Функция POSIX tcflush отменяет незавершенный ввод и/или неотправленные на вывод данные, в зависимости от переданного аргумента. Соответствующая операция ioctl состоит из вызова tty_icancel, который обслуживает все терминалы, и/или вызова зависящей от устройства функции, адрес которой хранится в tp->tty_ocancel. Вызов tcflow аналогичным образом транслируется в вызов ioctl. Чтобы приостановить или вновь разрешить вывод, в переменную tp->tty_inhibited записывается значение TRUE или FALSE, а затем устанавливается флаг tp->tty_events. Чтобы приостановить или вновь разрешить ввод, удаленному терминалу отправляется, соответственно, код STOP (обычно CTRL+S) или START (CTRL+Q). Для этого используется специфичная для устройства подпрограмма, адрес которой хранится в tp->tty_echo.

Большая часть остальных операций, обрабатываемых do_ioctl, выполняются одной строчкой кода, то есть просто вызовом соответствующей функции. В случае операций KIOCSMAP (загрузка раскладки клавиатуры) и TIOCSFON (загрузка шрифта) делается проверка того, что устройство действительно является консолью, так как эти операции неприменимы к другим типам терминалов. Если используются виртуальные терминалы, на всех консолях будет одна раскладка клавиатуры и шрифт. Аппаратное обеспечение не позволяет легко изменить это поведение. Операции, работающие с размером окна, переносят данные структуры winsize между пользовательским процессом и задачей терминала. Обратите особое внимание на комментарий под кодом для операции TIOCSWINSZ. Когда процесс меняет размер окна, от ядра ожидается, что оно отправит сигнал SIGWINCH всем процессам из той же группы процессов. По стандарту POSIX сигнал не требуется, но любой, кто хочет использовать эти структуры, должен подумать о том, чтобы добавить сюда код, инициирующий сигнал.

Последние два варианта из do_ioctl поддерживают функции tcgetpgrp и tcsetpgrp стандарта POSIX. С этими типами не связано никаких действий, и они всегда возвращают ошибку. И здесь нет ничего неправильного. Эти функции необходимы для поддержки *управления задачами*, для возможности приостанавливать и перезапускать процесс с клавиатуры. Управление задачами не требуется POSIX и не поддерживается в MINIX. Тем не менее, по стандарту POSIX, эти функции должны иметься даже в отсутствие управления задачами, с целью гарантировать переносимость программ.

Основная задача функции do_open проста — она увеличивает значение переменной tp->tty_openct для устройства, чтобы впоследствии можно было судить, открыто оно или нет. Вместе с тем сначала нужно провести несколько проверок. Согласно стандарту POSIX, первый процесс, открывающий терминал, должен быть *лидером сеанса*, и, когда этот процесс завершается, у других процессов из той же группы отбирается доступ к терминалу. Демонам необходимо иметь возможность

выводить сообщения об ошибках, и если вывод демонов не перенаправлен в файл, он должен поступать на экран, который нельзя закрыть. Соответственно, в MINIX существует устройство /dev/log. Физически это то же устройство, что и /dev/console, но оно адресуется отдельным младшим номером и обрабатывается иначе. Это устройство только для записи, поэтому do_open возвращает ошибку EACCESS, если сделана попытка открыть его для чтения. Еще одна проверка, которую делает do_open, это проверка флага O_NOCTTY. Если флаг не установлен и устройство не является /dev/log, терминал становится управляющим терминалом для данной группы процессов. Для чего номер процесса, сделавшего вызов, заносится в поле tp->tty_pgrp записи таблицы tty_table. Затем инкрементируется переменная tp->tty_openct и отправляется ответное сообщение.

Терминальное устройство может быть открыто несколько раз, и потом функции do_close нечего делать, кроме как уменьшить tp->tty_openct на единицу. Следующая проверка препятствует закрытию устройства, если это /dev/log. Если обнаруживается, что устройство закрыто в последний раз, для отмены ввода вызывается tp->tty_icancel. Также вызываются специфические для устройства функции tp->tty_ocancel и tp->tty_close. Различным полям структуры tty закрытого устройства присваиваются значения по умолчанию и отправляется ответное сообщение.

Последней из обработчиков сообщений является функция do_cancel. Она вызывается, когда для блокированного процесса, пытающегося выполнить чтение или запись, получен сигнал. Возможны три различных состояния, подлежащие контролю.

1. Когда процесс был завершен, он мог выполнять чтение.
2. Когда процесс был завершен, он мог выполнять запись.
3. Процесс мог быть приостановлен tcdrain до тех пор, пока его вывод не будет завершен.

Для каждого из этих случаев делается проверка и вызывается основная функция, tp->icancel, или специфическая для устройства функция, адрес которой хранится в поле tp->ocancel. В последнем случае единственное, что требуется, — сбросить флаг tp->tty_ioreq, тем самым обозначив, что операция ioctl завершена. В завершение устанавливается флаг tp->tty_events и отправляется ответное сообщение.

Код поддержки драйвера терминала

Мы рассмотрели функции верхнего уровня, вызываемые в главном цикле tty_task, а сейчас настала пора обратиться к коду, обеспечивающему их работу. Мы начнем с функции handle_events. Как упоминалось ранее, при каждом проходе главного цикла задачи терминала для каждого терминального устройства проверяется флаг tp->tty_events, и если терминалу требуется внимание, вызывается handle_events. Подпрограммы do_read и do_write также вызывают handle_events. Эта подпрограмма должна выполняться быстро. Она сбрасывает флаг tp->tty_events и обращается к специфическим для устройства функциям записи или чтения, опираясь на указатели tp->tty_devread и tp->tty_devwrite. Функции вызываются без усло-

вий, так как нет возможности проверить, что вызвало установку флага, чтение или запись. При разработке было решено, что проверять для каждого устройства два флага — более дорогостоящая операция, чем вызывать две функции в том случае, если устройство активно. Кроме того, большую часть времени отображается эхо вводимых символов, и полученный от терминала символ должен быть выведен на экран, поэтому необходимы оба вызова. Как упоминалось при обсуждении обработки tcsetattr в do_ioctl, POSIX не запрещает откладывать выполнение управляющих операций до тех пор, пока не будет завершен текущий вывод, поэтому сразу после вызова tty_devwrite имеет смысл позаботиться об операциях ioctl. Это делается последующими строками, где, если имеются текущие управляющие запросы, вызывается dev_ioctl.

Так как флаг tp->tty_events устанавливается прерываниями и от быстрого устройства символы могут поступать с большой скоростью, существует вероятность, что за время выполнения функций чтения и записи, а также dev_ioctl, произошло другое прерывание и флаг снова возведен. Извлечению данных из буфера, куда они помещаются обработчиком прерываний, придается большой приоритет. Поэтому handle_events повторяет вызов функций чтения и записи, если обнаруживается, что флаг снова установлен. Когда входной поток останавливается (это может быть и выходной поток, но для входа более характерны подобные повторяющиеся запросы), чтобы передать символы из входной очереди в буфер сделавшего вызов процесса, вызывается in_transfer. Эта функция сама отправляет ответное сообщение, если переданные символы завершают запрос (в каноническом режиме такое может произойти в том случае, когда получено запрошенное количество символов, либо когда получен символ конца строки, либо когда достигнут конец файла). Если это произошло, переменная tp->tty_left после возврата в handle_events будет равна нулю. Соответствующий контроль выполняется, и, если количество переданных символов достигло необходимого минимума, отправляется ответное сообщение.

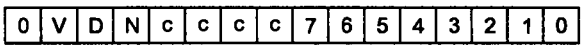
Далее перейдем к функции in_transfer, которая отвечает за передачу данных из входной очереди в адресном пространстве задачи в буфер пользовательского процесса, запросившего ввод. Напрямую копировать блок данных в этом случае невозможно. Входная очередь представляет собой круговой буфер, символы в которой необходимо проверять на признак конца файла или, если действует канонический режим, проверять, что ввод продолжается с новой строки. Кроме того, символы во входной очереди 16-битные, а буфер принимающего процесса является массивом 8-битных символов. Поэтому используется промежуточный локальный буфер. Символы один за одним проверяются и помещаются в буфер, и когда тот заполняется, вызывается функция phys_copy, которая переносит содержимое промежуточного буфера в приемный буфер пользовательского процесса.

Для того чтобы определить, есть ли работа для in_transfer, задействуются три переменные: tp->tty_inleft, tp->tty_eotct и tp->tty_min. Первые две управляют главным циклом процедуры. Ранее упоминалось, что в tp->tty_inleft изначально помещается количество символов, запрошенных в вызове read. Обычно эта переменная уменьшается на единицу с каждым новым полученным символом, но она

может внезапно быть сброшена в ноль, если достигается состояние, сигнализирующее о завершении ввода. Когда переменная становится равной нулю, генерируется ответное сообщение процессу, запросившему чтение. Поэтому она также используется как индикатор отправки сообщения. И тогда, когда в начале кода in_transfer оказывается, что переменная tp->tty_inleft уже равна нулю, это вполне веская причина, чтобы прервать выполнение без ответного сообщения.

Во второй части условия сравниваются переменные tp->tty_eotct и tp->tty_min. В каноническом режиме обе переменные относятся к количеству полных введенных строк, а в неканоническом режиме — имеют отношение к символам. Переменная tp->tty_eotct увеличивается на единицу с каждым занесенным во входную очередь «переносом строки» или байтом и уменьшается подпрограммой in_transfer, когда строка или байт удаляются из очереди. Таким образом, эта переменная подсчитывает количество строк или символов, полученных задачей терминала, но еще не переданных процессу. Поле tp->tty_min хранит значение минимального числа строк (в каноническом режиме) или символов (в неканоническом), которое должно быть передано для удовлетворения запроса. В каноническом режиме это значение всегда равно 1, а в неканоническом оно может принадлежать интервалу от 0 до MAX_INPUT (в MINIX эта константа равна 255). Тем самым вторая половина проверки приводит в каноническом режиме к тому, что подпрограмма завершается сразу же, если еще не получена хотя бы одна целая строка. Передача производится после получения всей строки, потому содержимое очереди можно изменить, если, например, пользователь ввел символ KILL или ERASE до нажатия ENTER. В неканоническом режиме немедленный возврат имеет место, если запрошенное количество символов еще недоступно.

Несколькими строками далее переменные tp->tty_inleft и tp->tty_eotct управляют главным циклом процедуры. В каноническом режиме передача производится до тех пор, пока в очереди не останется больше полностью введенных строк. В неканоническом в поле tp->tty_eotct подсчитывается количество ожидающих передачи символов. Переменная tp->tty_min необходима, чтобы определить, нужно ли входить в цикл, но в определении условия завершения цикла не участвует. После того как цикл начался, передаются либо все полученные символы, либо запрошенное количество, в зависимости от того, что меньше.



V: IN_ESC после LNEXT (Ctrl + V)
D: IN_EOF, конец файла (Ctrl + D)
N: IN_EOT, разрыв строки (NL и др.)
cccc: счетчик отображения символов
7: Бит 7, может быть обнулен, если установлен ISTRIP
6-0: Биты 0–6, код ASCII

Рис. 3.27. Поля кода символа, находящегося во входной очереди

Во входной очереди символы представлены в виде 16-разрядных величин. Пользовательскому процессу передаются только младшие 8 бит из 16. Назначе-

ние старших битов кода иллюстрируется рис. 3.27. Здесь есть флаги, показывающие, входит ли символ в ESC-последовательность (CTRL+V), означает ли он конец файла или содержит один из кодов, сопоставленных концу строки.

Четыре бита используются для подсчета символов, чтобы знать, сколько места осталось для вывода эха. Первый из условных операторов внутри цикла проверяет, установлен ли признак конца файла (D на рисунке). Эта проверка делается в начале потому, что символ конца файла не должен ни передаваться программе, ни использоваться при подсчете символов. При передаче каждого символа у него маскируется старший бит, и в локальный буфер записывается только ASCII-значение.

Существует несколько способов сигнализировать об окончании ввода, но от специфической для устройства функции ввода ожидается, что она распознает, когда вводимый символ является переносом строки, CTRL+D или одним из подобных символов, и соответствующим образом отметит их. Функции in_transfer остается только проверить эту отметку, бит IN_EOT (на рис. 3.27 это N). Если символ опознан, значение tp->tty_eotct уменьшается на единицу. В неканоническом режиме подобным образом подсчитываются все символы, и все они отмечаются битом IN_EOT, поэтому значение tp->tty_eotct уменьшается с каждым полученным символом. Следовательно, единственное различие в работе главного цикла процедуры в двух разных режимах работы терминала можно обнаружить только в последних его строках. Там обнуляется переменная tp->tty_eotct, когда получен символ, отмеченный как разрыв строки, но делается это лишь тогда, когда действует канонический режим. Таким образом, когда управление вновь передается в начало цикла, тот корректно завершается после символа разрыва строки. Но сказанное верно только для канонического режима — в неканоническом разрывы строк игнорируются.

Когда цикл завершается, локальный буфер символов для передачи обычно частично полон. В том случае, если tp->tty_inleft достигла нуля, генерируется и отправляется ответное сообщение. В каноническом режиме это делается всегда, а в неканоническом число символов в буфере сравнивается с минимальным передаваемым количеством, и если символов недостаточно, ответное сообщение не отправляется. Если у вас достаточно хорошая память, чтобы запомнить вызовы in_transfer (в do_read и handle_events), вы, наверно, будете удивлены. Там код, следующий за вызовом, отправлял ответное сообщение, когда in_transfer передала больше символов, чем указано в tp->tty_min. И здесь именно этот случай. Причина, по которой ответное сообщение не отправляется непосредственно из in_transfer, будет раскрыта чуть позже, при обсуждении следующей функции, вызывающей in_transfer при других обстоятельствах.

Эта следующая функция носит имя in_process. Она вызывается из аппаратно-зависимого кода для выполнения общих действий, необходимых всегда. Ее аргументами являются указатель на структуру tty для исходного устройства, указатель на массив 8-битных символов, подлежащих обработке, и счетчик. Новое значение счетчика возвращается в вызвавшую программу. Функция in_process весьма велика, но ее работа не сложна. Она добавляет 16-разрядные символы во входную очередь, откуда их в дальнейшем извлечет in_transfer.

Функцией in_transfer символы разделяются на несколько категорий.

1. Нормальные символы, которые добавляются в очередь, расширенные до 16 бит.
2. Символы, управляющие дальнейшей обработкой, модифицируют флаги, но сами в очередь не помещаются.
3. Символы для эхо-отображения применяются немедленно, без занесения в очередь.
4. Символы, имеющие специальное значение, заносятся в очередь с установленными специальными битами, например, такими, как бит EOT.

Сначала рассмотрим совершенно обычную ситуацию, когда совершенно обычный символ, такой как «x» (ASCII-код 0x78), вводится в середине короткой строки, без влияния ESC-кодов, на терминале, «настроенном» по умолчанию. Полученный от устройства ввода, этот символ занимает биты от 0 до 7 на рис. 3.27. В первом условном операторе внутри цикла старший значащий бит сбрасывается в ноль, если установлен бит ISTRIP. Но по умолчанию MINIX не обнуляет старший бит, позволяя вводить 8-битные символы. В любом случае символ «x» это не затронет. Расширенная обработка ввода по умолчанию в MINIX не разрешена, поэтому проверка бита IEXTERN в переменной tp->tty_termios.c_lflags делается, но последующие проверки вследствие установленных нами условий (не действует ESC-код, сам символ не является ESC-кодом и это не символ REPRINT) не выполняются.

Проверки в следующих нескольких строках обнаруживают, что символ не является специальным символом _POSIX_VDISABLE, ни CR, ни NL. Наконец, одна проверка успешна: действует канонический режим, и символ не является специальным. Рассматриваемая нами буква «x» не является ни символом ERASE, ни KILL, ни EOF (CTRL+D), NL или EOL, поэтому первый тест на бит IXON дает отрицательный результат. Этот бит, разрешающий обработку символов START (CTRL+S) и STOP (CTRL+Q), по умолчанию установлен, но «x» к таковым, очевидно, не относится. Далее обнаруживается, что установлен бит ISIG, разрешающий обработку INTR и QUIT, но, опять же, совпадения в данном случае не обнаруживается.

Фактически первое интересное событие для обычного символа происходит в одной из последних проверок, где выясняется, заполнилась ли уже входная очередь. В этом случае символ игнорируется и, так как активен канонический режим, пользователь не увидит эха символа на экране. Игнорирование символа обеспечивает оператор continue, который вызывает переход в начало цикла. В данном случае рассматриваем рядовые условия выполнения, поэтому предположим, что буфер еще не полон. Не проходит и следующий тест, выделяющий специальный неканонический режим работы. Поэтому управление передается на строку, содержащую вызов функции echo, которая, так как по умолчанию флаг tp->tty_termios.c_lflag установлен, показывает пользователю введенный символ.

Наконец, в следующих строках символ подготавливается к занесению во входную очередь. В этот момент увеличивается значение tp->tty_incount, а tp->tty_eotct не изменяется, так как это обычный символ, не отмеченный битом EOT.

Последняя строка в цикле вызывает функцию in_transfer в том случае, если только что помещенный в очередь символ привел к ее заполнению. В обычных условиях, в которых протекает действие, in_transfer ничего бы не сделала, поскольку значение tp->tty_eotct равно нулю, tp->tty_min — единице (предполагается, что очередь обслуживалась нормально и предшествующий ввод был принят после завершения строки), и последняя проверка привела бы к немедленному возврату.

Итак, рассмотрев работу in_transfer для обычного символа в стандартных условиях, мы вновь вернемся к нашему началу, но теперь взглянем, что происходит в менее обыденной ситуации. Первым будет на рассмотрении символ escape, который позволяет символам, в обычной ситуации имеющим специальное значение, передаваться пользовательскому процессу. Действие этого символа включается установкой флага tp->tty_escaped. Когда ESC-режим активен, флаг сбрасывается, а к текущему символу добавляется бит IN_ESC, на рис. 3.27 обозначенный как V. Это вызывает дополнительную обработку при выводе эха символа на экран; чтобы сделать такие символы видимыми, они отображаются как «^» плюс сам символ. Кроме того, бит IN_ESC предотвращает интерпретацию символа как специального. Следующие несколько строк проверяют сам ESC-символ, LNEXT (по умолчанию CTRL+V). При его обнаружении устанавливается флаг tp->tty_escaped и дважды вызывается rawecho, чтобы вывести «^» и примыкающий символ забоя. Это напоминает пользователю, что действует ESC-режим, а последующий ввод затирает «^». Символ LNEXT — пример символа, влияющего на обработку последующего ввода (в данном случае затрагивается только один последующий символ). Такой символ не заносится в очередь, и после двух вызовов rawecho цикл переходит в начало. Порядок указанных двух проверок важен, так как им обеспечивается возможность передать LNEXT пользовательскому процессу. Для этого символ необходимо ввести дважды.

Следующий специальный символ, обрабатываемый in_process, — это REPRINT (CTRL+R). Обнаружение этого символа инициирует вызов reprint, что приводит к повторному отображению выведенного эха. В дальнейшем сам символ REPRINT игнорируется и не оказывает влияния на входную очередь.

Изучать, как обрабатываются все остальные специальные символы, было бы утомительно, а исходные коды функции in_process достаточно прямолинейны. Поэтому мы упомянем лишь еще несколько деталей. Во-первых, использование специальных битов из старшего байта 16-разрядного значения во входной очереди позволяет легко разбивать символы на классы, имеющие сходный эффект. Так, все символы EOT (CTRL+D), LF и альтернативный символ EOL (по умолчанию не задан) отмечаются битом EOT, на рис. 3.27 это бит D, что позволяет их легко распознавать. Наконец, мы объясним странное поведение in_transfer, описанное ранее. Ответное сообщение не генерируется каждый раз при завершении функции, хотя, казалось бы, после вызова in_transfer всегда следует код, генерирующий ответ. Вспомните, что вызов in_transfer, делаемый в in_process, когда очередь полна, не имеет эффекта в каноническом режиме. Но если требуется неканоническая обработка, каждый символ помечается битом EOT, и, таким образом, tp->tty_eotct подсчитывает все символы. В свою очередь, это приводит к переходу

в главный цикл in_transfer при ее вызове, когда вызов делается по причине заполнения очереди в неканоническом режиме. В таком случае не нужно после возврата из in_transfer передавать задаче терминала никаких сообщений, ведь после этого наверняка будут считаны еще символы. Конечно, хотя в неканоническом режиме отдельный вызов read ограничен размером входной очереди (в MINIX — 255 символов), в неканоническом режиме read должна быть способна передать количество символов по POSIX, задаваемое константой _POSIX_SSIZE_MAX. В MINIX это значение равно 3267.

Несколько следующих функций из tty.c обеспечивают поддержку ввода символов. Функция echo особым образом обрабатывает некоторые символы, но большинство из них просто отображаются ею на выводе того терминала, который используется для ввода. Когда вывод от пользовательского процесса направлен на устройство, может оказаться, что в этот момент на устройство производится эхо-отображение. Когда пользователь попытается удалить последний введенный символ, это усложнит дело. Чтобы решить проблему когда производится обычный вывод, флаг tp->tty_reprint всегда устанавливается аппаратно-зависимыми функциями вывода. Поэтому функция, которая вызывается для обработки символа забоя, может сообщить, что к экрану применен смешанный вывод. Поскольку в echo для вывода символов также привлекаются аппаратные процедуры вывода, текущее значение флага tp->tty_reprint сохраняется в локальной переменной rp и восстанавливается после вывода. Исключение составляет случай, когда только что начался ввод новой строки. Здесь переменной rp присваивается значение FALSE, тем самым обеспечивается, что флаг tp->tty_reprint будет сброшен по завершении выполнения echo.

Вы могли заметить, что функция echo возвращает значение. Например, вы могли встретить такую запись в коде in_process:

```
ch = echo(tp, ch);
```

Возвращаемое значение говорит о том, сколько знакомест занял на экране вывод, и может достигать восьми в случае символа TAB. Эти символы подсчитываются в поле, которое на рис. 3.27 обозначено как cccc. Обычные символы занимают одно знакоместо, но специальные (за исключением TAB, CR, NL или DEL (0x7F)) требуют два, они отображаются как «^» и печатный ASCII-символ. С другой стороны, NL или CR «занимают» ноль знакомест. Конечно же, сам вывод эха должен осуществляться при помощи аппаратно-зависимых подпрограмм, и когда устройству нужно передать символ, делается косвенный вызов подпрограммы, адрес которой хранится в tp->tty_echo, как делается, например, для обычных символов.

Следующая функция, rawecho, вызывается тогда, когда необходимо обойти специальную обработку, присущую echo. Она проверяет, установлен ли флаг ECHO, и если да, отправляет введенный символ аппаратно-зависимой подпрограмме tp->tty_echo без всякой дополнительной обработки. Локальная переменная rp используется для того, чтобы сохранить значение флага tp->tty_reprint.

Когда in_process обнаруживает символ забоя, вызывается функция backover. Она манипулирует содержимым входной очереди, пытаясь по возможности уда-

лить из нее последний введенный символ, то есть если очередь не пуста и последний символ не является переносом строки. Здесь проверяется значение флага tp->tty_reprint, упоминавшегося при обсуждении функции echo и rawecho. Когда этот флаг равен TRUE, это влечет вызов reprint, чтобы показать на экране чистую копию редактируемой строки. Чтобы узнать, сколько знакомест необходимо очистить, проверяется значение поля len у последнего введенного символа (на рис. 3.27 это поле cccc), и для каждого знакоместа при помощи rawecho выводится последовательность «забой-пробел-забой», стирающая символ с экрана.

Следующая функция имеет имя reprint. В дополнение к тому, что ее вызывает backover, она может вызываться, когда пользователь нажимает клавишу REPRINT (CTRL+R). Цикл в коде этой функции сканирует входную очередь в обратном направлении до последнего «разрыва строки». Если разрыв строки является последним введенным символом, значит, для функции нет поля деятельности, и она заканчивает работу. Если это не так, сначала функция выводит на экран символ CTRL+R, который отображается как «^R», переходит на следующую строку и повторно показывает содержимое очереди, начиная с последнего символа разрыва строки до конца.

Вот теперь мы добрались до out_process. Как и in_process, она вызывает аппаратно-зависимые функции вывода, но устроена проще. Сама эта функция вызывается специфическими подпрограммами вывода, действующими при работе последовательного интерфейса RS-232 и псевдотерминала, но не требуется для подпрограмм консоли. Эта функция обрабатывает круговой байтовый буфер, но не удаляет из него символы. Единственное, что она меняет в массиве, — вставляет перед символами NL символы CR, если установлены биты OPOST (разрешение обработки вывода) и ONLCR (преобразование NL в CRLF) в поле tp->tty_termios.oflag. В MINIX по умолчанию эти биты установлены. Работа функции заключается в поддержании значения переменной tp->tty_position в структуре tty устройства. Усложняют жизнь символы табуляции и забоя.

Далее следует подпрограмма dev_ioctl. Она поддерживает do_ioctl, выполняя функции tcdrain и tcflush, когда do_ioctl вызывается с аргументом TCSADRAIN и TCSAFLUSH. Вызов do_ioctl не может немедленно начать выполнение в том случае, если вывод еще не завершен, поэтому информация о запросе сохраняется в полях структуры tty, зарезервированных для этой цели. Когда запустится handle_ events, она после вызова аппаратно-зависимой подпрограммы проверит значение флага tp->tty_ioreq, и если имеется отложенная операция, вызовет dev_ioctl. Сама функция dev_ioctl проверяет tp->tty_outleft, чтобы узнать, завершен ли вывод. Если это так, выполняются те же действия, которые do_ioctl выполнила бы в варианте без задержки. При работе tcdrain единственное, что нужно сделать, — это сбросить поле tp->tty_ioreq и отправить ответное сообщение файловой системе с просьбой разбудить приостановленный процесс, сделавший вызов. Вариант TCSAFLUSH вызова tcsetaddr прибегает к tty_icancel, чтобы прервать ввод. В обоих вариантах tcseatddr структура termios, адрес которой передается при вызове ioctl, копируется в поле tp->tty_termios. Затем вызывается setattr и посылается ответное сообщение, как и в случае с tcdrain, с целью разблокировать процесс-инициатор вызова.

Следующая на очереди — процедура setattr. Как вы могли видеть, она вызывается из do_ioctl и dev_ioctl, чтобы изменить атрибуты терминального устройства, а также из do_close, когда нужно сбросить атрибуты в значения по умолчанию. Эта функция всегда вызывается после того, как записываются новые данные в структуру termios, поскольку лишь изменения значений недостаточно. Если управляемое устройство переведено в неканонический режим, необходимо установить бит IN_EOT у всех символов во входной очереди, как если бы они изначально были напечатаны в неканоническом режиме. Так как нельзя узнать, какой атрибут только что был изменен, невозможно и определить, следует ли устанавливать бит у символов в очереди. Поэтому проще все свести к одному случаю, нежели чем проверять содержимое очереди.

Далее нас интересуют значения параметров MIN и TIME. В каноническом режиме tp->tty_min всегда равно 1. Проверка выполняется в теле первого условного оператора после вызова unlock. В неканоническом режиме различные комбинации этих параметров позволяют реализовать четыре разных режима работы, показанных в табл. 3.10. В tp->tty_min сначала записывается значение, переданное через tp->tty_termiso.cc[VMIN], а после, при равенстве его нулю и при нулевом значении в tp->tty_termiso.cc[VTIME], оно изменяется.

Наконец, setattr обеспечивает, что вывод не останавливается, если отключено управление XON/XOFF, отправляет сигнал SIGHUP, если скорость передачи установлена в ноль, и делает косвенный вызов подпрограммы, адрес которой хранится в tp->tty_ioctl, чтобы выполнить те действия, которые могут быть выполнены только на уровне устройства.

Функция tty_reply уже много раз упоминалась в предшествующих обсуждениях. Ее алгоритм прост, она формирует сообщение и отправляет его. Если по какой-то причине отправить сообщение не удалось, происходит сбой системы. Оставшиеся функции столь же просты. Функция sigchar заставляет менеджер памяти отправить сигнал. Если установлен флаг NOFLSH, очищается входная очередь — обнуляется счетчик полученных строк или символов, а указатели на конец и начало очереди становятся равными. Это действие по умолчанию. Флаг может быть также установлен, когда ожидается сигнал SIGHUP, чтобы продолжить ввод и вывод и после получения сигнала. Функция tty_icancel очищает входную очередь, как это делает sigchar, а в дополнение вызывает аппаратно-специфичную функцию tp->tty_icancel, с целью очистить буфер самого устройства при наличии такого буфера.

Функция tty_init вызывается один раз на каждое устройство при запуске tty_task. Она устанавливает значения по умолчанию. Изначально в поля tp->tty_icancel, tp->tty_ocancel, tp->tty_ioctl и tp->tty_close записывается указатель на заглушку, tty_devnop. Затем tty_init вызывает одну из специфичных для устройства функций инициализации, в зависимости от того, к какой категории относится терминал: консоль, последовательная линия или псевдотерминал. Эта функция сохраняет в полях структуры ссылки на реальные функции, специфичные для данного устройства. (Вспомните, что если в определенной категории не инициализировано ни одного устройства, соответствующий макрос трубит немедленный отбой, в результате чего не компилируется ни одна из частей кода неиспользуемых уст-

ройств.) Затем вызов init_scr инициализирует драйвер консоли и обращается к процедуре инициализации клавиатуры.

Функция tty_wakeup невелика, но играет очень важную роль в работе задачи терминала. Когда стартует обработчик прерываний часов, значение глобальной переменной tty_timeout (файл glo.h) сравнивается с ее прошлым значением, скажем, на каждом такте таймера. Если новое значение меньше, вызывается функция tty_wakeup. Значение tty_timeout устанавливается в ноль обработчиками прерываний драйверов терминалов, поэтому после каждого прерывания от терминала обеспечивается вызов tty_wakeup. Кроме того, как мы увидим далее, значение tty_timeout изменяется подпрограммой settimer, когда терминал в неканоническом режиме выполняет вызов read и требуется задержка. Когда tty_wakeup получает управление, она прежде всего предотвращает дальнейшие вызовы tty_wakeup, для чего записывает в tty_timeout значение TIME_NEVER, которое соответствует времени далеко в будущем. Затем функция сканирует список значений таймеров до тех пор, пока не находит таймер, время срабатывания которого позже текущего (список отсортирован так, что в его голове находится таймер с самым ранним срабатыванием). Найденное значение определяет момент следующего пробуждения, и оно заносится в tty_timeout. Кроме того, tty_wakeup сбрасывает переменную tp->tty_min в 0, благодаря чему следующее чтение с этого устройства будет выполнено, даже если не будет получено ни одного байта. Также для этого устройства устанавливается флаг tp->tty_events, чтобы задача терминала обратила на него внимание при следующем запуске. Затем устройство удаляется из списка таймеров. Наконец, делается вызов функции interrupt, чтобы отправить задаче сообщение. Как упоминалось ранее при обсуждении задачи часов, функция tty_wakeup логически является частью кода, обслуживающего прерывания, так как вызывается только оттуда.

Следующая функция, settimer, устанавливает таймеры, определяющие возврат из вызова read в неканоническом режиме. Она принимает два аргумента, tp, указатель на структуру tty, и on, целое число, имеющее значение TRUE или FALSE. Сначала в списке структур tty, на голову которого указывает timelist, ищется запись, совпадающая с параметром tp. Если такая запись обнаруживается, она исключается из списка. Если settimer вызвана для того, чтобы сбросить таймер, дело сделано. Если вызов был направлен на установку таймера, в поле tp->tty_time структуры tty устройства фиксируется текущее время плюс добавка, равная значению TIME структуры termios (время измеряется в десятых долях секунды). Новая запись добавляется в список так, чтобы сохранить его упорядоченность. Наконец, только что указанный тайм-аут сравнивается со значением в глобальной переменной tty_timeout, и если новый истечет раньше, он записывается в переменную на место старого.

Последней функцией в файле tty.c является tty_devnop, пустая функция, которая косвенно вызывается тогда, когда устройство не требует специальных действий. Как можно было видеть, адрес этой функции по умолчанию используется для инициализации различных указателей в tty_init перед вызовом подпрограммы инициализации самого устройства.

3.9.5. Реализация драйвера клавиатуры

Теперь мы обратимся к аппаратно-зависимому коду, обеспечивающему работу консоли в MINIX, которая состоит из клавиатуры IBM PC и отображаемого в память экрана. Физические устройства консоли полностью различны, у стандартных настольных систем экран поддерживается при помощи контроллера (которая может быть одной из дюжины типов), а работу клавиатуры обеспечивают схемы на материнской плате, взаимодействующие с однокристальным 8-разрядным компьютером в клавиатуре. Для поддержания двух различных устройств требуется два различных набора программного обеспечения, которое в MINIX находится в файлах keyboard.c и console.c.

С точки зрения операционной системы, клавиатура и экран являются частями одного устройства, /dev/console. Если у видеоконтроллера достаточно памяти, то в систему может быть включена поддержка *виртуальных консолей*, и, помимо /dev/console, могут существовать дополнительные логические устройства, /dev/ttyc1, /dev/ttyc2 и т. д. В любой момент времени на экране видна только одна из этих консолей, эта же консоль получает ввод с единственной клавиатуры. Логически клавиатура подчинена консоли, хотя подтверждается это только двумя достаточно незначительными фактами. Во-первых, в структуре консоли tty, хранящейся в tty_table, есть отдельные поля для ввода и вывода, например tty_devread и tty_devwrite, указатели на функции в файлах keyboard.c и console.c, заполняемые при запуске. Тем не менее есть только одно поле tty_priv, ссылающееся исключительно на структуры данных консоли. Во-вторых, перед входом в главный цикл tty_task проводит инициализацию всех логических устройств. Для /dev/console процедура инициализации расположена в файле console.c, и код для инициализации клавиатуры вызывается из нее. С другой стороны, подразумеваемая иерархия может быть перевернута. Имея дело с устройствами ввода/вывода, мы всегда сначала рассматривали ввод, а затем вывод, и сейчас мы продолжим эту традицию, рассмотрев код keyboard.c в текущем разделе и отложив console.c до следующего.

Файл keyboard.c, как и многие другие, начинается с нескольких директив #include. Но одна из них необычна. Файл keymaps/us-std.src не является заголовочным файлом C, это файл исходных кодов, который при компиляции определяет раскладку клавиатуры по умолчанию, попадающую в keyboard.o в виде инициализированного массива. Сам этот файл довольно велик, и лишь несколько типичных записей из него приведены в табл. 3.13. После директив #include следуют макросы, задающие разнообразные константы. Первая группа привлекается для низкоуровневого взаимодействия с контроллером клавиатуры. Большая часть из них задают адреса портов ввода/вывода или битовые комбинации, используемые при таком взаимодействии. Следующая группа макросов описывает символические имена для специальных клавиш. Макрос kb_addr всегда возвращает указатель на первый элемент в массиве kb_lines, так как аппаратное обеспечение IBM поддерживает только одну клавиатуру. Константа KB_IN_BYTES, имеющая значение 32, определяет размер клавиатурного буфера. Следующие 11 переменных хранят различные параметры, необходимые для правильной

интерпретации нажатия клавиши. Они имеют разное назначение. Например, флаг capslock переключается с TRUE на FALSE, и наоборот при нажатии клавиши Caps Lock. Флаг shift устанавливается в TRUE, когда нажимается клавиша SHIFT, и сбрасывается, когда она отпускается. Переменная esc устанавливается, когда получен код опроса клавиатуры escape. Она всегда сбрасывается после того, как получен один символ.

Структура kb_s необходима для отслеживания введенных кодов опросов клавиатуры. Коды опроса в этой структуре хранятся в циклическом буфере, ibuf, имеющем размер KB_IN_BYTES. Объявляется массив из таких структур, kb_lines[NR_CONS], содержащий по одной структуре на консоль, но фактически используется только первый его элемент, так как текущая kb_s всегда определяется макросом kbaddr. Тем не менее мы обычно будем ссылаться на kb_lines[0] при помощи указателя на эту структуру, например kb->ihead, ради единства с рассмотрением других устройств. Конечно, неиспользуемые ячейки массива отнимают небольшое количество памяти, но, с другой стороны, если кто-либо выпустит компьютер, поддерживающий несколько клавиатур, ОС MINIX будет готова к такому нововведению, потребуется только изменить макрос kbaddr.

Макрос map_key0 возвращает ASCII-код, соответствующий данному коду опроса, без учета модификаторов, то есть попадающий в первую колонку карты клавиатуры. Старший брат этого макроса map_key выполняет преобразование кода опроса в ASCII-код с учетом всех модификаторов, действующих при вводе символа.

Когда нажимается или отпускается клавиша, вызывается обработчик прерываний клавиатуры, подпрограмма kbd_hw_int. Чтобы узнать от контроллера клавиатуры код опроса, она вызывает scan_keyboard. Когда прерывание инициировано отпусканием клавиши, у кода опроса устанавливается старший значащий бит, и в этом случае нажатие игнорируется, если это не одна из клавиш модификаторов. Если же прерывание обусловлено нажатием любой клавиши или отпусканием модификатора, необработанный код опроса помещается в круговой буфер, при наличии в нем свободного места. После этого устанавливается флаг tp->tty_events и вызывается force_timeout, чтобы гарантировать, что при следующем сигнале таймера задача часов запустит задачу терминала. В табл. 3.16 представлен пример содержимого кругового буфера для короткой строки, содержащей два символа в верхнем регистре, каждый из которых предваряется кодом опроса, соответствующим нажатию SHIFT, а следом за кодом символа идет код отпускания SHIFT.

Таблица 3.16. Содержимое входного буфера для строки текста, введенной с клавиатуры. Вторая строка таблицы описывает нажатия клавиш. L+, L-, R+ и R-означают соответственно нажатие и отпускание правой и левой клавиши SHIFT. Код отпускания клавиши на 128 больше кода нажатия

42	35	170	18	38	38	24	57	54	17	182	24	19	38	32	28
L+	h	L-	e	l	l	o	SP	R+	w	R-	o	r	l	d	CR

Когда возникает прерывание от часов, задача терминала получит управление и, обнаружив, что у консоли установлен флаг tp->tty_events, вызовет специфичную для данного устройства подпрограмму kb_read, на которую ссылается поле tp->tty_devread структуры tty консоли. Функция kb_read извлекает из циклического буфера коды опроса и помещает в свой локальный буфер ASCII-коды. Этот локальный буфер должен быть достаточно емок для того, чтобы вместить ESC-последовательности, генерируемые в ответ на нажатия некоторых клавиш на цифровой клавиатуре. Затем, чтобы поместить символы во входную очередь, вызывается in_process. Во избежание клавиатурного прерывания во время уменьшения значения переменной tp->icount, эта операция защищена вызовами lock и unlock. Вызов make_bread возвращает ASCII-код в виде целого числа. В этой точке специальные клавиши, например клавиши цифровой клавиатуры и функциональные, имеют коды большие, чем 0xFF. Коды в диапазоне от HOME до INSRT (от 0x101 до 0x10C, эти константы задаются в include/minix/keymap.h) при помощи массива numpad_map преобразуются в трехсимвольные ESC-последовательности, показанные в табл. 3.17. Эти последовательности затем передаются в in_process. Большие по величине коды не передаются в in_process, среди них ищутся коды, соответствующие комбинациям клавиш ALT+LEFT, ALT+RIGHT и от ALT+F1 до ALT+F12. Если обнаруживается одна из таких комбинаций, вызывается функция switch_console, переключающая виртуальные консоли.

Таблица 3.17. ESC-последовательности, генерируемые для клавиш цифровой клавиатуры. Когда коды опроса преобразуются в ASCII-коды, специальным клавишам присваиваются «псевдоASCII»-коды, большие 0xFF

Клавиша	Код опроса	«ASCII»	ESC-последовательность
Home	71	0x101	ESC [H
Up	72	0x103	ESC [A
Pg Up	73	0x107	ESC [V
-	74	0x10A	ESC [S
Left	75	0x105	ESC [D
5	76	0x109	ESC [G
Right	77	0x106	ESC [C
+	78	0x10B	ESC [T
End	79	0x102	ESC [Y
Down	80	0x104	ESC [B
Pg Dn	81	0x108	ESC [U
Ins	82	0x10C	ESC [@

Функция make_break преобразует коды опроса в ASCII и обновляет значение переменных, отслеживающих значения модификаторов. Но перед этим она ищет волшебную комбинацию клавиш CTRL+ALT+DEL, которую все пользователи PC знают как универсальный способ решения многих проблем (в MS-DOS). Тем не менее желательно, чтобы система корректно завершила свою работу, поэтому вместо передачи управления подпрограммам BIOS, процессу init — корню дере-

ва процессов — отправляется сигнал SIGABRT. Ожидается, что init обработает этот сигнал и в нормальном порядке завершит работу системы, прежде чем вернуться в монитор начальной загрузки, из которого можно управлять либо полным перезапуском системы, либо перезагрузкой MINIX. Конечно, было бы неправильно ожидать, что это сработает всегда. Большинство пользователей понимают опасность внезапной перезагрузки и не прибегают к CTRL+ALT+DEL до тех пор, пока не произойдет что-то действительно серьезное, после чего управлять системой будет невозможно. К этому моменту может сложиться ситуация, когда правильно отправить сигнал другому процессу уже невозможно. Именно потому в make_break есть статическая переменная CAD_count. При большинстве сбоев система обработки прерываний продолжает работать, значит, клавиатурный ввод продолжает поступать, и задача часов остается работоспособной. MINIX учитывает поведение пользователей, которые, когда система не реагирует на нажатия, начинают в сердцах лупцевать по клавишам. Если попытка отправить SIGABRT процессу init провалилась и пользователь обратился к магической комбинации CTRL+ALT+DEL во второй раз, делается прямой вызов wreboot, которая принудительно возвращает управление монитору начальной загрузки.

Основная часть make_break устроена не сложно. В переменную make записывается признак, было ли прерывание вызвано нажатием или отпусканием клавиши, после чего в переменную ch помещается ASCII-код, возвращаемый функцией map_key. Далее следует оператор switch, проверяющий значение ch. Рассмотрим два случая: случай обычной клавиши и случай специальной клавиши. Если нажата обычная клавиша, ни одно из условий в switch не будет выполнено, в варианте по умолчанию также ничего не произойдет, поскольку обычные символы принимаются только при нажатии клавиши. Если каким-то образом обычная клавиша была воспринята при отпускании, ее код заменяется значением −1, которое игнорируется вызывающей функцией. Обработка клавиш ALT, CALOCK, NLOCK и SLOCK сложнее, но во всех этих вариантах действия похожи: в переменную записывается либо новое состояние модификатора (если модификатор действует только тогда, когда удерживается клавиша), либо инвертированное старое состояние (для клавиш наподобие Caps Lock).

Нужно рассмотреть еще один вариант, код EXTKEY и переменную esc. Не путайте этот случай с клавишей ESC на клавиатуре, которой соответствует код опроса 0x1B. Код EXTKEY нельзя сгенерировать по отдельности, нажав какую-либо клавишу или их комбинацию. Это *префикс расширенных клавиш* для клавиатур PC, первый байт двухбайтового кода опроса, означающего, что передаваемый далее код опроса не является частью обычного набора клавиш PC. Во многих случаях программное обеспечение интерпретирует обычный и расширенный коды одинаково. Например, это почти всегда так для обычной клавиши «/» и серой клавиши «/» на цифровой клавиатуре. В других случаях может потребоваться различать их нажатия. Так, во многих раскладках клавиатур для языков, отличных от английского, левая и правая клавиши ALT интерпретируются по-разному, позволяя вводить три разных символа с одной клавиши. Код опроса у обеих клавиш одинаков и равен 56, но при нажатии правой ALT код опроса предваряется кодом EXTKEY. Когда получен код EXTKEY, устанавливается флаг esc, и в этом слу-

чае make_break делает возврат прямо из оператора switch, тем самым обходя операторы в конце функции, записывающие в переменную esc нулевое значение. В результате esc действует только на один символ, который будет получен следующим. Если вы знакомы с особенностями обычного программирования ввода от клавиатур PC, то это также будет вам понятно, хотя и немного странно, так как BIOS PC не позволяет считывать код опроса для клавиши ALT и возвращает другое значение для расширенного кода.

Функция set_leds управляет светодиодами на клавиатуре, индицирующими состояние клавиш Num Lock, Caps Lock и Scroll Lock. Чтобы указать клавиатуре, что следующий записанный в порт байт управляет индикаторами, в порт записывается управляющий байт, LED_CODE. Состояние всех трех светодиодов кодируется тремя битами этого байта. Две следующие функции необходимы для поддержки данной операции. Функция kb_wait вызывается тогда, когда необходимо определить момент готовности клавиатуры к получению управляющей последовательности, а kb_ack проверяет, что команда была подтверждена. Обе эти команды используют активное ожидание, непрерывно считывая состояние до тех пор, пока не будет получено нужное значение. Такая методика не рекомендуется для операций ввода/вывода, но переключение индикаторов не должно происходить часто, поэтому временные издержки не так велики. Заметьте, что и kb_wait, и kb_ack могут завершиться неудачей, что видно из кода возврата функций. Правда, переключение индикаторов на клавиатуре — не самая важная задача, поэтому возвращаемое значение не проверяется, и set_leds выполняется «вслепую».

Так как клавиатура является составной частью консоли, ее подпрограмма инициализации, kb_init, вызывается из scr_init в файле console.c, а не напрямую из tty_init в tty.c. Если включены виртуальные консоли (то есть константа NR_CONS в include/minix/config.h больше 1), kb_init вызывается для каждой логической консоли. Хотя после того, как kb_init будет запущена для первой консоли, а для других необходимо только записать адрес процедуры kb_read в поле tp->tty_devread, нет никакого вреда в том, чтобы лишний раз повторить и остальные команды инициализации. В оставшейся части kb_init устанавливаются значения некоторых переменных, задается состояние индикаторов на клавиатуре, а клавиатура сканируется, чтобы удостовериться, что не будет прочитано никаких остаточных нажатий на клавиши. Когда все готово, функция инициализации вызывает put_irq_handler, а затем enable_irq, чтобы при следующем нажатии или отпускании клавиши сработал обработчик kbd_hw_int.

Следующие три функции довольно просты. Подпрограмма kbd_loadmap почти тривиальна. Она вызывается из do_ioctl с целью скопировать карту клавиатуры из пользовательского адресного пространства. При этом новая раскладка записывается поверх раскладки по умолчанию, сгенерированной благодаря тому, что исходный файл раскладки включен в начало keyboard.c.

Функция func_key вызывается из kb_read и проверяет, не нажата ли специальная клавиша, которая должна быть обработана локально. Эти клавиши и производимый ими эффект перечислены в табл. 3.18. Вызываемые функцией подпрограммы расположены в различных файлах. Коды F1 и F2 активизируют функции из файла dmp.c, который мы рассмотрим в следующем разделе. Код F3 приводит

к вызову функции toggle_scroll из состава console.c, которая также будет рассмотрена в следующем разделе. Коды CF7, CF8 и CF9 обслуживаются вызовами функции sigchar из tty.c. Когда MINIX скомпилирована с поддержкой сети, обрабатывается еще один код, F5, распечатывающий статистику Ethernet. Кроме этого, доступно еще большое число различных кодов, которые можно использовать для передачи на консоль различных отладочных сообщений и прочих специальных событий.

Функция scan_keyboard взаимодействует с аппаратным интерфейсом клавиатуры, считывая и записывая байты из портов ввода/вывода. Контроллер клавиатуры информируется о том, что символ был считан последовательностью трех команд, начинающейся со второго вызова in_byte. Эти команды считывают байт, затем записывают его обратно с установленным старшим битом, а затем перезаписывают его еще раз, со сброшенным битом. В результате те же самые данные не встретятся заново при следующем чтении. В этой функции не делается никаких проверок, что не есть проблема, так как scan_keyboard вызывается только из обработчика прерываний (из kb_init, как единственное исключение, чтобы удалить любой мусор).

Таблица 3.18. Функциональные клавиши, обнаруживаемые func_key()

Клавиша	Действие
F1	Отображается таблица процессов
F2	Показывает информацию о занятой процессом памяти
F3	Переключает между программной и аппаратной прокруткой
F5	Показывает статистику Ethernet (если есть поддержка сети)
CF7	Отправляет сигнал SIGQUIT, тот же эффект, что и CTRL+\
CF8	Отправляет сигнал SIGINT, тот же эффект, что и DEL
CF9	Отправляет сигнал SIGKILL, тот же эффект, что и CTRL+U

Завершает файл keyboard.c функция wreboot. Вызываемая в результате краха системы, она дает пользователю возможность просмотреть отладочную информацию при помощи функциональных клавиш. Цикл в этой ветви кода функции — еще один пример активного ожидания. Клавиатура непрерывно опрашивается до тех пор, пока не будет нажата ESC. Безусловно, никто не может заявить, что после сбоя, когда ожидается команда на перезагрузку, имеет смысл применять более эффективные методики. Внутри цикла вызывается func_keys, это позволяет обрабатывать функциональные клавиши с целью сбора информации, которая поможет понять причины сбоя. Дальнейшие детали возврата в монитор начальной загрузки мы обсуждать не будем, так как они сильно завязаны на аппаратное обеспечение и не имеют особого отношения к операционным системам.

3.9.6. Реализация драйвера экрана

Если имеется достаточное количество видеопамяти, экран IBM PC можно сконфигурировать как несколько виртуальных терминалов. В этом разделе мы будем

обсуждать аппаратно-зависимый код консоли. Кроме того, мы опишем подпрограммы вывода отладочной информации, взаимодействующие с клавиатурой и дисплеем на низком уровне. Эти средства предоставляют ограниченные возможности для взаимодействия с пользователем, но работают даже тогда, когда другие части системы не функционируют, и могут дать полезную информацию даже после практически тотального краха системы.

Специфичный для отображаемого в память экрана PC код расположен в файле console.c. Здесь объявляется структура console, которая по своему смыслу является расширением структуры tty из файла tty.c. При инициализации в поле tp->tty_priv структуры tty для консоли записывается указатель на принадлежащую этой консоли структуру console. Первое поле структуры является указателем, содержащим обратную ссылку на структуру tty. Остальные компоненты структуры содержат вполне ожидаемую для видеоустройства информацию: переменные, хранящие текущие координаты курсора, адрес начала области памяти, отведенной для дисплея, и ее размер, адрес, на который ссылается регистр базы чипа контроллера, и текущий адрес курсора. Другие переменные используются для работы с ESC-последовательностями. Так как символы принимаются в виде 8-битных байтов и перед выводом в видеопамять должны быть скомбинированы с байтом атрибутов, блок данных подготавливается к передаче в c_ramqueue, массиве, объема которого достаточно для хранения 80 16-битных пар «символ-атрибут». Каждой виртуальной консоли требуется собственная структура console, для хранения которой выделяется область памяти в cons_table. Как мы уже делали в случае структур tty и kb_s, мы обычно будем ссылаться на поля console через указатель, например: cons->c_tty.

Для каждой консоли в поле tp->tty_devwrite хранится адрес функции cons_write. Она вызывается только из одного места, обработчика handle_events в файле tty.c. Большая часть остальных функций в console.c существуют для того, чтобы обеспечивать работу cons_write. Когда она вызывается впервые после того, как процесс-клиент сделал вызов write, выводимые данные расположены в буфере в адресном пространстве этого процесса. Определить положение этого буфера можно при помощи полей tp->tty_outproc и tp->out_vir структуры tty. Поле tp->tty_outleft говорит о том, сколько символов должно быть передано, а поле tp->tty_outcum изначально равно нулю, сообщая о том, что ни одного символа еще не выведено. Это обычная ситуация для cons_write, так как эта функция обычно выводит все данные, указанные в вызове. Но если пользователь хочет замедлить процесс вывода, он может ввести с клавиатуры символ STOP (CTRL+S), который приводит к тому, что устанавливается флаг tp->tty_inhibited. Когда этот флаг установлен, cons_write немедленно возвращает управление, даже если вызов write еще не выполнен полностью. Причем handle_events продолжит вызывать cons_write, и, когда флаг tp->tty_inhibited будет сброшен (для этого необходимо ввести символ START (CTRL+Q), прерванная передача данных будет возобновлена.

Единственным аргументом cons_write является указатель на структуры tty для данной консоли, поэтому первым действием инициализируется указатель cons, содержащий адрес структуры console. Затем необходимо проверить, действительно ли есть какая-либо работа, так как handle_events вызывает cons_write неоднократно.

Если нет, функция быстро завершается. После этого она входит в свой главный цикл. Этот цикл очень похож на цикл в функции in_transfer в файле tty.c. При помощи вызовов phys_copy локальный буфер объемом 64 символа заполняется данными из пользовательского буфера, обновляются указатель на начало буфера и счетчик символов, после чего все символы из локального буфера переносятся в массив cons->c_ramqueue, дополненные байтом атрибутов. Позже эти данные будут выведены на экран посредством flush. Как вы могли видеть на рис. 3.26, существует несколько способов осуществить эту передачу. Можно вызывать out_char для каждого символа, но лишь в расчете на ситуацию, когда при выводе символов не потребуется ни одна из специальных функций out_char, не выводится ESC-последовательность, ширина экрана не будет превышена и cons->c_ramqueue не заполнена. Если все функции out_char не требуются, символ можно поместить в cons->c_ramqueue напрямую, вместе с байтом атрибутов (поле cons->c_attr), а все переменные cons->c_rwords (индекс в очереди), cons->c_column (отслеживает текущий столбец на экране) и tbuf (указатель на буфер) инкрементируются. Прямой перенос символов в cons->c_ramqueue соответствует штриховой линии с левой стороны рис. 3.26. При необходимости вызывается out_char. Этот вызов занимается всеми подсчетами и по мере надобности вызывает функцию flush, которая выполняет окончательную передачу данных в видеопамять. Перекачка данных из пользовательского буфера в локальный производится до тех пор, пока значение в поле tp->tty_outleft говорит о том, что еще есть символы и не установлен флаг tp->tty_inhibited. Если передача останавливается, в результате завершения вызова write или потому, что установлен флаг tp->tty_inhibited, то чтобы передать последние символы из очереди в память экрана, опять вызывается flush. Когда операция завершена, при помощи tty_reply отправляется ответное сообщение (операция завершена, когда поле tp->tty_outleft содержит нулевое значение).

В дополнение к вызовам cons_write из handle_events, символы на консоль могут выводить функции echo и rawecho в аппаратно-независимой части задачи терминала. Если текущим терминалом является консоль, косвенные вызовы через указатель tp->tty_echo перенаправляются к функции cons_echo. Она делает свою работу, вызывая out_char и затем flush. Пользователь вводит данные символ за символом, и ему предпочтительно, чтобы эхо отображалось сразу же, без видимой задержки, поэтому помещать символы в очередь было бы недостаточно.

Итак, теперь мы добрались до функции out_char. Сначала она проверяет, вводится ли сейчас ESC-последовательность, вызывая parse_escape, и если это так, немедленно завершается. В противном случае управление передается в конструкцию switch, которая проверяет различные особые случаи: нулевой символ, символ забоя, символ звонка и т. п. Несложно проследить, как обрабатывается большая часть этих ситуаций. Самые сложные варианты — символы перевода строки и табуляции, поскольку они сложным образом меняют координаты курсора на экране и могут вызвать прокрутку. Последняя проверка выполняется на код ESC. Если он обнаруживается, устанавливается флаг cons->c_esc_state и последующие вызовы out_char будут перенаправляться в parse_escape до тех пор, пока последовательность не завершится. Если превышена ширина экрана, при необходимости делается прокрутка экрана и вызывается flush. Прежде чем поместить символ

в выходную очередь, проверяется, заполнена ли она, и если заполнена, вызывается flush. Как мы видели ранее в cons_write, при занесении символа в очередь необходимо учесть это, обновив значения нескольких переменных.

Далее следует функция scroll_screen. Она выполняет как прокрутку вверх, то есть нормальную прокрутку, ожидаемую при заполнении нижней строки экрана, и прокрутку вниз, которая необходима при попытке установить курсор выше верхней границы экрана. Для каждого направления прокрутки возможны три метода. Это проистекает из требования поддержки различных типов видеокарт.

Мы рассмотрим случай прокрутки вверх. Сначала переменной chars присваивается значение, равное размеру экрана минус 1. Когда прокрутка делается программно, для ее выполнения достаточно одного вызова функции vid_vid_copy, которая скопирует chars символов на одну строку ниже в памяти. Эта функция умеет переходить в начало области при достижении ее конца, и наоборот. Так, если ей указано скопировать блок памяти, начало которого выходит за верхнюю границу памяти, то не укладывающиеся в область видеопамяти данные будут взяты из ее нижней части, то есть видеопамять рассматривается как круговой (замкнутый) массив. Простота этого вызова не скрывает такой недостаток, как низкая скорость выполнения операции. Даже несмотря на то, что подпрограмма vid_vid_copy написана на языке ассемблера (ее код хранится в файле klib386.s), для ее выполнения необходимо скопировать 3840 байт, что довольно большая работа даже для тщательно «вылизанного» кода.

Программная прокрутка никогда не выбирается по умолчанию. Пользователь может включить ее, если аппаратная не работает или по каким-то причинам нежелательна. Одна из таких причин — желание использовать команду screendump, чтобы иметь возможность сохранить экранную память в файл. При использовании аппаратной прокрутки эта команда не дает ожидаемого результата, так как начало видеопамяти наверняка не совпадает с началом видимой на экране области. Если выбрана не программная прокрутка, то в первой части составного условия проверяется значение переменной wrap. Эта переменная содержит TRUE для старых экранов, поддерживающих аппаратную прокрутку, и, если проверка не выполняется, в ветви else производится простая аппаратная прокрутка. Соответственно, значение указателя на начало экранной области, используемого видеоконтроллером, cons->c_orig, изменяется так, чтобы указывать на первый символ той строки, которая окажется наверху экрана. Если wrap равна FALSE, проверка составного условия продолжается. Теперь проверяется, поместится ли перемещаемый блок памяти в той области памяти, которая отведена для консоли. Если нет, то при помощи vid_vid_copy содержимое копируется физически, в начало области видеопамяти. Если же адреса не перекрываются, делается простая аппаратная прокрутка, всегда практикуемая в более старых видеоконтроллерах. Для этого изменяется значение cons->c_org, и новое значение указателя на начало области заносится в нужный управляющий регистр контроллера. Соответствующий вызов делается позднее, как и вызов, очищающий нижнюю строку экрана.

Код прокрутки вниз очень похож на тот, что прокручивает экран вверх. В конце нижняя строка экрана очищается при помощи вызова mem_vid_copy, обновляются значения некоторых переменных и делается проверка того, что координаты

курсора имеют приемлемые значения. При необходимости, если, например, ESC-последовательность переместила курсор на столбец с отрицательным номером, координаты корректируются. В завершение вычисляется, где должен быть курсор, и это значение сравнивается с cons->c_cur. Если значения не совпадают, а обрабатываемая память принадлежит текущей виртуальной консоли, то, чтобы записать корректные значения в регистр контроллера, делается вызов подпрограммы set_6845.

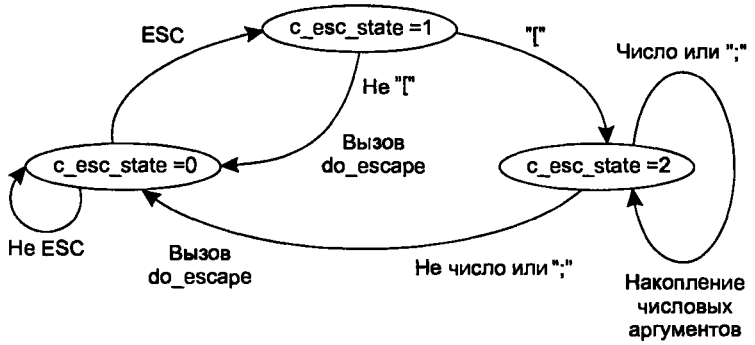


Рис. 3.28. Конечный автомат для обработки ESC-последовательностей

На рис. 3.28 показано, как можно представить разбор ESC-последовательностей при помощи конечного автомата. Этот автомат реализуется подпрограммой parse_escape, вызываемой в начале кода out_char, если поле cons->c_esc_state не равно нулю. Сам символ ESC обнаруживается в out_char, и переменная cons->c_esc_state переводится в состояние 1. Когда получен следующий символ, функция parse_escape подготавливается к обработке дальнейшей информации: значение '\0' заносится в поле cons->c_esc_intro, указатель на начало массива параметров, cons->c_esc_paramv[0], заносится в cons->c_esc_paramp, а сам массив параметров заполняется нулями. Затем проверяется первый символ, следующий за ESC. Допустимыми значениями являются «[» и «M». В первом случае «[» копируется в переменную cons->c_esc_intro, и автомат переходит в состояние 2. Во втором случае вызывается функция do_escape, которая выполняет действие, и автомат возвращается в состояние 0. Если же за ESC последовал недопустимый символ, он игнорируется, а дальше все обрабатывается как обычно.

Когда автомат получает последовательность ESC [, следующий полученный символ обрабатывается в состоянии 2. В этой точке возможны три варианта. Если на входе числовой символ, его значение добавляется к увеличенному в десять раз значению параметра, на который в текущий момент указывает cons->c_esc_paramp (сначала этот указатель ссылается на cons->c_esc_paramv[0], и все параметры равны нулю). Здесь состояние автомата пока не меняется. Это позволяет передавать параметры в виде последовательности чисел, накапливая их итог, хотя максимальное значение, в текущий момент распознаваемое MINIX, равно 80. Оно может быть использовано в последовательности, перемещающей курсор в указанную позицию на экране. Если получено не число, а точка с запятой, тогда

указатель на текущий параметр перемещается к следующему параметру, чтобы начать считывание его значения. Благодаря такому подходу, если изменить константу MAX_ESC_PARAMS в сторону массива большего объема, код менять не придется. Наконец, в третьем случае, когда получен символ, не являющийся ни числом, ни точкой с запятой, вызывается do_escape.

Функция do_escape весьма объемна, несмотря на относительно скромную поддержку ESC-последовательностей в MINIX. Но какая бы длина ни была, код должен быть достаточно прост. После того как делается вызов flush, нужно убедиться, что содержимое экрана полностью обновлено. Оператор if выполняет простую проверку, является ли следующий за ESC символ преамбулой ESC-последовательности или нет. Если нет, возможно только одно действие, перемещение курсора на строку вверх, этому соответствует ESC-последовательность ESC M. Обратите внимание, что проверка значения осуществляется оператором switch, это сделано в предчувствии появления новых вариантов, то есть последовательностей, не соответствующих формату ESC [. Потому обрабатывается вариант без преамбулы типичным для многих последовательностей образом: по значению переменной cons->c_row определяется, необходима ли прокрутка. Если курсор находится в нулевой строке, делается вызов scroll_screen с параметром SCROLL_DOWN. Если нет, курсор просто сдвигается на одну строку вверх, для чего переменная cons->c_row уменьшается на 1 и вызывается flush. Если обнаружена преамбула ESC-последовательности, срабатывает другая ветвь верхнего оператора if. Сначала проверяется, не «[» ли это, то есть единственная возможная преамбула, обрабатываемая в текущий момент в MINIX. Если символ корректен, в переменную value записывается значение первого полученного параметра, или ноль, если параметров нет. Если последовательность некорректна, ничего не происходит, за исключением того, что switch с большим телом пропускается, а состояние автомата сбрасывается в 0 перед вызовом do_escape. В более интересном случае, когда последовательность правильна, выполняется оператор switch. Все возможные варианты мы рассматривать не будем. Вместо этого мы обсудим только наиболее представительные типы действий, управляемых ESC-последовательностями.

Первые пять последовательностей без числовых аргументов генерируются клавишами-стрелками и клавишей Home на клавиатурах IBM PC. Две последовательности, ESC [A и ESC [B, сходны с ESC M, с той разницей, что числовой параметр позволяет перемещаться более чем на одну строку, а при достижении границ экрана содержимое не прокручивается. Функция flush в этом случае обнаруживает попытки передвинуть курсор за границы экрана и ограничивает его перемещение. Две другие последовательности, ESC [C и ESC [D, перемещающие курсор вправо и влево, аналогичным образом ограничены flush. Когда они генерируются клавишами управления курсором, числовой аргумент не передается, поэтому происходит перемещение на одну строку или столбец.

Далее, последовательность ESC [H может иметь два числовых параметра, например ESC [20;60H. Эти параметры задают положение курсора в абсолютных координатах, а не относительно предыдущего места расположения, и для правильной интерпретации преобразуются из координат, отсчитываемых с 1, в координаты

с началом в 0. Клавиша Home на клавиатуре генерирует последовательность без параметров (с параметрами по умолчанию), которая перемещает курсор в положение (1;1).

Две следующие последовательности, ESC [sJ и ESC [sK, очищают либо часть строки, либо часть всего экрана, в зависимости от переданного параметра. В обоих случаях подсчитывается количество символов. Например, для последовательности ESC [1J в count заносится количество символов с начала экрана до текущего положения курсора, и это количество и параметр положения, dst, который может быть равен началу экрана, cons->c_org, используются как аргументы для вызова mem_vid_copy. Аргументы процедуры таковы, что она заполняет указанную область экрана текущим цветом фона.

Четыре следующие последовательности вставляют новые строки и удаляют строки и пробелы в текущем местоположении курсора, и их работа не нуждается в детальном рассмотрении. Последний вариант, последовательность ESC [*n*m (обратите внимание, что *n* — числовой аргумент, а «m» — литера, часть последовательности), оказывает влияние на параметр cons->c_attrib. Это байт атрибутов, который при записи символов в видеопамять чередуется с кодами символов.

Функция set_6845 вызывается тогда, когда необходимо обновить информацию чипа видеоконтроллера. У контроллера 6845 есть внутренние 16-разрядные регистры, которые программируются по 8 бит за раз. Поэтому для записи одного регистра требуются четыре операции с портами ввода/вывода. Так как прерывания могут нарушить последовательность действий, делаются вызовы lock и unlock, запрещающие прерывания на время обмена информацией. Некоторые из регистров чипа контроллера 6845 перечислены в табл. 3.19.

Таблица 3.19. Некоторые из регистров чипа 6845

Регистры	Назначение
10–11	Размер курсора
12–13	Начальный адрес видимой части экрана
14–15	Положение курсора

Функция beep вызывается при выводе символа CTRL+G. Она подает на внутренний динамик прямоугольный сигнал, опираясь на встроенные аппаратные возможности PC. Звук появляется путем некоторых магических манипуляций с портами ввода/вывода, которые интересны только программистам на ассемблере. Более интересно то, как при помощи задачи часов устанавливается сигнальный таймер, который используется для инициации функции. Адрес следующей подпрограммы, stop_beep, передается в сообщении задаче часов. Она прекращает звуковой сигнал после того, как истечет заданное время, и сбрасывает флаг beeping, используемый для того, чтобы избыточные вызовы звукового сигнала не превращались в сам звук.

Подпрограмма scr_init вызывается из tty_init столько раз, сколько указано в NR_CONS. При каждом вызове аргументом подпрограммы является указатель на структуру, один из элементов массива tty_table. В подпрограмме вычисляется значение line, будущий индекс в массиве cons_table, это значение проверяется на

корректность, и, если все правильно, оно используется для инициализации указателя cons, ссылающегося на текущую запись в массиве cons_table. К этому моменту поле cons->c_tty может быть инициализировано указателем на главную структуру tty для устройства, а в tp->tty_priv, в свою очередь, может быть записан указатель на структуру cons_t устройства. Затем для инициализации клавиатуры вызывается подпрограмма kb_init и устанавливаются указатели на специфичные для данного устройства подпрограммы. После этого tp->tty_devwrite ссылается на cons_write, а tp->tty_echo содержит указатель на cons_echo. Далее определяется адрес ввода/вывода регистра базы видеоконтроллера и, в соответствии с типом видеоконтроллера, устанавливается флаг wrap (этот флаг определяет способ прокрутки). Затем в глобальной таблице дескрипторов запоминается дескриптор области видеопамяти.

В дальнейшем происходит инициализация виртуальных консолей. При инициализации каждой консоли с разным значением tp вызывается scr_init, и, таким образом, для каждой консоли в scr_init используются собственные значения line и cons, и каждая консоль «арендует» собственный участок видеопамяти. Затем каждый экран очищается, и, наконец, консоль с нулевым номером становится активной.

Оставшиеся подпрограммы из файла console.c устроены просто, имеют небольшой размер и заслуживают столько же слов. Функция putk уже упоминалась ранее. Она печатает на экран символ от лица подпрограмм ядра, не прося о помощи файловую систему. Функция toggle_scroll делает именно то, что означает ее название, она изменяет значение флага типа прокрутки: аппаратная или программная. Помимо этого, она выводит текст в текущей позиции курсора, сообщая, какой режим выбран. Функция cons_stop инициализирует консоль, переводя ее в состояние, которое ожидает монитор начальной загрузки. Это делается перед выходом из системы или перезагрузкой. Функция cons_org0 используется только тогда, когда по клавише F3 изменяется режим прокрутки или когда идет подготовка к выходу из системы. Функция select_console выбирает (активизирует) виртуальную консоль. При вызове ей передается индекс новой консоли, и она дважды вызывает set_6845, чтобы показать на экране данные из соответствующей части видеопамяти.

Две последние подпрограммы сильно зависят от особенностей программного обеспечения. Функция con_loadfont загружает в видеоконтроллер шрифт, обеспечивая выполнение операции TIOCSFON вызова ioctl. Эта функция при помощи серии вызовов ga_program делает так, что становится видимой память шрифтов контроллера, которая в обычной ситуации не адресуема. Затем, чтобы скопировать шрифт в ставшую доступной область памяти, вызывается phys_copy, а после этого другая последовательность команд возвращает устройство в нормальный режим работы.

Отладочный вывод

И еще одну группу процедур, которую мы упомянем в рассмотрении задачи терминала, сначала предполагалось использовать только временно, при отладке MINIX. После того как отладка завершена, их можно удалить, но многие пользователи

находят эти функции полезными и оставляют их. Особенно полезны эти подпрограммы при модификации системы.

Как вы видели ранее, в начале кода kb_read вызывается func_key, которая определяет различные коды, используемые для управления и отладки. Процедуры вывода отладочной информации из файла dmp.c вызываются, когда обнаруживаются коды клавиш F1 и F2. Первая из них, p_dmp, показывает основную информацию обо всех процессах, включая сведения о занятой памяти. Эта подпрограмма вызывается при нажатии F1. Вторая подпрограмма, map_dump, при нажатии F2 представляет более подробную информацию о занятой памяти. Функция proc_name обеспечивает работу p_dmp, определяя имя процесса.

Так как весь код отладки содержится в коде ядра и не выполняется ни как пользовательский процесс, ни как задача, он зачастую работает даже тогда, когда произошел серьезный сбой системы. Конечно, эти подпрограммы доступны только из консоли. Информация, ими выводимая, не может быть перенаправлена в файл или на какое-либо другое устройство, поэтому нет вариантов с получением твердой копии или работой через сеть.

Мы предполагаем, что первым шагом при дальнейшем развитии MINIX может стать расширение процедур отладочного вывода так, чтобы они выводили более подробные сведения о том аспекте системы, который совершенствуется.

3.10. Задача системы в MINIX

Вследствие того, что файловая система и менеджер памяти работают вне ядра системы, иногда возникает потребность передать ядру какую-то информацию. Но такая структура запрещает серверам напрямую писать информацию в таблицу ядра. Например, менеджером памяти обслуживается системный вызов fork. Когда создается новый процесс, ядро должно об этом узнать, чтобы обеспечить его планирование. Но как менеджер памяти скажет об этом ядру?

Решением проблемы является системная задача, которая взаимодействует с менеджером памяти и файловой системой при помощи стандартного механизма сообщений и, помимо этого, имеет доступ к структурам памяти ядра. Эта задача, называемая *задачей системы*, на рис. 2.14 образует второй уровень, работая, как и другие рассмотренные нами задачи. Ее отличие в том, что она не связана с каким-либо устройством. Как и прочие задачи, она реализует интерфейс, но в данном случае это не интерфейс со внешним миром, а связь с самой законспирированной частью системы. Она имеет те же самые привилегии, что и задачи ввода/вывода, и вместе с ними входит в образ ядра, поэтому имеет смысл рассмотреть ее именно здесь, а не в другой главе.

Системная задача воспринимает 19 типов сообщений, перечисленных в табл. 3.20. Основная функция задачи системы, sys_task, имеет ту же структуру, что и у других задач. Эта функция получает сообщение, вызывает соответствующую обслуживающую функцию и отправляет ответ. Итак, рассмотрим все эти сообщения и все обслуживающие процедуры.

Сообщение SYS_FORK отправляет менеджер памяти, уведомляя им ядро о появлении нового процесса. Ядру необходимо знать об этом, чтобы запланировать новый процесс. В сообщении передаются номера ячеек таблицы процессов, занимаемых родительским и дочерним процессами. Собственные таблицы процессов есть и у менеджера памяти, и у файловой системы, и один индекс *k* соответствует одному процессу во всех трех таблицах. Таким образом, менеджер памяти может сообщить только номера ячеек родительского и дочернего процесса, а ядро поймет, какие процессы имеются в виду.

Подпрограмма do_fork сначала проверяет, не передал ли менеджер памяти ядру неправильную информацию. Для проверки используется макрос isoksusern из файла proc.h, который проверяет, что ячейки в таблице процессов содержат верные данные. Подобные проверки выполняются и другими обслуживающими подпрограммами из system.c. Это чистой воды паранойя, но небольшие дополнительные проверки вреда не принесут. Завершив тестирование, do_fork копирует запись из ячейки родительского процесса в ячейку дочернего. При этом необходимы некоторые дополнительные действия. Дочерний процесс освобождается от любых текущих сигналов, и потомок не наследует состояние трассировки родителя. Кроме того, конечно же, вся статистическая информация у дочернего процесса сбрасывается в 0.

Таблица 3.20. Типы сообщений, понимаемые системной задачей (FS – файловая система, MM – менеджер памяти)

Тип сообщения	Отправитель	Назначение
SYS_FORK	MM	Процесс разветвился
SYS_NEWMAP	MM	Устанавливает для процесса новую карту памяти
SYS_GETMAP	MM	Менеджеру памяти необходима карта памяти процесса
SYS_EXEC	MM	Получает указатель стека после вызова exec
SYS_XIT	MM	Процесс завершился
SYS_GETSP	MM	Менеджеру памяти необходим указатель стека процесса
SYS_TIMES	FS	Файловой системе нужна информация о процессорном времени процесса
SYS_ABORT	Оба	Сбой: MINIX не может продолжить работу
SYS_SENDSIG	MM	Посылает процессу сигнал
SYS_SIGRETURN	MM	Восстановление после того, как сигнал обработан
SYS_KILL	FS	Отправляет процессу сигнал после вызова kill
SYS_ENDSIG	MM	Восстановление после сигнала из ядра
SYS_COPY	Оба	Копирование данных между процессами
SYS_VCOPY	Оба	Копирование нескольких блоков данных между процессами
SYS_GBOOT	FS	Определяет параметры загрузки
SYS_MEM	MM	Менеджер памяти требует следующий свободный блок памяти
SYS_UMAP	FS	Преобразует виртуальный адрес в физический
SYS_TRACE	MM	Выполняет действия системного вызова ptrace

После вызова fork менеджер памяти выделяет для нового процесса память. Ядро должно знать, какую память занимает дочерний процесс, чтобы правильно установить у него сегментный регистр. Дать ядру информацию о карте памяти любого процесса позволяет сообщение SYS_NEWMAP. Оно может быть использовано и тогда, когда системный вызов brk изменил карту памяти.

Этот тип сообщений обслуживается функцией do_newmap, которая сначала копирует новую карту памяти в адресное пространство менеджера памяти. Сама карта в сообщении не содержится, ввиду своего размера. Теоретически, менеджер памяти может сказать, что новая карта памяти расположена по адресу m, а m окажется неправильным адресом. Менеджер памяти не должен так поступать, но ядро не руководствуется только доверием. После проверки карта напрямик копируется в поле p_map записи целевого процесса в таблице процессов. При помощи вызова alloc_segments информация извлекается из карты и помещается в поля p_reg, содержащие значения сегментных регистров. Эти действия не сложны, но сильно зависят от модели процессора, поэтому вынесены в отдельную функцию.

Сообщение SYS_NEWMAP часто используется при нормальной работе системы MINIX. Сходное сообщение, SYS_GETMAP, посылается только при первоначальном запуске файловой системы и запрашивает передачу информации о карте памяти в обратном направлении, от ядра к менеджеру памяти. Обслуживается оно функцией do_getmap. Коды этой функции и функции обработки предыдущего сообщения сходны, основное различие в том, что отправитель и получатель в вызове phys_copy поменяны местами.

Когда процесс делает системный вызов exec, менеджер памяти устанавливает для него новый стек, содержащий аргументы и переменные окружения. Результирующий указатель стека передается ядру при помощи сообщения SYS_EXEC, обслуживаемого функцией do_exec. Сначала в этой функции делается обычная проверка корректности информации о процессе, а затем проверяется поле PROC2 сообщения. Это поле используется как признак, трассируется ли процесс, и ничего общего с идентификацией процесса не имеет. Если трассировка в действии, то, чтобы отправить процессу сигнал SIGTRAP, делается вызов cause_sig. В нормальной ситуации этот сигнал привел к завершению процесса и выводу дампа ядра. Но в данном случае обычные последствия не имеют места, так как для трассируемого процесса менеджер памяти перехватывает все сигналы за исключением SIGKILL и в ответ на них приостанавливает процесс, позволяя программе-отладчику управлять его дальнейшим выполнением.

Системный вызов exec вызывает небольшую аномалию. Делая системный вызов, процесс отправляет менеджеру памяти сообщение и блокируется. Для других системных вызовов ответное сообщение вновь запускает процесс. Но в случае exec отклика нет, поскольку только что загруженный образ процесса не ожидает сигналов. Поэтому обслуживающая функция do_exec разблокирует процесс самостоятельно, сбрасывая флаг RECEIVING (19 строка кода функции). Следующей строкой кода образ приводится в состояние готовности к запуску, при этом, чтобы избежать возможного состояния состязания, используется функция lock_ready. В завершение сохраняется командная строка, чтобы процесс можно было

опознать, когда пользователь нажмет в консоли F1, просматривая состояние всех процессов.

В MINIX процессы могут завершать свою работу либо сделав системный вызов exit, либо получив сигнал. И в том и в другом случае менеджер памяти уведомляет ядро при помощи сообщения SYS_XIT. Работу здесь выполняет функция do_xit, которая сложнее, чем можно было бы ожидать. Учетная информация о процессе обрабатывается просто. Сигнальный таймер, если таковой есть, уничтожается, для этого поверх него записывается нулевое значение, исходя из того, что задача часов, когда истекает какой-либо таймер, всегда проверяет, интересен ли он кому-нибудь. Сложность do_xit связана с тем, что в момент уничтожения процесс может находиться в очереди, пытаясь отправить или получить сообщение. В коде это проверяется, и если процесс обнаруживается в очереди сообщений какого-либо другого процесса, его запись аккуратно удаляется из этой очереди.

В противоположность предыдущему сообщению, которое было немного сложным, SYS_GETSP обрабатывается совершенно тривиально. Это сообщение менеджер памяти использует, чтобы выяснить значение указателя стека какого-либо процесса. Последнее необходимо системным вызовам brk и sbrk для того, чтобы определить, не произошло ли перекрытия сегмента данных и стека. Обработчиком сообщения является функция do_getsp.

Теперь мы перейдем к одному из тех немногих сообщений, в которых заинтересована только файловая система, это сообщение SYS_TIMES. Оно используется для реализации системного вызова times, возвращающего процессу-инициатору вызова информацию о процессорном времени. Все действия обработчика сводятся к тому, что он помещает запрошенные значения в ответное сообщение. Чтобы предупредить возможные состязания при доступе к счетчикам времени, его тело ограничивается вызовами lock и unlock.

Может случиться так, что менеджер памяти или файловая система обнаружат ошибку, делающую невозможной дальнейшую работу. Например, если во время запуска файловая система узнает, что главный блок корневого устройства невосстановимо поврежден, она паникует и отправляет ядру сообщение SYS_ABORT. Кроме того, суперпользователь может принудительно вернуть систему в монитор начальной загрузки или перезагрузить систему при помощи команды reboot, которая обращается к системному вызову reboot. В любом из этих случаев задача системы выполняет подпрограмму do_abort, которая при необходимости копирует в монитор загрузки инструкции, после чего в завершение процесса вызывает wreboot.

Большая часть работы по обработке сигналов выполняется менеджером памяти, который проверяет, в состоянии ли целевой процесс поймать или игнорировать сигнал, если отправитель это позволяет и т. д. Тем не менее менеджер памяти сам не может отправить сигнал, так как для этого требуется поместить некоторую информацию в стек процесса, получившего сигнал.

До появления POSIX работать с сигналами было проблематично, поскольку перехваченный сигнал восстанавливал принятый по умолчанию ответ на сигнал. Поэтому, если требовалось продолжать специальную обработку последующих

сигналов, программист не мог обеспечить надежность. Сигналы асинхронны, и второй сигнал мог прибыть до того, как программа вновь установит специальную обработку. Обработка сигналов в стиле POSIX решила это проблему, но решение далось ценой усложнения механизма. Старый вариант обработки сигналов мог быть реализован путем помещения в стек процесса-получателя некоторой информации, по аналогии с тем, как это делается при прерывании. Программист должен был написать обработчик, заканчивающийся инструкцией возврата, извлекающей из стека необходимую для продолжения работы информацию. В POSIX при получении сигнала объем сохраняемой информации слишком большой, что сопровождается неудобствами в использовании. Перед тем как процесс сможет вновь продолжить свое исполнение, требуется выполнить дополнительные действия. Поэтому менеджеру памяти при обработке сигнала приходится отправлять системной задаче два сообщения. Вознаграждением за дополнительные усилия является более надежная обработка сигналов.

Когда процессу нужно послать сигнал, системной задаче отправляется сообщение SYS_SENDSIG. Оно обрабатывается функцией do_sendsig. Необходимая для обработки POSIX-сигнала информация хранится в структуре sigcontext, поля которой содержат состояние регистров процессора, и структуре sigframe, описывающей, как сигналы должны обрабатываться процессом. Обе структуры нуждаются в инициализации, но главная задача do_sendsig — поместить требуемую информацию в стек процесса-адресата и изменить значения его счетчика команд и указателя стека, чтобы в следующий раз, когда планировщик запустит процесс, был выполнен код обработчика сигнала.

Когда обработчик сигнала в стиле POSIX завершает свою работу, он не вытаскивает из стека адрес, на котором было прервано нормальное исполнение процесса, как в случае старого варианта обработки сигналов. Программист, пишущий обработчик сигнала, заканчивает его инструкцией return (на языке программирования высокого уровня), но манипуляции со стеком, произведенные вызовом sendsig, приводят к тому, что выполнение return приводит к системному вызову sigreturn. После этого менеджер памяти отправляет системной задаче сообщение SYS_SIGRETURN, обрабатываемое функцией do_sigreturn. Эта функция копирует структуру sigcontext обратно в адресное пространство ядра и восстанавливает регистры процесса. В следующий раз, когда планировщик запустит процесс, выполнение последнего будет продолжено, и вся информация о ранее установленных обработчиках сигналов будет сохранена.

Системный вызов sigreturn, в отличие от многих других вызовов, упомянутых ранее в этом разделе, не проистекает из стандарта POSIX. Это изобретение MINIX, с целью удобства выполнения дополнительных действий, требуемых после завершения обработчика сигнала. Программисты не должны использовать этот системный вызов, так как он не будет распознан другими операционными системами, кроме того, в любом случае этот вызов не должен делаться явно.

Некоторые сигналы передаются из самого ядра или обрабатываются в ядре, прежде чем попасть к менеджеру памяти. Среди них — сигналы, берущие свое начало в задачах, например сигнал срабатывания таймера от задачи часов или сигнал от задачи терминала, вырабатываемый нажатием одной из специальных

клавиш. Сюда же относятся и сигналы об исключениях, обнаруживаемые процессором (например, деление на ноль или недопустимая инструкция). Сигналы, инициируемые файловой системой, также обрабатываются сначала в ядре. Когда файловой системе необходимо запросить у ядра генерацию подобного сигнала, она использует сообщение SYS_KILL. Название этого сигнала может ввести вас в заблуждение, оно не имеет ничего общего с системным вызовом kill, с помощью которого обычные процессы посылают сигналы. Обработчиком SYS_KILL служит функция do_kill, которая сначала делает обычные проверки корректности данных сообщения, а затем, чтобы передать сигнал процессу, взывает cause_sig. Сигналы, исходящие из ядра, также передаются через эту функцию. Она инициирует сигнал, предавая менеджеру памяти сообщение KSIG.

Закончив с одним из сигналов типа KSIG, менеджер памяти передает обратно системной задаче сообщение SYS_ENDSIG. Оно обрабатывается функцией do_endsig, которая уменьшает количество текущих сигналов, и, когда счетчик достигает нуля, сбрасывает у процесса бит SIGPENDING. Если после этого у процесса не остается никаких флагов, препятствующих его выполнению, вызывается lock_ready, чтобы вновь разрешить процессу работать.

Сообщение SYS_COPY — одно из наиболее часто используемых. Оно необходимо для того, чтобы позволить файловой системе и менеджеру памяти копировать блоки информации в память пользовательских процессов и из нее.

Когда пользователь делает системный вызов read, файловая система смотрит, нет ли в ее кэше затребованных блоков. Если нет, она посылает соответствующей дисковой задаче сообщение с просьбой загрузить этот блок в кэш. Затем она отправляет системной задаче сообщение, указывающее ей копировать этот блок пользовательскому процессу. В худшем случае для считывания блока необходимо семь сообщений, показанных на рис. 3.29. Эти сообщения составляют значительную часть накладных расходов в MINIX, являющихся ценой модульной структуры системы.

Отступая в сторону, нужно упомянуть, что на процессорах 8086, не имеющих защиты, было достаточно просто смошенничать и позволить файловой системе скопировать данные в адресное пространство пользовательского процесса, но это бы нарушало принципы построения системы. Если кому-то, у кого есть доступ к столь древней машине, захочется увеличить за счет этого механизма производительность, ему стоит внимательно изучить его «шестеренки», чтобы увидеть, от какого количества возможных проблем он избавляет себя за счет небольшой потери в эффективности. На компьютерах с процессорами Pentium, имеющими механизм защиты, подобное повышение быстродействия невозможно.

Запрос SYS_COPY обрабатывается прямолинейно. Обработчиком служит функция do_copy, основные действия которой сводятся к извлечению параметров из сообщения и к вызову phys_copy.

Одним из способов борьбы с неэффективностью механизма передачи сообщений является упаковка нескольких запросов в одно сообщение, в MINIX — SYS_VCOPY. Это сообщение содержит указатель на массив с параметрами копирования нескольких блоков памяти. В цикле в коде функции do_vcopy для каждого запроса из массива извлекаются адрес и длина исходного блока и адрес, куда его

необходимо скопировать, и все данные передаются phys_copy. Это подобно тому, как дисковые устройства могут выполнять несколько передач данных по одному запросу.

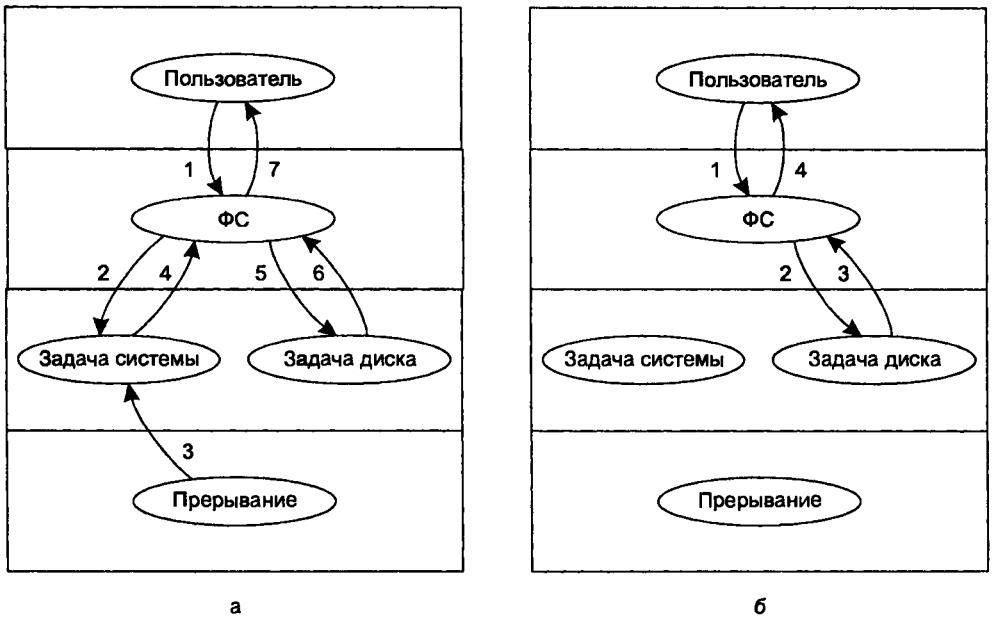


Рис. 3.29. *а* — в худшем случае для считывания блока требуется семь сообщений; *б* — в лучшем случае требуются четыре сообщения

Системная задача принимает еще несколько типов сообщений, которые большей частью довольно понятны. Два из них используются только при запуске системы. Файловая система, чтобы запросить параметры загрузки, передает сообщение SYS_GBOOT. В файле include/minix/boot.h есть структура, bparam_s, при помощи которой монитор начальной загрузки может передать различные параметры ядру системы. Выполняет эту операцию функция do_gboot, которая просто копирует данные из одной части памяти в другую. Кроме того, менеджер памяти при загрузке посылает системной задаче серии сообщений SYS_MEM, чтобы определить базовые адреса и длины доступных блоков памяти. Этот запрос обслуживает do_mem.

Сообщение SYS_UMAP используется процессами, не входящими в ядро, с целью вычислить физический адрес по заданному виртуальному. Расчет выполняет функция do_umap, вызывающая umap — функцию, используемую для этой цели в ядре.

Представитель последнего типа сообщений, на котором мы остановимся, — это SYS_TRACE. Оно обеспечивает работу системного вызова ptrace, применяемого для отладки. Отладка не является фундаментальной функцией операционной системы, но поддержка со стороны ОС идет ей только на пользу. Благодаря опе-

рационной системе отладчик может считывать и изменять память, принадлежащую отлаживаемому контексту, а также и регистры процессора, хранящиеся в таблице процессов, когда программа не работает.

Обычно процесс выполняется до тех пор, пока не заблокируется на операции ввода/вывода или не истечет его квант времени. Но большинство процессоров позволяют ограничить работу процесса выполнением единственной инструкции или сделать так, чтобы процесс выполнялся до тех пор, пока не будет достигнута определенная инструкция (для этого задается *точка останова*, breakpoint). При помощи подобных средств значительно упрощается детальный анализ программ.

Вызов ptrace помогает реализовать одиннадцать операций. Некоторые из них целиком перекладываются на менеджер памяти, но для большинства менеджер памяти посылает системной задаче сообщение SYS_TRACE, а та уже вызывает функцию do_trace. Внутри нее в операторе switch в зависимости от типа операции выполняется нужный код. Сами операции, в общем, просты. В MINIX бит P_STOP в таблице процессов означает, что процесс отлаживается, этот бит устанавливается отладчиком при необходимости остановить процесс (команда T_STOP) и сбрасывается, чтобы вновь позволить ему выполняться (команда T_RESUME). Отладка опирается на встроенную поддержку со стороны процессора, в процессорах Intel она контролируется одним из битов регистра признаков. Когда этот бит установлен, процессор, выполнив одну инструкцию, генерирует исключение SIGTRAP. Как ранее уже упоминалось, менеджер памяти останавливает отлаживаемую программу, если она получила сигнал. Значение бита TRACEBIT изменяется командами T_STOP и T_STEP. Контрольные точки могут устанавливаться двумя способами: либо при помощи команды T_SETINS, которая заменяет инструкцию специальным кодом, что приводит к генерации SIGTRAP, либо с помощью команды T_SETUSER изменить значение специального регистра точки останова. В любой системе, на которую переносима MINIX, вероятнее всего, отладчик можно будет реализовать при помощи подобных методик, но перенос этих функций потребует изучения конкретного аппаратного обеспечения.

Большинство выполняемых do_trace команд возвращают или модифицируют данные либо из адресного пространства отлаживаемого процесса, либо из его записи в таблице процессов. Разрешать менять значения некоторых регистров и флагов процессора слишком опасно, поэтому, чтобы предотвратить опасные действия, в команде T_SETUSER делается множество проверок.

В конце файла system.c находятся несколько вспомогательных функций, используемых в различных местах кода по всему ядру. Когда задаче необходимо вызвать сигнал (например, задача часов вызывает SIGALARM, а задача терминала — SIGINT), она обращается к функции cause_sig. Эта подпрограмма устанавливает бит в поле p_pending записи процесса в таблице процессов, после чего проверяет, ждет ли менеджер памяти сообщений от ANY (то есть находится в бездействии и ожидает следующего запроса). Если менеджер памяти отлеживается, она вызывает inform, чтобы он обработал сигнал.

Функция inform, как было описано выше, вызывается после проверки активности менеджера памяти. В дополнение к cause_sig, она вызывается из mini_rec (файл proc.c), когда менеджер памяти блокируется и остались текущие сигналы

ядра. Функция inform создает сообщение типа KSIG и отправляет его менеджеру памяти. После того как сообщение будет скопировано в приемный буфер менеджера памяти, вызвавший cause_sig процесс или задача продолжит свое выполнение. При этом процесс не ждет, когда получит управление менеджер памяти, как было бы в варианте обычного механизма обмена сообщениями, когда отправитель сообщения блокируется. Тем не менее, перед тем как завершить работу, cause_sig вызывает функцию lock_pick_proc, которая планирует менеджер памяти на запуск. Так как приоритет задач больше, чем у серверов, менеджер памяти не будет запущен до тех пор, пока не отработают все задачи. Планировщик получает управление, когда завершается вызвавшая сигнал задача. Если менеджер памяти окажется самым приоритетным среди готовых к запуску процессов, он будет запущен на выполнение.

Широко используемая подпрограмма umap преобразует виртуальные адреса в физические. Как уже упоминалось, она вызывается из do_umap, которая обслуживает сообщение SYS_UMAP. Аргументами являются указатель на запись в таблице процессов, принадлежащую тому процессу, виртуальный адрес которого преобразуется, флаг, обозначающий, какому из сегментов (код, данные или стек) принадлежит адрес, сам виртуальный адрес и количество байтов. Количество байтов принимается во внимание umap тогда, когда нужно проверить, укладывается ли буфер заданного размера, начинающийся с указанного виртуального адреса, в адресном пространстве процесса. В самом преобразовании адреса размер буфера не участвует. Функция umap используется всеми задачами, копирующими данные из адресного пространства пользовательских процессов или в него. Для драйверов устройств было бы удобно обращаться к umap, задавая процесс не указателем на запись в таблице процессов, а по номеру процесса. Есть такая функция — это numap. Чтобы преобразовать номер процесса в указатель, она вызывает proc_addr, а затем делает вызов umap.

Последняя в system.c — функция alloc_segments. Она вызывается из do_newmap, а также из процедуры ядра main при его инициализации. Код этой функции существенно зависит от аппаратного обеспечения. Она берет описания сегментов из записи в таблице процессов и манипулирует с регистрами и дескрипторами процессоров Pentium, ответственными за аппаратную защиту памяти.

Резюме

Вводом/выводом часто пренебрегают, хотя он заслуживает более серьезного отношения. Значительная доля любой операционной системы связана с вводом/выводом. Мы начали с рассмотрения аппаратного обеспечения ввода/вывода и связи устройств ввода/вывода с контроллерами. Затем мы рассмотрели четыре уровня программного обеспечения ввода/вывода: обработчики прерываний, драйверы устройств, независимое от устройств программное обеспечение и библиотеки плюс спулеры, работающие в пространстве пользователя.

Далее мы изучили проблему взаимной блокировки и инструментарий борьбы с ней. Взаимная блокировка возникает, когда имеется группа процессов, полу-

чивших монопольный доступ к некоторым ресурсам, и каждому процессу в группе необходим помимо этого другой ресурс, принадлежащий другому процессу. В таком случае все процессы блокируются и никогда не будут выполнены. Взаимную блокировку можно исключить, если система построена в расчете на упреждение подобной ситуации. Например, можно разрешить каждому процессу удерживать не более одного ресурса в каждый момент времени. Другой способ избежания блокировки — проверять каждый запрос, определяя, ведет ли он к опасной ситуации (в которой возможно возникновение блокировки), и отменять или откладывать опасные запросы.

В MINIX драйверы устройств реализованы в виде процессов, встроенных в ядро. Мы рассмотрели драйверы RAM-диска, жесткого диска, часов и драйвер терминала. Задача синхронного таймера и задача системы не являются драйверами устройств, но имеют точно такую же структуру. У каждой из этих задач есть главный цикл, в котором принимаются и обрабатываются запросы, в конечном итоге формируется и отправляется ответное сообщение о результатах. Все задачи разделяют общее адресное пространство. Задачи RAM-диска, жесткого диска и гибкого диска используют один общий главный цикл и разделяют некоторые функции. Тем не менее каждая задача является независимым процессом. Несколько различных терминалов на системной консоли, последовательные интерфейсы и сетевые подключения обслуживаются единственной задачей терминала.

Драйверы устройств по-разному взаимодействуют с системой прерываний. Устройства, которые могут выполнить свою работу быстро, например RAM-диск или отображаемый в память экран, вообще не прибегают к прерываниям. У жесткого диска большая часть работы выполняется в самом коде задачи, а обработчики прерываний возвращают информацию о состоянии. Обработчик прерываний часов самостоятельно выполняет некоторые подсчеты и отправляет сообщение задаче часов, когда остается что-то, что не может быть выполнено в обработчике. Обработчик прерываний клавиатуры только накапливает введенные символы и никогда не отправляет сообщений своей задаче. Вместо этого он меняет значение переменной, которую проверяет обработчик прерываний часов; последний и отправляет сообщение драйверу терминала.

Вопросы

1. Благодаря прогрессу в сфере технологии производства микросхем стало возможным поместить в одну недорогую микросхему весь контроллер, включая всю логику доступа к шине. Как это повлияло на модель, изображенную на рис. 3.1?
2. Если бы контроллер диска записывал принятые от диска байты в память по мере их получения, без внутренней буферизации, имело бы смысл чередование? Обоснуйте ответ.
3. Основываясь на скорости вращения и геометрии дисков дисковода и жесткого диска, скажите, какова битовая скорость обмена данными между бу-

ферами контроллера и самим диском. Сравните с другими формами ввода/вывода (последовательные линии и сеть).

4. Представьте себе гибкий диск, у которого шаг чередования секторов равен двум, как на рис. 3.3, *в*. На каждой дорожке диска восемь секторов по 512 байт. Скорость вращения диска 300 об/мин. Сколько времени потребуется, чтобы прочитать все секторы дорожки в правильном порядке, если предположить, что головка диска уже позиционирована на нужную дорожку, а чтобы сектор 0 переместился под головку, требуется 1/2 оборота диска? Чему равна скорость считывания данных? Теперь повторите то же задание для диска без чередования с теми же характеристиками. Как сильно снижается скорость из-за чередования?

5. Коммутатор терминалов DM-11, который давным-давно использовался на PDP-11, чтобы определить, является ли входящий бит единицей или нулем, считывал значение с каждой из линий в семь раз чаще скорости передачи данных. Считывание значения с одной линии требует 5,7 мс. Сколько линий со скоростью 1200 бод мог бы поддерживать DM-11?

6. Локальная сеть используется следующим образом. Пользователь обращается к системному вызову, чтобы записать пакеты данных в сеть. Затем операционная система копирует данные в буфер ядра. После этого данные копируются в схему сетевого адаптера. После того как все байты попадают в контроллер, они посылаются по сети со скоростью 10 Мбит/с. Получающий данные сетевой контроллер сохраняет каждый бит спустя 1 мкс после его отправки. Когда последний бит получен, центральный процессор компьютера-адресата прерывается, и ядро копирует прибывший пакет в свой буфер, чтобы исследовать его. Поняв, какому пользователю предназначается пакет, ядро копирует данные в пространство пользователя. Если предположить, что каждое прерывание и его обработка занимает 1 мс, размер пакетов равен 1024 байт (не считая заголовков), а копирование одного байта 1 мкс, то чему будет равна максимальная скорость, с которой один процесс может передавать данные другому процессу? Предположите, что отправитель блокируется, пока получатель не закончит работу и не отправит обратно подтверждение. Для простоты допустим, что временем получения подтверждения можно пренебречь.

7. Что такое независимость от устройств?

8. В каком из четырех уровней программного обеспечения ввода/вывода выполняются следующие действия:
   1) вычисление номеров дорожки, сектора и головки для чтения диска;
   2) поддержание кэша последних блоков;
   3) запись команд в регистры устройства;
   4) проверка разрешения доступа пользователя к устройству;
   5) преобразование двоичного целого числа в ASCII-символы для вывода на печать.

9. Почему файлы, посылаемые на принтер, обычно перед печатью накапливаются на диске?
10. Рассмотрим рис. 3.6. Предположим, что на шаге *n* процесс C вместо ресурса R запрашивает ресурс S. Приведет ли это к взаимоблокировке? А если он запросит оба ресурса, то есть и S и R?
11. Внимательно посмотрите на рис. 3.8, *б*. Если процесс D запросит еще одну единицу, приведет это к безопасному состоянию или к небезопасному? Что будет, если запрос поступит от процесса C вместо процесса D?
12. Все траектории на рис. 3.9 горизонтальны или вертикальны. Можете ли вы представить себе условия, при которых также были бы возможны наклонные траектории?
13. Предположим, процесс A на рис. 3.10 запросил последний ленточный накопитель. Приведет ли это к взаимной блокировке?
14. У компьютера есть шесть накопителей на магнитной ленте и *n* процессов, соревнующихся за право их использовать. Каждому процессу может потребоваться два накопителя. При каких значениях *n* в системе не будет тупиков?
15. Может ли система находиться в состоянии, не являющимся ни состоянием взаимоблокировки, ни безопасным состоянием? Если да, приведите пример. Если нет, докажите, что все состояния либо являются тупиками, либо они безопасны.
16. Распределенная система, использующая почтовые ящики, имеет два примитива межпроцессного взаимодействия: send (послать) и receive (получить). Второй примитив указывает процесс, от которого следует получить сообщение, и блокируется, если сообщения от процесса недоступны, даже несмотря на то, что могут ожидаться сообщения от других процессов. Здесь нет общих ресурсов, но процессам необходимо часто связываться друг с другом относительно других вопросов. Возможна ли взаимоблокировка? Аргументируйте ответ.
17. Сотни одинаковых процессов в электронных системах межбанковского перевода денежных средств работают следующим образом. Каждый процесс читает входную строку, определяющую количество денег для перевода, кредитовый и дебетовый счета. Затем он блокирует оба счета и выполняет транзакцию, а после завершения перевода снимает блокировку. При параллельно работающем большом количестве процессов существует опасность, что, имея заблокированным счет x, процесс будет неспособен заблокировать счет y, поскольку счет y уже окажется заблокированным процессом, в данный момент ожидающим счет x. Разработайте схему действия, избегающую взаимоблокировок. Не освобождайте запись счета до тех пор, пока вы не закончите транзакцию. (Иначе говоря, не позволяются решения, в которых один счет блокируется и затем немедленно освобождается, если другие счета заблокированы.)

18. Алгоритм банкира работает в системе, где есть m классов ресурсов и n процессов. При стремящихся к бесконечности m и n количество операций, которое нужно выполнить для проверки безопасности состояния, пропорционально $m^a n^b$. Что представляют собой величины a и b?

19. Золушка и Принц расторгают брак. Чтобы разделить свое имущество, они согласились на следующий алгоритм. Каждое утро любой из них может послать письмо адвокату другого, в котором запрашивает один предмет имущества. Поскольку день уходит на доставку писем, они пришли к соглашению, что если оба обнаруживают, что запросили один и тот же предмет в один и тот же день, на следующий день они посылают письмо с отменой запроса. Среди прочего имущества у них есть собака Вуфер, конура Вуфера, их канарейка Твитер и клетка Твитера. Животные любят свои жилища, поэтому было принято соглашение, что любой вариант раздела имущества, отделяющий животное от его дома, является недействительным, после которого весь раздел имущества требуется начать заново. И Золушка и Принц отчаянно хотят заполучить Вуфера. Поскольку они могут уехать (отдельно друг от друга) в отпуск, каждый супруг запрограммировал персональный компьютер для обработки переговоров. Когда они возвращаются из отпусков, компьютеры все еще ведут переговоры. Почему? Возможна ли взаимоблокировка? Возможно ли «голодание»? Аргументируйте ответ.

20. Сообщение, формат которого показан в табл. 3.4, используется для того, чтобы отправлять запросы драйверам блочных устройств. Какие поля можно было бы опустить в сообщении для символьных устройств и есть ли такие поля?

21. Драйвер диска получает запросы на чтение/запись к цилиндрам 10, 22, 20, 2, 40, 6 и 38. Перемещение блока головок с одного цилиндра на соседний занимает 6 мс. Сколько потребуется времени на перемещение головок при использовании алгоритма:

    1) обслуживания в порядке поступления запросов;

    2) обслуживания в первую очередь ближайшего цилиндра;

    3) элеваторного (сначала блок головок двигается вверх);

    4) во всех случаях начальное положение блока головок на цилиндре 20.

22. Продавец персональных компьютеров, посещая университет на юго-западе Амстердама для продажи партии компьютеров, заявляет, что его компания приложила существенные усилия по ускорению их версии UNIX. В качестве примера он отмечает, что в их драйвере диска применяется элеваторный алгоритм, а также обслуживание очереди запросов к одному цилиндру в порядке секторов. На студента Гарри Хакера его речь производит настолько сильное впечатление, что он покупает один компьютер. Гарри приносит компьютер домой и пишет программу, читающую случайные 10 000 блоков диска. К его изумлению, замеренная им производительность идентична той, которой можно было ожидать при использовании алгорит-

ма обслуживания запросов в порядке поступления. Означает ли это, что продавец лгал?

23. В UNIX каждый процесс состоит из двух частей: в адресном пространстве пользователя и ядра. Является ли ядерная часть подпрограммой или сопрограммой?
24. На некотором компьютере обработчик прерываний от таймера выполняет свои действия за 2 мс (включая накладные расходы по переключению процессов). Прерывания от таймера поступают с частотой 60 Гц. Какая часть времени работы центрального процессора расходуется на таймер?
25. В тексте было описано два применения сторожевых таймеров: ожидание запуска двигателя дисковода и выполнение возврата каретки на печатных терминалах. Приведите еще один пример.
26. Почему терминалы, использующие интерфейс RS-232, управляются прерываниями, а терминалы с отображением на память — нет?
27. Рассмотрим работу терминала. Драйвер посылает один символ, после чего блокируется. За передачей символа в линию следует прерывание, затем драйвер разблокируется и посылает следующий символ и т. д. Какую часть времени центрального процессора занимает управление модемом, если обработка прерывания, вывод одного символа и каждая блокировка требует 4 мс? Будет ли этот метод работать с линиями 110 бод? А что насчет линий 4800 бод?
28. Вообразите в памяти терминал, содержащий 1200 × 800 пикселов. Для прокрутки окна центральный процессор (или контроллер) должен переместить все строки текста вверх, копируя их биты из одной части видеопамяти в другую. Допустим, в окне 66 строк по 80 символов в строке (всего 5280 символов), а каждый символ имеет 8 пикселов в ширину и 16 пикселов в высоту. Сколько времени займет прокрутка всего окна, если для копирования одного байта требуется 500 нс? Если все строки имеют по 80 символов в длину, чему будет равна эквивалентная скорость терминала в бодах? Помещение одного символа на экран занимает 50 мкс. Подсчитайте скорость, если терминал цветной, 4 бит/пиксел.
29. Для чего в операционных системах нужны ESC-последовательности, например CTRL+V в MINIX?
30. Получив символ DEL (SIGINT), драйвер экрана MINIX очищает всю очередь на вывод для этого экрана. Почему?
31. У многих терминалов, использующих интерфейс RS-232, есть ESC-последовательности для удаления текущей строки и перемещения всех нижних строк на одну строку вверх. Как, по-вашему, реализована эта операция внутри терминала?
32. На оригинальном компьютере IBM PC с цветным дисплеем запись в видеопамять в любое время, кроме того интервала, когда электронный луч совершал вертикальный обратный ход, вызывала появление уродливых пятен по всему экрану. На экран выводятся 25 строк по 80 символов, каж-

дый из которых помещается в квадрат 8 × 8 пикселов. Каждый ряд из 640 пикселов рисуется за один горизонтальный проход луча, что занимает 63,6 мкс, включая горизонтальное обратное движение луча. Экран перерисовывается 60 раз/с. При каждом выводе экрана требуется период времени на вертикальный обратный ход луча. Какую часть времени видеопамять оказывается доступной для записи?

33. Напишите графический драйвер для цветного дисплея IBM или любого другого подходящего растрового дисплея. Драйвер должен воспринимать команды, устанавливающие цвет отдельных пикселов, перемещающие прямоугольные блоки по экрану и любые другие, какие вы сочтете интересными. Пользовательские программы должны взаимодействовать с драйвером, открывая /dev/graphics и записывая туда команды.
34. Модифицируйте драйвер дисковода в MINIX, реализовав кэширование дорожек.
35. Напишите драйвер дисковода, который работает как символьное, а не блочное устройство, чтобы обойти системный кэш блоков. Таким образом, пользователи смогут считывать большие блоки данных, которые при помощи DMA копируются непосредственно в адресное пространство пользователя, что значительно повышает производительность. Такой драйвер прежде всего был бы интересен программам, считывающим необработанное содержимое диска, не используя файловую систему. В эту категорию программ попадают программы, проверяющие структуру файловой системы.
36. Реализуйте системный вызов UNIX profil, которого нет в MINIX.
37. Измените драйвер терминала в MINIX так, чтобы в дополнение к специальной клавише, удаляющей последний введенный символ, была клавиша, удаляющая последнее слово.
38. В систему MINIX был добавлен новый диск со сменным носителем. Этому устройству необходимо раскручивать диск каждый раз, после того как сменен носитель, и время раскрутки довольно велико. Ожидается, что в процессе работы системы носитель будет часто сменяться. Неожиданно подпрограмма waitfor из at_wini.c оказалась неудовлетворительной. Разработайте новый вариант процедуры waitfor, которая после одной секунды активного ожидания нужного значения будет работать так: задача диска засыпает на одну секунду, затем проверяет значение порта, до тех пор пока не будет обнаружено нужное значение или не истечет заданный параметром TIMEOUT интервал времени.

Глава 4
Управление памятью

Память представляет собой важный ресурс, требующий тщательного управления. Несмотря на то что в наши дни память среднего домашнего компьютера в тысячи раз превышает ресурсы IBM 7094 — машины, бывшей в начале 60-х годов самой мощной в мире, — программы все равно превосходят в росте компьютерную память. Перефразированный закон Паркинсона гласит: «Программное обеспечение увеличивается в размерах до тех пор, пока не заполнит всю доступную на данный момент память».

В этой главе мы рассмотрим, как операционная система управляет памятью. В идеале каждый программист хотел бы иметь неограниченную по объему и скорости память, при этом также являющуюся энергонезависимой, то есть сохраняющую свое содержимое при выключении электричества — отключении от источников питания. Раз уж мы взялись за эту тему, то почему бы заодно не помечтать о дешевой памяти? К сожалению, технологии не могут обеспечить подобные пожелания. Вследствие этого память в компьютерах имеет *иерархическую структуру*. Небольшая часть ее представляет собой очень быструю, дорогую, энергозависимую (то есть теряющую информацию при выключении питания) кэш-память. Кроме того, компьютеры обладают десятками мегабайтов среднескоростной, средней ценовой категории, также энергозависимой оперативной памяти ОЗУ (RAM) и десятками или сотнями гигабайтов медленного, дешевого, энергонезависимого пространства на жестком диске. Одной из задач операционной системы является координация использования всех этих составляющих.

Часть операционной системы, отвечающая за управление памятью, называется *модулем управления памятью* или *менеджером памяти*. Он следит за тем, какая часть памяти используется в данный момент, а какая — свободна; при необходимости выделяет память процессам и по их завершении освобождает ресурсы; управляет обменом данных между оперативной памятью и диском, если та слишком мала для того, чтобы вместить все процессы.

В этой главе мы изучим несколько различных схем управления памятью, от самой простой до весьма сложной и запутанной. Но начнем мы с самого начала и прежде всего рассмотрим наиболее элементарную систему управления памятью, а затем постепенно перейдем ко все более и более совершенным конструкциям.

Ближе к началу книги мы обращали внимание на то, что в компьютерном мире история имеет тенденцию к повторениям, и хотя простейшие схемы управления памятью больше не используются в настольных ПК, они все еще работают в карманных компьютерах, встроенных системах и смарт-картах. Именно по этой причине их до сих пор стоит изучать.

4.1. Простые способы управления памятью

Системы управления памятью можно разделить на два класса: перемещающие процессы между оперативной памятью и диском во время их выполнения и те, которые этого не делают. Второй вариант проще, поэтому начнем с него, а способы с подкачкой (подкачка процессов полностью — swapping или страничная — paging) мы изучим во вторую очередь. Читая главу 4, следует помнить, что обычный и постраничный варианты подкачки в значительной степени являются искусственными процессами, вызванными отсутствием достаточного количества оперативной памяти для одновременного хранения всех программ. Если же когда-нибудь оперативная память настолько увеличится в размерах, что ее будет достаточно для любых целей, аргументы в пользу той или иной схемы управления могут потерять актуальность.

4.1.1. Однозадачная система без подкачки на диск

Самая простая из возможных моделей управления памятью заключается в том, что в каждый конкретный момент времени работает только одна программа, при этом память разделяется между программами и операционной системой. На рис. 4.1 показаны три варианта такой схемы. Операционная система может находиться в нижней части памяти, то есть в ОЗУ (оперативное запоминающее устройство, RAM (Random Access Memory — память с произвольным доступом)); см. рис. 4.1, *а*.

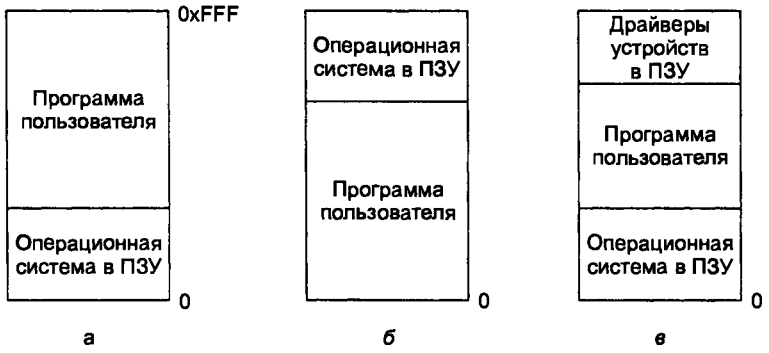


Рис. 4.1. Три простейшие модели организации памяти при наличии операционной системы и одного пользовательского процесса. Существуют также и другие возможные варианты

Или же операционная система может располагаться в самой верхней части памяти — в ПЗУ (постоянное запоминающее устройство, ROM (Read-Only Memory — память только для чтения)), как продемонстрировано на рис. 4.1, *б*. И третий способ: драйверы устройств могут размещаться в ПЗУ, а остальная часть системы — ниже в ОЗУ, как показано на рис. 4.1, *в*. Первая модель раньше

применялась на мэйнфреймах и мини-компьютерах, но в настоящее время практически не употребляется. Вторая схема сейчас используется в некоторых карманных компьютерах и встроенных системах, а третья модель устанавливалась на ранних персональных компьютерах (например, работающих с MS-DOS), при этом часть системы в ПЗУ носила название *BIOS* (Basic Input Output System – базовая система ввода/вывода).

Когда система организована таким образом, в каждый конкретный момент времени может работать только один процесс. Как только пользователь набирает команду, операционная система копирует запрашиваемую программу с диска в память и выполняет ее, а после окончания процесса выводит на экран приглашение и ждет новой команды. Получив инструкции, она загружает другую программу в память, записывая ее поверх предыдущей.

4.1.2. Многозадачность с фиксированными разделами

Однозадачные системы сложно использовать где-либо еще, кроме простейших встроенных систем. Большинство современных систем позволяет одновременный запуск нескольких процессов. Наличие нескольких процессов, работающих одномоментно, означает, что когда один процесс приостановлен в ожидании завершения операции ввода/вывода, другой вправе использовать центральный процессор. Таким образом, многозадачность увеличивает загрузку процессора. Сетевые серверы всегда имеют возможность одновременной работы нескольких процессов (для разных клиентов), но и большинство клиентских машин (то есть настольных компьютеров) в наши дни не уступают им в этом смысле.

Самый легкий способ достижения многозадачности представляет собой простое разделение памяти на n (возможно не равных) разделов. Такое разбиение можно выполнить, например, вручную при запуске системы.

Когда задание поступает в память, его можно расположить во входной очереди к наименьшему разделу, достаточно большому для того, чтобы вместить это задание. Так как в данной схеме размер разделов одинаков, все не используемое работающим процессом пространство в разделе пропадает. На рис. 4.2, *а* показано, как выглядит система с фиксированными разделами и отдельными очередями входных заданий.

Недостаток упорядочения входящих заданий по отдельным очередям становится очевидным, когда к большому разделу очередь отсутствует, в то время как к маленькому выстроилось в нетерпении довольно много задач; в нашем примере на рис. 4.2, *а* это разделы 1 и 3. Небольшие задания должны ждать своей очереди, чтобы попасть в память, и это все несмотря на то, что память в основном свободна. Альтернативная схема заключается в организации одной общей очереди для всех разделов, как показано на рис. 4.2, *б*: как только раздел освобождается, задачу, ближайшую к началу очереди и подходящую для выполнения в этом разделе, можно загрузить в него и начать ее обработку. Поскольку нежелательно тратить большие разделы на маленькие задачи, существует другая стратегия. Она состоит в том, что каждый раз после освобождения раздела происходит по-

иск в очереди наибольшего из приемлемых по размерам для этого раздела заданий, и именно это задание выбирается для обработки. Заметим, что последний алгоритм дискриминирует мелкие задачи, как недостойные того, чтобы под них отводился целый раздел, хотя обычно крайне желательно предоставить для небольших программ (часто интерактивных) элитное обслуживание.

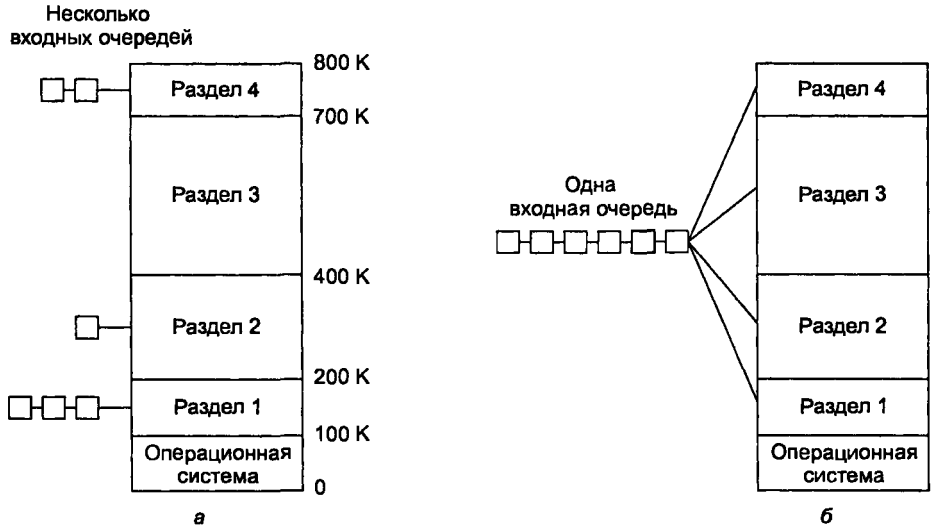


Рис. 4.2. *а* — фиксированные разделы памяти с отдельными входными очередями для каждого раздела; *б* — фиксированные разделы памяти с одной очередью на вход

Один из выходов из положения — создание хотя бы одного маленького раздела памяти, который позволит выполнять маленькие задания без долгого ожидания освобождения больших разделов.

Или же устанавливается следующее правило: задачу, имеющую право быть выбранной для обработки, можно держать в просителях не больше k раз. Каждый раз, когда ее выполнение откладывается, к счетчику добавляется единица. Когда значение счетчика становится равным k, игнорировать задачу более нельзя.

Подобная схема, где утром оператор задает фиксированные разделы и после этого они не изменяются, в течение многих лет практиковалась в системах OS/360 на больших мэйнфреймах компании IBM. Она носила название *MFT* (Multiprogramming with a Fixed number of Tasks — мультипрограммирование с фиксированным количеством задач, или OS/MFT). Она легка для понимания и не менее проста в исполнении: входящее задание стоит в очереди до тех пор, пока не станет доступным соответствующий раздел, затем оно загружается в этот раздел памяти и там работает до завершения процесса. Сейчас очень мало (если они вообще сохранились) операционных систем, поддерживающих такую модель.

Настройка адресов и защита

Многозадачность вносит две существенные проблемы, требующие решения, — это настройка адресов для перемещения программы в памяти и защита. Посмотрите на рис. 4.2. Из рисунка становится ясно, что разные задачи будут загружены по различным адресам. Когда программа компонуется (то есть в едином адресном пространстве объединяются основной модуль, написанные пользователем процедуры и библиотечные подпрограммы), компоновщик должен знать, с какого адреса будет начинаться программа в памяти.

Например, предположим, что первая команда представляет собой вызов процедуры с абсолютным адресом 100 внутри двоичного файла, создаваемого компоновщиком. Если эта программа загружается в раздел 1 (по адресу 100 К), команда обратится к абсолютному адресу 100, принадлежащему операционной системе. А нужно вызвать процедуру по адресу 100 К + 100. Если же программа загружается во второй раздел, команду нужно переадресовать на 200 К + 100 и т. д. Эта проблема известна как проблема *перемещения программ в памяти* или *настройки адресов.*

Одним из возможных решений является модификация команд во время загрузки программы в память. В программе, загружаемой в первый раздел, к каждому адресу прибавляется 100 К, в программе, которая попадает во второй раздел, к адресам добавляется 200 К и т. д. Чтобы выполнить подобную настройку адресов во время загрузки, компоновщик должен включить в двоичную программу список или битовый массив с информацией о том, какие слова в программе являются адресами (и их нужно перераспределить), а какие — кодами машинных команд, постоянными или другими частями программы, которые не нужно изменять. Таким образом работает операционная система OS/MFT.

Настройка адресов во время загрузки не решает проблемы защиты. Вредоносные программы всегда могут перескочить на какую-нибудь новую команду вместо ожидаемой. Поскольку при такой системе предпочтительно используется абсолютная адресация памяти, а не смещение относительно содержимого какого-либо регистра, не существует способа, который позволил бы запретить программе обращаться к любому слову в памяти для его чтения или записи. В многопользовательских системах крайне нежелательно разрешать процессам доступ к области памяти, принадлежащей другим пользователям.

Для защиты компьютера IBM 360 разработчики приняли следующее решение: они разделили память на блоки по 2 Кбайт и назначили каждому блоку 4-битный защитный код. Регистр PSW (Program Status Word — слово состояния программы) содержал 4-разрядный ключ. Аппаратура IBM 360 перехватывала все попытки работающих процессов обратиться к любой части памяти, защитный код которой отличался от содержимого регистра слова состояния программы. Поскольку только операционная система была вправе изменять коды защиты и ключи, предотвращалось вмешательство пользовательских процессов в дела друг друга и в работу операционной системы.

Альтернативное решение сразу обеих проблем (защиты и перераспределения) заключается в оснащении машины двумя специальными аппаратными регистрами, называемыми *базовым* и *ограничительным регистрами.* При плани-

ровании процесса в базовый регистр загружается адрес начала раздела памяти, а в ограничивающий регистр — длина раздела. К каждому автоматически формируемому адресу перед его передачей в память прибавляется содержимое базового регистра. Таким образом, если базовый регистр содержит величину 100 К, команда CALL 100 будет превращена в команду CALL 100K+100 без изменения самой команды. Кроме того, адреса проверяются по отношению к лимитирующему регистру для гарантии, что они не используются для адресации памяти вне текущего раздела. Базовый и ограничительный регистры защищаются аппаратно, чтобы не допустить их изменений пользовательскими программами.

Неудобство, присущее этой схеме, — требуется выполнять операции сложения и сравнения при каждом обращении к памяти. Операция сравнения может быть выполнена быстро, но сложение относительно нее — медленное действие, что обусловлено временем распространения сигнала, за исключением тех случаев, когда применяется специальная микросхема сложения.

Такая система адресации использовалась в CDC 6600 — первом суперкомпьютере в мире. В центральном процессоре Intel 8088 для первых IBM PC применялась упрощенная ее версия: были базовые регистры, но отсутствовали ограничительные. Сейчас эту схему можно встретить лишь в немногих компьютерах.

4.2. Подкачка

Организация памяти в виде фиксированных разделов проста и эффективна для работы с пакетными системами. Каждое задание, по мере продвижения в начало очереди, загружается в раздел памяти и остается там до своего завершения. До тех пор пока в памяти может храниться достаточное количество задач для обеспечения постоянной занятости центрального процессора, нет причин что-либо усложнять.

Но совершенно другая ситуация сложилась с системами разделения времени или персональными компьютерами, ориентированными на работу с графикой. Оперативной памяти иногда оказывается недостаточно для того, чтобы вместить все текущие активные процессы, и тогда избыток процессов приходится хранить на диске, а для обработки динамически переносить их в память.

Существует два основных подхода к управлению памятью, зависящие (отчасти) от доступного аппаратного обеспечения. Самая простая стратегия, называемая *свопингом* (swapping) или *обычной подкачкой*, заключается в том, что каждый процесс полностью копируется в память, работает некоторое время и затем полностью же возвращается на диск. Другая стратегия, носящая название *виртуальной памяти*, позволяет программам работать даже тогда, когда они только частично находятся в оперативной памяти. Ниже мы изучим свопинг, а вторую стратегию рассмотрим в разделе «Виртуальная память» данной главы.

Работа системы свопинга проиллюстрирована на рис. 4.3. На начальной стадии в памяти находится только процесс A. Затем создаются или загружаются с диска процессы B и C. На рис. 4.3, *г* процесс A выгружается на диск. Затем появляется процесс D, а процесс B завершается. Наконец, процесс A снова возвраща-

ется в память. Так как теперь процесс A расположен в другом месте, его адреса должны быть перенастроены или программно во время загрузки в память, или (более заманчивый вариант) аппаратно во время выполнения программы.

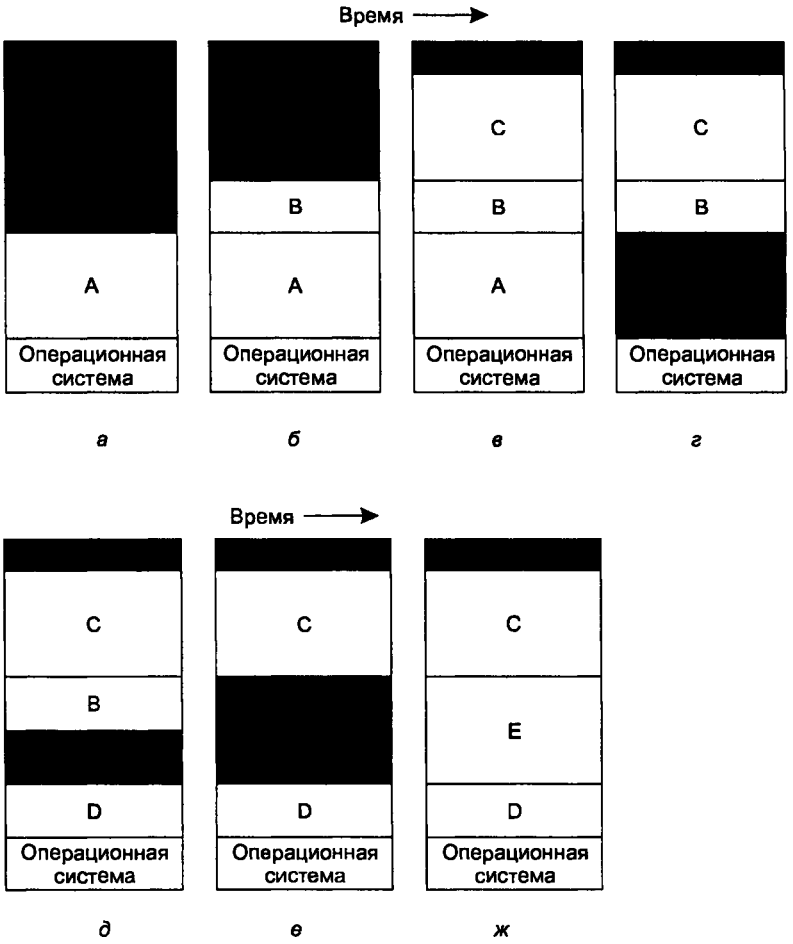


Рис. 4.3. Распределение памяти изменяется по мере того, как процессы поступают в память и покидают ее. Заштрихованы неиспользуемые области памяти

Основная разница между фиксированными разделами на рис. 4.2 и непостоянными разделами на рис. 4.3 заключается в том, что во втором случае количество, размещение и размер разделов изменяются динамически по ходу поступления и завершения процессов, тогда как в первом варианте все эти параметры фиксированы. Гибкость схемы, в которой нет ограничений, связанных с определенным количеством разделов, и где каждый из разделов может быть очень большим или совсем маленьким, оптимизирует использование памяти, но, кроме

того, усложняет операции размещения процессов и освобождения памяти, а также отслеживание происходящих изменений.

Когда в результате подкачки процессов с диска в памяти появляется множество неиспользованных фрагментов, их можно объединить в один большой блок, передвинув все процессы в сторону младших адресов настолько, насколько это возможно. Такая операция называется *уплотнением* или *сжатием памяти*. Обычно ее не выполняют по причине экономии времени работы процессора. Например, на машине с 256 Мбайт оперативной памяти, которая может копировать 4 байта за 40 нс, уплотнение всей памяти займет около 2,7 с.

Еще один момент, на который стоит обратить внимание: сколько памяти должно быть предоставлено процессу, когда он создается или копируется с диска? Если процесс имеет фиксированный — никогда не изменяющийся — размер, размещение происходит просто: операционная система предоставляет точно необходимое количество памяти, ни больше, ни меньше, чем нужно.

Однако если область данных процесса может расти, например, в результате динамического распределения памяти из кучи[1], что встроено во многие языки программирования, проблема предоставления памяти возникает каждый раз, когда процесс пытается увеличиться. Если участок неиспользованной памяти расположен рядом с процессом, его можно отдать в пользу процесса, таким образом позволив процессу вырасти на размер этой области. Если же процесс соседствует с другим процессом, для его увеличения нужно или переместить достаточно большой свободный участок памяти, или перегрузить на диск один или больше процессов, с целью создать незанятый фрагмент достаточного размера. Если процесс не может расти в памяти, а область на диске, предоставленная для подкачки, переполнена, процесс будет вынужден ждать освобождения памяти или же будет уничтожен.

Если предположить, что большинство процессов будут увеличиваться во время работы, вероятно, сразу стоит предоставлять им несколько больше памяти, чем требуется, а всякий раз, когда процесс копируется на диск или перемещается в памяти, нужно обрабатывать служебные данные, связанные с перемещением или подкачкой процессов, больше не умещающихся в предоставленной им памяти. Но когда процесс выгружается на диск, вместе с ним должно сохраняться содержимое только действительно используемого адресного пространства, так как очень расточительно также перемещать и придерживаемую им «про запас» память. На рис. 4.4, *а* показана конфигурация памяти с предоставлением пространства для роста двух процессов.

Если процесс имеет два наращиваемых сегмента, например сегмент данных, используемый как куча для динамически назначаемых и освобождаемых переменных, и сегмент стека для обычных локальных переменных и адресов возврата, предлагается альтернативная схема распределения памяти, показанная на рис. 4.4, *б*. Здесь мы видим, что у каждого процесса вверху предоставленной ему области памяти находится стек, который расширяется вниз, и сегмент данных,

[1] Кучей (heap) называется область памяти, выделяемая программе для динамически размещаемых структур данных. — *Примеч. перев.*

расположенный отдельно от текста программы и увеличиваемый вверх. Область памяти между ними является ничейной, задействуемой в интересах любого из сегментов. Если ее становится недостаточно, процесс нужно или перенести на другое, большее свободное место, или выгрузить на диск до появления свободного пространства необходимого размера, или уничтожить.

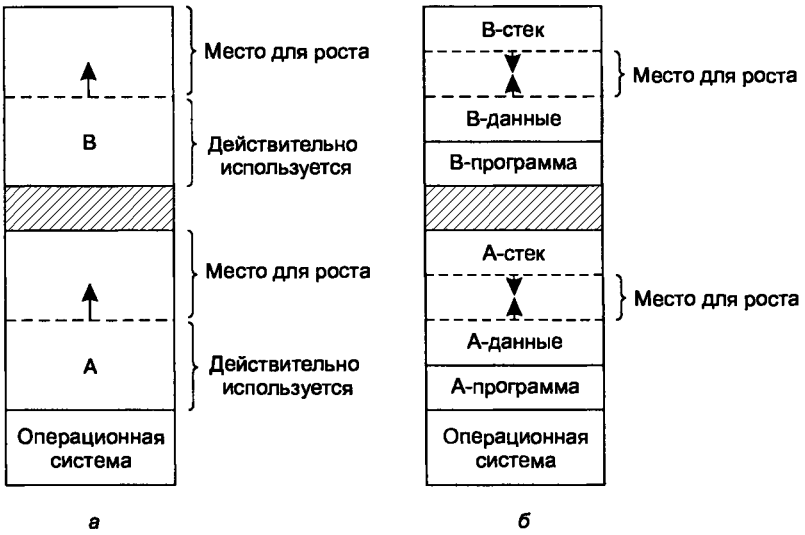


Рис. 4.4. *а* — предоставление пространства для роста области данных; *б* — предоставление пространства для роста стека и области данных

4.2.1. Управление памятью с помощью битовых массивов

Если память выделяется динамически, этим процессом должна управлять операционная система. Существует два способа учета использования памяти: *битовые массивы*, иногда называемые *битовыми картами*, и списки свободных участков. В этом и следующем разделах мы по очереди рассмотрим оба метода.

При работе с битовым массивом память разделяется на блоки размером от нескольких слов до нескольких килобайтов. В битовой карте каждому свободному блоку соответствует один нулевой бит, а каждому занятому блоку — бит, установленный в 1 (или наоборот). На рис. 4.5 показана часть памяти и соответствующий ей битовый массив.

Размер единичного — мерного — блока весьма важен на стадии разработки системы. Чем он меньше, тем больше битовый массив. Однако даже при маленьком «кванте» памяти в четыре байта, то есть для 32 битов памяти, потребуется 1 бит в карте. Тогда область размером в $32n$ будет соответствовать n битам карты, таким образом, битовая карта займет всего лишь 1/33 часть памяти. Если отдать предпочтение большим блокам, битовый массив становится меньше, но при

этом может оказаться неиспользуемой существенная часть последнего блока каждого процесса (если размер процесса не кратен размеру минимального блока).

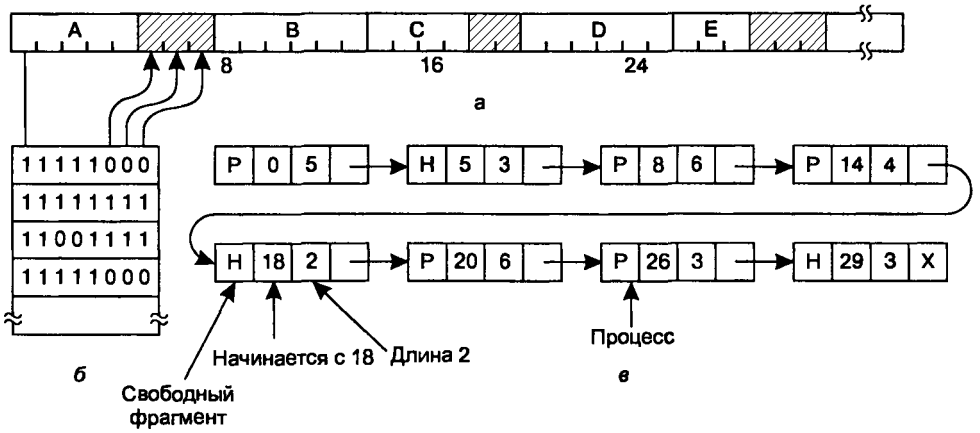


Рис. 4.5. *а* — область памяти с пятью процессами и тремя свободными участками; *б* — соответствующая битовая карта; *а* — та же информация в виде списка

Битовый массив предоставляет простой способ отслеживания слов в памяти фиксированного объема, поскольку в своих размерах отталкивается только от количества памяти и единичного блока. У этой схемы есть основная врожденная проблема — при решении переместить k-блочный процесс в память модуль управления памяти должен найти в битовой карте серию из k следующих друг за другом нулевых битов. Поиск серии заданной длины в битовой карте является медленной операцией (так как искомая последовательность битов может пересекать границы слов в битовом массиве). В этом состоит аргумент против битовых карт[1].

4.2.2. Управление памятью с помощью списков

Другой способ отслеживания состояния памяти предоставляет поддержка списков занятых и свободных фрагментов памяти, где фрагментом является или процесс, или участок между двумя процессами. Память, показанная на рис. 4.5, *а*, представлена в виде однонаправленного списка сегментов на рис. 4.5, *в*. Каждая запись в списке указывает: является ли область памяти свободной (H, от hole — дыра) или занятой процессом (P, process); адрес, с которого начинается эта область; ее длину; содержит указатель на следующую запись.

[1] Не совсем продуманное заявление мудрого Таненбаума. На самом деле все не так медленно, если использовать scasb и битовые команды (процессоры Intel). Например, битовые массивы были применены в незаслуженно забытой OS/2 для файловой системы HPFS (самой быстрой из существующих), с целью пометки занятых/свободных секторов. Проблема объема битового массива решалась просто разбиением диска на восьмимегабайтовые (по умолчанию) участки (полосы, bands) и хранением битовой карты (2 K) в конце каждого участка. — *Примеч. ред.*

В нашем примере список упорядочен по адресам. Такая сортировка имеет следующее преимущество: когда процесс завершается или выгружается на диск, изменение списка представляет собой тривиальную операцию. Процесс обычно имеет двух соседей (кроме случаев, когда он находится на самом «верху» или на «дне» памяти). Соседями могут быть процессы или свободные фрагменты, что приводит к четырем комбинациям, показанным на рис. 4.6. На рис. 4.6, *а* корректировка списка требует замены P на H. На рис. 4.6, *б* и 4.6, *в* две записи соединяются в одну, а список становится на запись короче. На рис. 4.6, *г* объединяются три записи, а из списка удаляются два элемента. Так как ячейка таблицы процессов для завершившегося процесса обычно будет непосредственно указывать на запись в списке для этого процесса, возможно, удобнее иметь список с двумя связями, чем с одной (последний показан на рис. 4.5, *в*). Такая двунаправленная структура упрощает поиск предыдущей записи и оценку возможности конкатенации.

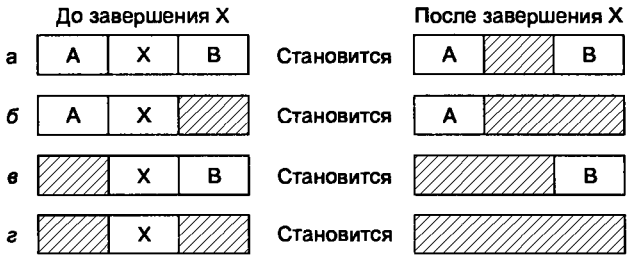


Рис. 4.6. Четыре комбинации соседства для завершения процесса X

Если процессы и свободные участки хранятся в списке, отсортированном по адресам, существует несколько алгоритмов для предоставления памяти процессу, создаваемому заново (или для существующих процессов, загружаемых с диска). Допустим, менеджер памяти знает, сколько памяти нужно предоставить. Простейший алгоритм представляет собой выбор *первого подходящего участка*. Менеджер памяти просматривает список областей до тех пор, пока не находит достаточно большой свободный участок. Затем этот участок делится на два: одна часть отдается процессу, а другая остается неиспользуемой. Так происходит всегда, кроме статистически нереального случая точного соответствия свободного участка и пожеланий процесса. Это быстрый алгоритм, поскольку поиск минимизирован настолько, насколько возможно.

Алгоритм *следующего подходящего участка* действует с минимальными отличиями от правила «первый подходящий». Он работает так же, как и первый алгоритм, но всякий раз, когда находится соответствующий свободный фрагмент, запоминается его адрес. И когда алгоритм в следующий раз вызывается для поиска, он стартует с того самого места, где остановился в прошлый раз, вместо того чтобы снова и снова начинать поиск от головы списка, как это делает алгоритм «первый подходящий». Моделирование работы алгоритма по Бэйсу показало, что производительность схемы «следующий подходящий» несколько хуже, чем «первый подходящий» [5].

Другой хорошо известный алгоритм называется *«самый подходящий участок»*. Здесь выполняется поиск по всему списку и выбирается наименьший по размеру подходящий свободный фрагмент. Вместо того чтобы делить большую незанятую область, которая может понадобиться позже, этот алгоритм пытается найти участок, наиболее приближенный к действительно необходимым размерам.

За примерами работы алгоритмов «первый подходящий» и «самый подходящий» снова обратимся к рис. 4.5. Если необходим блок размером 2, правило «первый подходящий» предоставит область по адресу 5, а схема «самый подходящий» разместит процесс в свободном фрагменте по адресу 18.

«Самый подходящий» медленнее «первого подходящего», так как алгоритм каждый раз должен производить поиск во всем списке. Но, что немного удивительно, он выдает еще более плохие результаты, чем «первый подходящий» или «следующий подходящий», поскольку стремится заполнить память очень маленькими, практически бесполезными свободными областями, то есть фрагментирует память. Для варианта «первый подходящий» в среднем характерны большие свободные фрагменты.

Раз «подходящие» алгоритмы не всегда устраивают, можно попытаться решить проблему разделения памяти на практически точно совпадающие с процессом области и маленькие свободные фрагменты, то есть представить алгоритм «самый неподходящий участок». Он всегда выбирает самый большой свободный фрагмент, размер которого после дробления остается еще достаточным для дальнейшего использования. Однако моделирование показало, что это также не очень подходящая идея.

Все четыре алгоритма можно ускорить, если поддерживать отдельные списки для процессов и свободных областей. Тогда поиск будет производиться только среди незанятых фрагментов. Неизбежная цена, которую придется заплатить за увеличение скорости при размещении процесса в памяти, заключается в дополнительной сложности и замедлении работы при освобождении областей памяти, так как ставший свободным фрагмент необходимо удалить из списка процессов и вставить в список незанятых участков.

Если для процессов и свободных фрагментов поддерживаются отдельные списки, то последний можно отсортировать по размеру, тогда алгоритм «самый подходящий» будет работать быстрее. Когда он выполняет поиск в списке свободных фрагментов от самого маленького к самому большому, то, как только находит приемлемую незанятую область, алгоритм уже знает, что она — наименьшая из тех, в которых может поместиться задание, то есть наилучшая. В отличие от схемы с одним списком, дальнейший поиск не требуется. Таким образом, если список свободных фрагментов отсортирован по длинам, схемы «первый подходящий» и «самый подходящий» одинаково быстры, а алгоритм «следующий подходящий» не имеет смысла.

При поддержке отдельных списков для процессов и свободных фрагментов возможна небольшая оптимизация. Вместо создания отдельного набора структур данных для списка свободных участков, как это сделано на рис. 4.5, *в*, можно использовать сами свободные области. Первое слово каждого незанятого фрагмента может содержать размер фрагмента, а второе слово может указывать на

следующую запись. Узлы списка на рис. 4.5, *в*, для которых требовались три слова и один бит (P/H), больше не нужны.

Еще один алгоритм распределения называется *«быстрый подходящий»*, он поддерживает отдельные списки для некоторых из наиболее часто запрашиваемых размеров. Например, могла бы существовать таблица с n записями, в которой первая запись указывает на начало списка свободных фрагментов размером 4 Кбайт, вторая запись является указателем на список незанятых областей размером 8 Кбайт, третья — 12 Кбайт и т. д. Свободный фрагмент размером, скажем, 21 байт мог бы располагаться или в списке областей 20 Кбайт или в специальном списке участков дополнительных размеров. Согласно правилу «быстрый подходящий» поиск фрагмента требуемого размера происходит чрезвычайно быстро. Но этот алгоритм имеет тот же недостаток, что и прочие схемы, которые сортируют свободные области по размеру, а именно: если процесс завершается или выгружается на диск, поиск его соседей с целью узнать, возможно ли их объединение, является затратной операцией. А если не производить слияния областей, память очень скоро окажется разбитой на огромное число маленьких свободных фрагментов, в которые не поместится ни один процесс.

4.3. Виртуальная память

Уже достаточно давно люди впервые столкнулись с проблемой размещения программ, оказавшихся слишком большими и поэтому не помещавшихся в доступной физической памяти. Обычно принималось решение о разделении программы на части, называемые *оверлеями* (overlays). Нулевой оверлей обычно запускался первым. По завершении своего выполнения он вызывал следующий оверлей. Некоторые системы с перекрытием были очень сложными, позволяющими одновременно находиться в памяти нескольким оверлеям. Оверлеи хранились на диске и по мере необходимости динамически перемещались между памятью и диском средствами операционной системы.

Несмотря на то что фактическая работа по загрузке оверлеев с диска и выгрузке на диск выполнялась системой, делить программы на части должен был программист. Разбиение больших программ на маленькие модули поглощало много времени и было не слишком интересным занятием. Однако такая ситуация длилась недолго, так как вскоре кто-то придумал способ поручить всю эту работу компьютеру.

Разработанный метод известен как *виртуальная память* [34]. Основная идея виртуальной памяти исходит из того, что совместный размер программы, данных и стека может превысить количество доступной физической памяти. Операционная система хранит части программы, использующиеся в настоящий момент, в оперативной памяти, остальные — на диске. Например, программа размером 16 Мбайт сможет работать на машине с 4 Мбайт памяти, если тщательно продумать, какие 4 Мбайт должны находиться в памяти в каждый момент времени. При этом части программы, находящиеся на диске, будут меняться местами с частями в памяти по мере необходимости.

Виртуальная память не теряет работоспособности в многозадачной системе при множестве одновременно загруженных в память программ. Когда программа ждет перемещения в память очередной ее части, она находится в состоянии ожидания ввода/вывода и не может работать, поэтому центральный процессор может быть отдан другому процессу тем же самым способом, как в любой другой многозадачной системе.

4.3.1. Страничная организация памяти

Большинство систем виртуальной памяти опираются на технику, называемую *страничной организацией памяти* (paging). На любом компьютере существует множество адресов в памяти, к которым может обратиться программа. Когда программа использует следующую инструкцию

```
MOVE REG,1000
```

она делает это для того, чтобы скопировать содержимое памяти по адресу 1000 в регистр REG (или наоборот, в зависимости от компьютера). Адреса могут формироваться с использованием индексации, базовых регистров, сегментных регистров и другими путями.

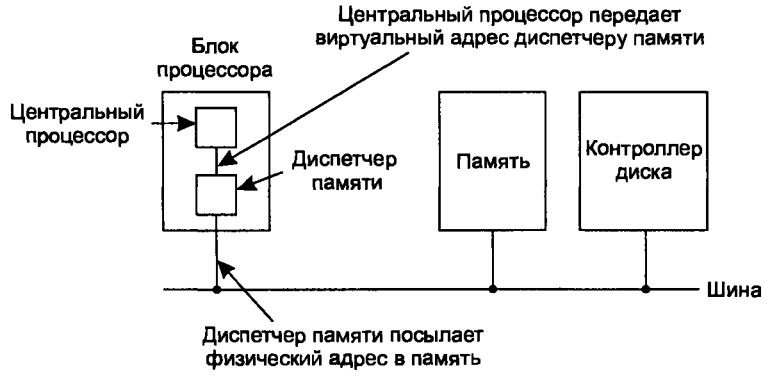


Рис. 4.7. Расположение и функции диспетчера памяти (MMU). Здесь диспетчер памяти показан как часть микросхемы процессора, в наши дни это обычно так и есть. Но логически он мог быть отдельной микросхемой, и так было некоторое время назад

Эти программно формируемые адреса, называемые *виртуальными адресами*, образуют *виртуальное адресное пространство*. На компьютерах без виртуальной памяти виртуальные адреса подаются непосредственно на шину памяти и при чтении или записи читается или записывается слово в физической памяти с тем же самым адресом. При применении виртуальной памяти виртуальные адреса не передаются напрямую шиной памяти. Вместо этого они направляются *диспетчеру памяти* (MMU — Memory Management Unit), который отображает виртуальные адреса на физические адреса, что продемонстрировано на рис. 4.7.

Очень простой пример того, как работает такого рода отображение, приведен на рис. 4.8. Мы рассматриваем компьютер, который может формировать 16-разрядные адреса, от 0 до 64 К. Это виртуальные адреса. Однако у этого компьютера есть только 32 Кбайт физической памяти, поэтому, хотя программы размером 64 Кбайт могут быть написаны, их нельзя целиком загрузить в память и запустить на выполнение. Полная копия образа памяти программы размером до 64 Кбайт должна присутствовать на диске, но в таком виде, чтобы ее можно было по мере надобности переносить в память по частям.

Пространство виртуальных адресов разделено на единичные блоки, называемые *страницами*. Соответствующие единицы в физической памяти называются *страничными блоками* (page frame). Страницы и их блоки имеют всегда одинаковый размер. В этом примере они равны 4 Кбайт, но в реальных системах использовались размеры от 512 байт до 64 Кбайт. Имея 64 Кбайт виртуального адресного пространства и 32 Кбайт физической памяти, мы получаем 16 виртуальных страниц и 8 страничных блоков. Передача данных между ОЗУ и диском всегда происходит в терминах страниц.

Когда программа пытается получить доступ к адресу 0, например, с помощью команды

```
MOVE REG,0
```

виртуальный адрес 0 передается диспетчеру памяти (MMU). Диспетчер памяти видит, что этот виртуальный адрес попадает на страницу 0 (от 0 до 4095), а та отображается на страничный блок 2 (адреса от 8192 до 12287). Диспетчер переводит виртуальный адрес 0 в физический адрес 8192 и выставляет последний на шину. Память ничего не знает о диспетчере памяти и видит просто запрос на чтение или запись слова по адресу 8192 и выполняет запрос. Таким образом, диспетчер памяти эффективно отображает все виртуальные адреса между 0 и 4095 на физические адреса от 8192 до 12287.

Точно так же инструкция

```
MOVE REG,8192
```

преобразуется в команду

```
MOVE REG,24576
```

поскольку виртуальный адрес 8192 находится на виртуальной странице 2, а эта страница отображается на физический страничный блок 6 (физические адреса от 24576 до 28671). В качестве третьего примера рассмотрим виртуальный адрес 20500, который адресует 20-й байт от начала виртуальной страницы 5 (виртуальные адреса от 20480 до 24575) и отображается на физический адрес 12288 + + 20 = 12308.

Сама по себе возможность отображения 16 виртуальных страниц на любой из восьми страничных блоков с помощью установки соответствующей карты в диспетчере памяти не решает проблемы, заключающейся в том, что размер виртуального адресного пространства больше физической памяти. Так как у нас есть только восемь физических страничных блоков, лишь восемь виртуальных страниц на рис. 4.8 воспроизводятся в физической памяти. Другие страницы, обозна-

ченные на рисунке крестиками, не отображаются. В аппаратном обеспечении страницы, физически присутствующие в памяти, отслеживаются с помощью *бита присутствия/отсутствия.*

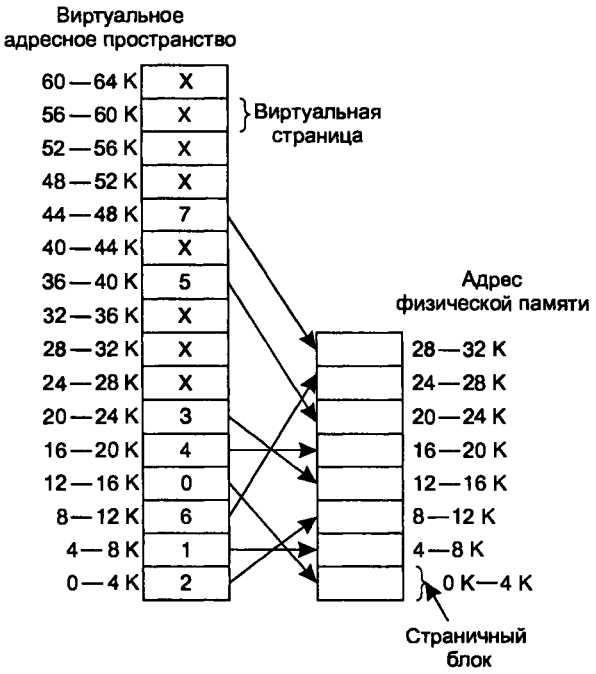


Рис. 4.8. Связь между виртуальными и физическими адресами, получаемая с помощью таблицы страниц

Что происходит, если программа пытается воспользоваться неотображаемой страницей, скажем, с помощью инструкции

```
MOVE REG,32780
```

которая обращается к байту 12 на виртуальной странице 8 (откладываемой с адреса 32768)? Диспетчер памяти замечает, что страница не отображается (обозначена крестиком на рисунке), и инициирует прерывание центрального процессора, передающее управление операционной системе. Такое прерывание называется *ошибкой из-за отсутствия страницы* или *страничным прерыванием* (page fault). Операционная система выбирает редко используемый страничный блок и записывает его содержимое на диск. Затем она считывает с диска страницу, на которую ссылается прерывание, в только что освободившийся блок, изменяет карту отображения и запускает заново прерванную команду.

Например, если операционная система решает удалить из оперативной памяти страничный блок 1, она загружает виртуальную страницу 8 по физическому адресу 4 К и производит два изменения в карте диспетчера памяти. Во-первых, отмечается содержимое виртуальной страницы 1 как неотображаемое для того,

чтобы перехватывать в будущем любые попытки обращения к виртуальным адресам между 4 К и 8 К. Затем заменяется крест в записи для виртуальной страницы 8 на номер 1, следовательно, когда прерванная команда будет выполняться заново, она спроецирует виртуальный адрес 32 780 на физический адрес 4108.

Теперь рассмотрим диспетчер памяти изнутри, чтобы увидеть, как он работает, и понять, почему мы выбрали размер страницы, являющийся степенью числа 2. На рис. 4.9 представлен пример виртуального адреса 8196 (0010000000000100 в двоичном виде), который отображается согласно карте менеджера памяти на рис. 4.8. Входной 16-разрядный виртуальный адрес разделяется на 4-разрядный номер страницы и 12 бит смещения. При четырех битах под номер страницы в нашей системе может существовать 16 страниц, а с 12 бит смещения мы можем адресоваться ко всем 4096 байтам внутри страницы.

Номер страницы используется в качестве индекса в *таблице страниц*, выдающей номер страничного блока, соответствующего виртуальной странице. Если бит присутствия/отсутствия равен 0, управление переходит к операционной системе. Если этот бит равен 1, номер страничного блока, найденный в таблице страниц, записывается в три старших бита выходного регистра, а 12 бит смещения копируются без изменения из входного виртуального адреса. Все вместе они составляют 15-разрядный физический адрес. Затем содержимое выходного регистра выставляется на шину памяти как адрес физической памяти.

4.3.2. Таблицы страниц

В теории отображение виртуальных адресов на физические происходит так, как мы только что описали. Виртуальный адрес делится на номер виртуальной страницы (старшие биты) и смещение (младшие биты). Например, при 16-разрядных адресах и размере страницы 4 Кбайт старшие 4 бита могут указывать одну из 16 виртуальных страниц, а нижние 12 бит могут определять байт смещения (от 0 до 4095) внутри выбранной страницы. Однако разбиение страницы на 3, 5 или какое-нибудь другое число битов также возможно. Разная дробность подразумевает различные размеры страниц.

Номер виртуальной страницы используется как индекс в таблице страниц для поиска записи этой страницы. По записи в таблице страниц находится номер физического блока страницы (если это имеет место). Данный номер присоединяется к старшим разрядам числа смещения, замещая собой номер виртуальной страницы и тем самым формируя физический адрес, который может быть послан в память.

Назначение таблицы страниц заключается в отображении виртуальных страниц на страничные блоки. Говоря математически, таблица страниц — это функция, имеющая в качестве аргумента номер виртуальной страницы и вырабатывающая в результате номер физического блока. На основе полученного результата поле виртуальной страницы в виртуальном адресе может быть заменено полем страничного блока, таким образом формируется физический адрес.

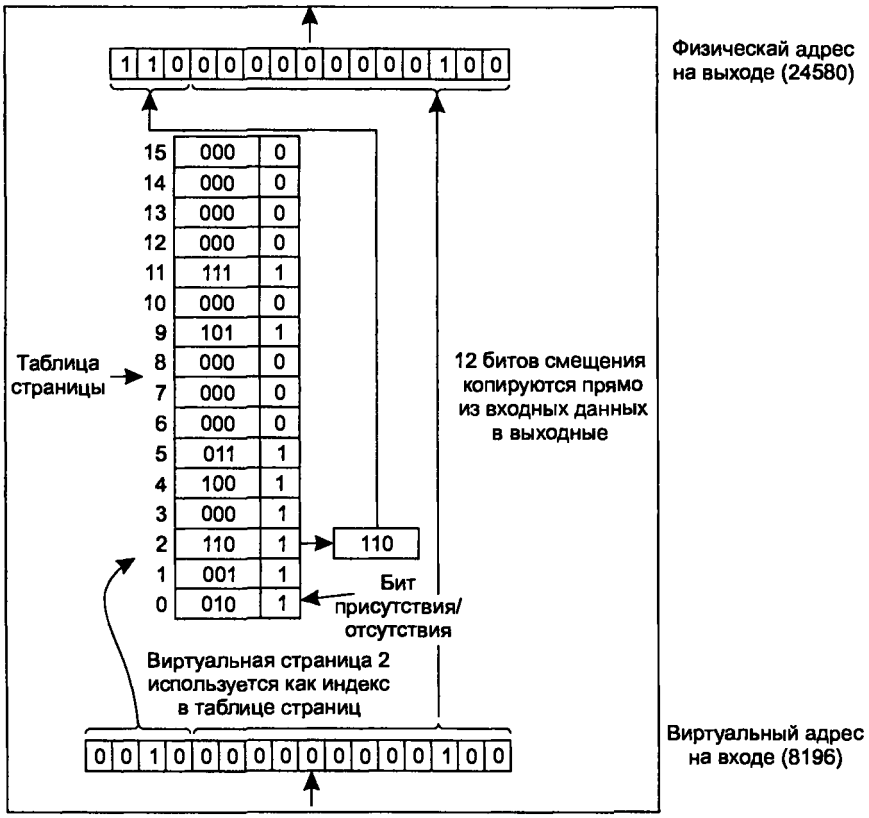


Рис. 4.9. Внутренняя операция диспетчера памяти в системе с шестнадцатью страницами размером 4 Кбайт

Несмотря на столь простое описание, нам придется столкнуться с двумя важными проблемами.

1. Таблица страниц может быть слишком большой.
2. Отображение должно быть быстрым.

Первое утверждение следует из того факта, что современным компьютерам сродни по крайней мере 32-разрядные виртуальные адреса. При размере страницы, скажем, 4 Кбайт, 32-разрядное адресное пространство будет состоять из 1 млн страниц, а 64-разрядное адресное пространство будет включать в себя намного больше страниц, чем то количество, с которым вы захотите иметь дело. При 1 млн страниц в виртуальном адресном пространстве таблица страниц должна состоять из 1 млн записей. И помните, что каждый процесс нуждается в своей собственной таблице страниц (так как у него есть свое собственное виртуальное адресное пространство).

Второе положение — это вывод из той реальности, что преобразование виртуальных адресов в физические должно быть выполнено для каждого обращения

к ячейке памяти. Типичная команда процессора часто включает в себя, помимо слова-опкода, также операнд(ы) памяти. В результате необходимо сделать 1, 2 или иногда больше обращений к таблице страниц в рамках отработки одной команды. Если выполнение команды занимает, скажем, 4 нс, то поиск в таблице страниц должен завершиться до истечения 1 нс, чтобы преобразование виртуальных адресов не стало главным узким местом системы.

Потребность в огромном, но при этом быстром страничном отображении накладывает существенные ограничения на способы построения компьютеров. Хотя проблема наиболее серьезно встает для старших моделей семейства, она также появляется и для младших моделей, когда стоимость и соотношение цена/производительность имеют критическое значение. В этом и следующих разделах мы рассмотрим устройство таблицы страниц в деталях и покажем несколько аппаратных решений, которые использовались в реальных компьютерах.

Простейшее конструкторское решение (по крайней мере, концептуально) заключается в поддержании таблицы страниц, состоящей из массива быстрых аппаратных регистров с одной записью для каждой виртуальной страницы, индексированного по номерам виртуальных страниц, как показано на рис. 4.9. Когда процесс запускается, операционная система загружает в регистры таблицу страниц процесса, данные берутся из копии, хранящейся в оперативной памяти. Во время выполнения процесса таблице страниц больше не нужно обращаться к памяти. Преимущество этого метода заключается в его простоте и отсутствии необходимости обращений к памяти во время преобразования адресов. Недостатком является его потенциально высокая стоимость (если таблица страниц велика). Необходимость загрузки полной таблицы в регистры при каждом контекстном переключении наносит ущерб производительности.

Другая крайность состоит в том, что таблица страниц целиком располагается в оперативной памяти. Тогда все необходимое оборудование — это единственный регистр, указывающий на начало таблицы страниц. Такая схема позволяет изменять карту памяти при контекстном переключении путем перезагрузки только одного регистра. Конечно, она имеет свой минус: во время выполнения каждой инструкции программы требуется одно или несколько обращений к памяти для чтения записей таблицы страниц. По этой причине данный метод редко используется в своем оригинальном виде, но ниже мы изучим несколько его разновидностей, имеющих намного более высокую производительность.

Многоуровневые таблицы страниц

Чтобы обойти проблему необходимости постоянного хранения в памяти огромных таблиц страниц, во многих компьютерах применяются многоуровневые таблицы страниц. Простой пример представлен на рис. 4.10. На рис. 4.10, *а* изображен 32-разрядный виртуальный адрес, разделенный на 10-разрядное поле PT1, 10-разрядное поле PT2 и 12-разрядное поле Offset (смещение). Так как под смещение отведено 12 бит, страницы имеют размер 4 Кбайт, и их всего 2^{20}.

Секрет метода многоуровневой организации заключается в том, чтобы не держать постоянно в памяти все таблицы страниц. В частности, те части, которые не нужны в данный момент, не должны быть резидентны. Предположим, например,

что процессу нужно 12 Мбайт: младшие 4 Мбайт памяти для текста программы, следующие 4 Мбайт для данных и старшие 4 Мбайт для стека. Между верхней границей данных и низом стека образуется гигантский свободный участок, который не используется.

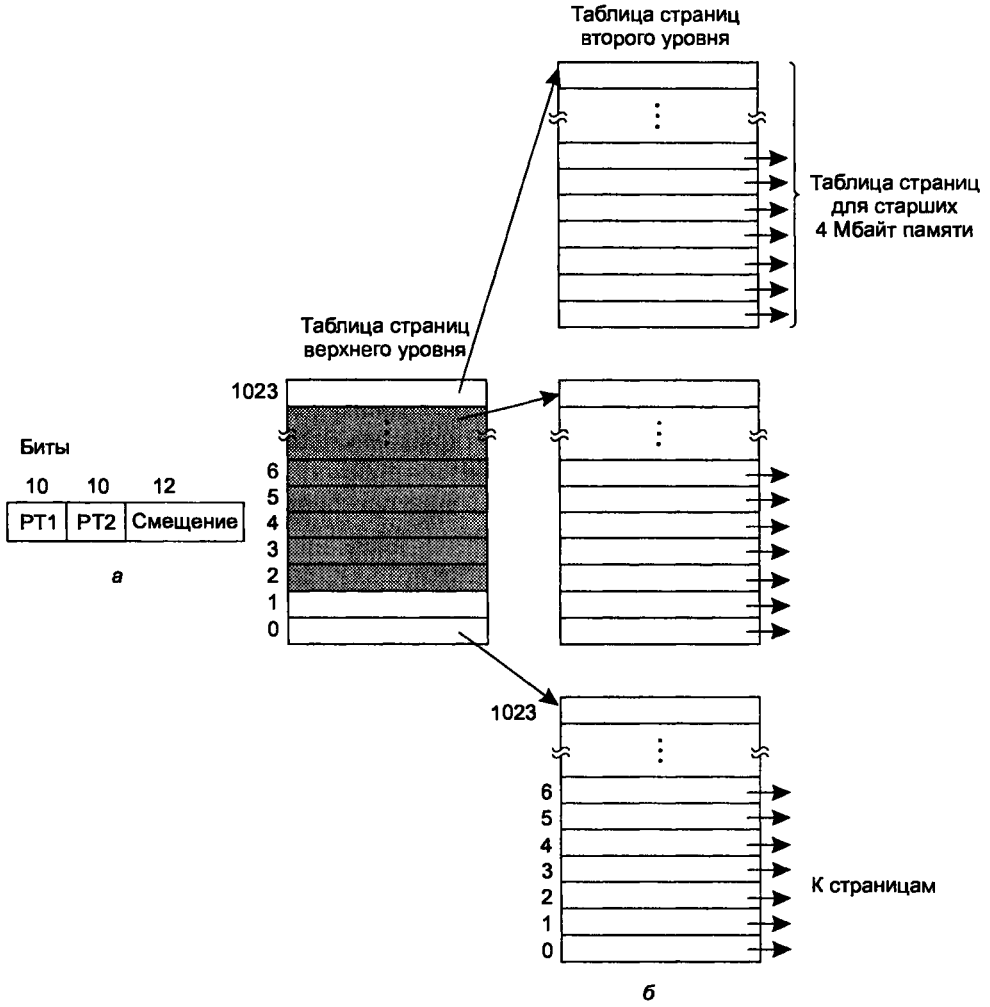


Рис. 4.10. *а* — разрядные адреса с полями двух таблиц страниц; *б* — двухуровневая таблица страниц

На рис. 4.10, *б* мы видим, как в данном примере работает двухуровневая таблица страниц. Слева находится таблица страниц верхнего уровня с 1024 записями, соответствующими 10-разрядному полю PT1. Когда виртуальный адрес предстает перед диспетчером памяти, тот сначала выделяет поле PT1 и использует его значение как индекс таблицы верхнего уровня. Каждая из этих 1024 записей

представляет 4 Мбайт, поскольку целое 4-гигабайтное (то есть 32-разрядное) виртуальное адресное пространство было нарезано на куски по 1024 байта.

Запись, место которой определяется по индексу в таблице страниц верхнего уровня, выдает адрес или номер страничного блока таблицы страниц второго уровня. Запись 0 в таблице страниц первого уровня указывает на таблицу страниц для текста программы, запись 1 указывает на таблицу страниц для данных, запись 1023 указывает на таблицу страниц для стека. Другие (заштрихованные) записи не задействованы. Поле PT2 теперь используется как индекс в выбранной таблице второго уровня для поиска номера страничного блока самой страницы.

В качестве примера рассмотрим 32-разрядный адрес 0x00403004 (4 206 596 в десятичном виде), который соответствует байту 12292 в данных. У этого виртуального адреса PT1 = 1, PT2 = 2 и Offset = 4. Менеджер памяти сначала по полю PT1, то есть по индексу в таблице страниц верхнего уровня, получает запись 1, которая соответствует адресам от 4 М до 8 М. Затем из поля PT2, по индексу из только что найденной таблицы второго уровня, извлекает запись 3, которая соответствует адресам от 12 288 до 16 383 внутри своего участка размером 4 М (то есть абсолютным адресам от 4 206 592 до 4 210 687). Эта запись содержит номер физического блока страницы, содержащей виртуальный адрес 0x00403004. Если данная страница не находится в памяти, бит присутствия/отсутствия в записи таблицы страниц будет равен нулю, что приведет к страничному прерыванию. Если страница в памяти, номер страничного блока, взятый из таблицы страниц второго уровня, присоединяется к смещению (4), образуя физический адрес. Этот адрес выставляется на шину и передается памяти.

Следует отметить одну интересную деталь на рис. 4.10. Хотя адресное пространство содержит больше миллиона страниц, фактически нужны только четыре таблицы: таблица верхнего уровня и таблицы нижнего уровня для памяти от 0 до 4 М, от 4 М до 8 М и для верхних 4 М. Битам присутствия/отсутствия для 1021 записи таблицы страниц верхнего уровня присвоено значение 0, что вызовет страничное прерывание при любом обращении к ним. Если это произойдет, операционная система заметит, что процесс пытается обратиться к области памяти, не предполагающей ссылок на нее, и предпримет соответствующее действие, например пошлет ему сигнал или уничтожит его. В описанном выше примере мы выбрали круглые значения для различных величин и размер поля PT1, равный размеру поля PT2, но в реальной практике, конечно, возможны другие цифры.

Система двухуровневой иерархии, показанная на рис. 4.10, может быть расширена для трех, четырех и больше уровней. Дополнительные слои дадут большую гибкость, но сомнительно, что следует усложнять систему больше, чем до трех уровней.

Теперь от структуры таблиц страниц в целом мы перейдем к описанию отдельного элемента записи таблицы. Точная структура элемента в значительной мере зависит от машины, но виды представленной информации примерно одни и те же. На рис. 4.11 мы привели образец записи в таблице страниц. Ее длина варьируется от компьютера к компьютеру, но 32 бита — это наиболее распространенный размер. Самым важным полем является *номер страничного блока.*

Прежде всего, задачей отображения страниц является определение данной величины. За этим полем следует бит присутствия/отсутствия. Если этот бит равен 1, запись имеет силу и может использоваться. Если он равен 0, виртуальная страница, которой соответствует эта запись, в данный момент отсутствует в памяти. Обращение к записи в таблице страниц, где индикатору присутствия/отсутствия присвоено нулевое значение, приводит к страничному прерыванию.

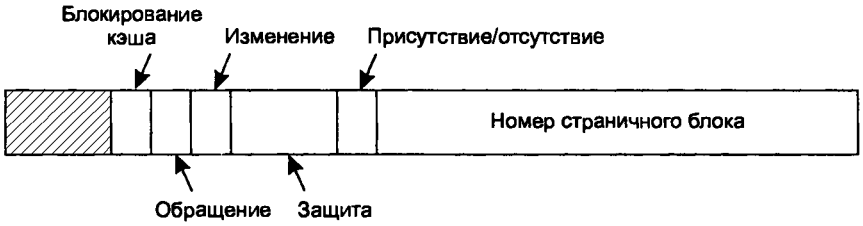


Рис. 4.11. Типичная запись в таблице страниц

Биты *защиты* говорят о том, какие разрешены виды доступа к этой странице. В простейшей форме это поле содержит один бит, равный 1 для чтения/записи и равный 0 только для чтения. Более сложные схемы имеют три бита, по одному для допуска каждой из операций чтения, записи и выполнения страницы.

Биты *изменения* и *обращения* отслеживают использование страницы. Когда страница записывается, аппаратура автоматически устанавливает бит изменения. Этот бит учитывается, когда операционная система решает освободить страничный блок. Если страница в нем была изменена (то есть она «грязная»), ее новая версия должна быть переписана на диск. Если она не была модифицирована (то есть страница «чистая»), ее можно просто удалить из памяти, так как все еще действительна копия на диске. Этот бит иногда называют *грязным битом*, так как он отражает состояние страницы.

Бит обращения устанавливается всякий раз, когда происходит обращение к странице для чтения или записи. Его значение помогает операционной системе при выборе страницы для удаления из памяти, когда случается страничное прерывание. Страницы, не использующиеся в данный момент, являются лучшими кандидатами, чем находящиеся в работе. Этот бит играет важную роль в нескольких алгоритмах перемещения страниц, которые мы изучим позже в текущей главе.

Наконец, последний бит позволяет запретить кэширование страницы. Данное свойство важно для страниц, отображающихся не на память, а на регистры устройств. Если операционная система находится в цикле ожидания ответа от некоторого устройства ввода/вывода, которому была только что отдана команда, существенно, чтобы аппаратура продолжала получать слово из устройства, а не использовало старую копию, хранимую кэш-памятью. При помощи этого бита кэширование можно отключить. Компьютеры, имеющие отдельное пространство адресов ввода/вывода и без отображения регистров ввода/вывода на память, не нуждаются в нем.

Заметим, что адрес места на диске, где хранится страница тогда, когда она не находится в памяти, не является частью таблицы страниц. Причина очень про-

ста. Таблица страниц содержит только ту информацию, которая нужна аппаратуре для перевода виртуального адреса в физический. Информация, необходимая операционной системе для обработки страничных прерываний, хранится в программных таблицах внутри самой ОС. Аппаратуре она не нужна.

4.3.3. Буферы быстрого преобразования адреса (TLB)

В большинстве схем со страничной организацией памяти таблицы страниц хранятся в памяти из-за их значительного размера. Потенциально такое устройство оказывает колоссальное влияние на производительность. Рассмотрим, например, команду процессора, копирующую содержимое одного регистра в другой. В отсутствие страничной организации памяти эта команда приводит только к одному обращению к памяти для выборки самой команды. Если же память организована постранично, потребуются дополнительные ссылки для доступа к таблице страниц. Так как скорость выполнения команд в основном ограничена скоростью, с которой центральный процессор выбирает команды и данные из памяти, необходимость двух обращений к таблице страниц на одну ссылку к памяти уменьшает производительность на 2/3. При таких условиях никто не стал бы внедрять этот метод.

Разработчики компьютеров многие годы размышляли об означенной проблеме и в результате придумали решение. Оно основано на наблюдении, что большинство программ склонно делать огромное количество обращений к небольшому количеству страниц, а не наоборот. То есть в таблице страниц лишь малая толика записей читается интенсивно, остальная часть едва ли вообще востребована.

В результате принятого решения компьютер снабжается небольшим аппаратным устройством, служащим для отображения виртуальных адресов в физические без прохода по таблице страниц. Параметры этого устройства, называемого буфером быстрого преобразования адреса (TLB — Translation Lookaside Buffer) или иногда *ассоциативной памятью*, представлены в табл. 4.1. Оно обычно находится внутри диспетчера памяти и поддерживает несколько записей. В нашем примере их восемь, но фактически записей редко бывает больше 64. Каждая запись содержит информацию об одной странице, а именно: номер виртуальной страницы, бит, устанавливаемый при изменении страницы, код защиты (разрешения на чтение/запись/выполнение) и номер физического страничного блока, в котором расположена эта страница. Эти поля однозначно соответствуют полям в таблице страниц. Еще один бит служит признаком того, действительна ли запись (то есть используется ли она в данный момент) или нет.

Пример, который мог бы сформировать TLB-буфер, представленный в таблице, — это циклический процесс, располагающийся в виртуальных страницах 19, 20 и 21. Соответственно, эти записи в табл. 4.1 имеют защитные коды для чтения и выполнения. Основные данные, используемые в текущий момент (скажем, обрабатываемый массив), находятся в страницах 129 и 130. Страница 140 содер-

жит индексы, требуемые для вычислений массива. И наконец, в страницах 860 и 861 находится стек.

Таблица 4.1. Буфер быстрого преобразования памяти для увеличения скорости страничной подкачки

Действительная запись	Виртуальная страница	Изменение	Защита	Страничный кадр
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Теперь рассмотрим, как же функционирует буфер быстрого преобразования адреса (TLB). Когда виртуальный адрес представляется диспетчером памяти для отображения, аппаратура сначала убеждается в том, что номер его виртуальной страницы присутствует в TLB, путем сравнения адреса со всеми записями одновременно (то есть параллельно). Если найдено имеющее силу совпадение и обращение не нарушает биты защиты, страничный блок берется прямо из TLB, без перехода к таблице страниц. Если номер виртуальной страницы присутствует в буфере, но инструкция пытается записать что-то на страницу, доступную только для чтения, формируется ошибка защиты точно так же, как это происходило бы из самой таблицы страниц.

Интересная ситуация получается, если номер виртуальной страницы не находится в буфере быстрого преобразования адреса. Диспетчер памяти обнаруживает этот факт и выполняет обычный поиск в таблице страниц. Затем он удаляет одну из записей из буфера и заменяет ее только что найденной записью из таблицы страниц. Таким образом, если страница снова вскоре будет затребована, во второй раз поиск окажется успешным, а не неудачным. Когда запись удаляется из буфера быстрого преобразования адреса, бит изменения копируется в запись таблицы страниц в памяти. Другие значения уже находятся там. Когда буфер загружается из таблицы страниц, все поля берутся из памяти.

Программное управление TLB

До сих пор мы предполагали, что каждая машина со страничной виртуальной памятью имеет таблицы страниц, распознаваемые аппаратным обеспечением и буфером быстрого преобразования адреса. При таком устройстве за управление TLB и обработку его ошибок полностью отвечает аппаратура менеджера памяти

(MMU). Передача управления операционной системе происходит только тогда, когда страница отсутствует в памяти.

В прошлом это допущение было справедливо. Однако многие современные RISC-компьютеры, включая машины SPARC, MIPS, Alpha и HP PA, выполняют почти все страничное управление программно. На этих машинах записи TLB явно загружаются операционной системой. Когда поиск в ассоциативной памяти заканчивается неудачей (промах), диспетчер памяти, вместо того чтобы переключаться на таблицу страниц для поиска и выбора необходимой страницы, формирует ошибку TLB и передает проблему в руки операционной системы. Система должна найти страницу, удалить запись из буфера, ввести новую запись и перезапустить прерванную инструкцию. И конечно, все это должно быть сделано при помощи небольшого числа команд, поскольку промахи в буфере быстрого преобразования адреса случаются намного чаще, чем ошибки из-за отсутствия страниц.

Достаточно удивительно то, что если буфер имеет небольшой размер (скажем, 64 записи) с целью минимизации промахов, программное управление буфером, оказывается, является приемлемо результативным. Главная выгода здесь заключается в намного более простом устройстве диспетчера памяти, что освобождает достаточное количество пространства в микросхеме процессора для кэша и других устройств, способных повысить производительность. Программное управление буфером быстрого преобразования адреса обсуждается в [84].

Для подъема производительности на компьютерах, программно управляющих TLB, разрабатывались различные стратегии поведения. Один подход состоит в попытке уменьшить как частоту неудачного поиска в буфере, так и его стоимость, когда он все-таки случается [4]. Чтобы уменьшить вероятность неудачного поиска в TLB, иногда операционная система может «интуитивно» вычислить, какие страницы, возможно, будут использоваться следующими, и предварительно загрузить записи для них в буфер. Например, когда клиентский процесс посылает сообщение серверному процессу на той же самой машине, очень вероятно, что сервер вскоре должен будет начать работу. Зная это, система может также проверить, где находятся страницы кода сервера, данных и стека, пока прерывание обрабатывается, чтобы осуществить вызов send, и преобразовать их адреса из виртуальных в физические до того, как они смогут стать причиной ошибки TLB-буфера.

Обычный путь обработки промахов TLB, аппаратно или программно, — это переход в таблицу страниц и выполнение операции индексации, с целью определить положение страницы, к которой происходит обращение. При программной реализации возникает проблема, суть которой в том, что страницы, содержащиеся в таблице страниц, могут отсутствовать в буфере быстрого преобразования адреса, что вызовет дополнительные ошибки TLB во время обработки. Их количество можно уменьшить, поддерживая большой (например, 4 Кбайт) программный кэш записей TLB с фиксированным расположением в памяти. Если сначала проверять программный кэш, операционная система способна в значительной степени снизить количество неудачных поисков в TLB.

4.3.4. Инвертированные таблицы страниц

Традиционные таблицы страниц, тип которых мы описывали до сих пор, требуют по одной записи на каждую виртуальную страницу, так как они индексируются по номеру этой страницы. Если адресное пространство состоит из 2^{32} байт с размером страницы 4096 байт, тогда в таблице страниц должно быть больше миллиона записей. Отсюда, таблица страниц будет занимать минимум 4 Мбайт. В достаточно больших системах это, вероятно, осуществимо.

Однако, поскольку 64-разрядные компьютеры встречаются все чаще, ситуация радикально меняется. Если теперь адресное пространство увеличилось до 2^{64} байт с размером страницы 4 Кбайт, нам требуется таблица страниц с 2^{52} записями. Если каждая запись равна 8 байтам, таблица займет больше 30 Тбайт. Выделение 30 Тбайт только для таблицы страниц нереально сейчас и не будет реальным когда-либо в будущем. Следовательно, для 64-разрядного страничного виртуального пространства необходимо другое решение.

Одним из таких решений является *инвертированная таблица страниц*. В этой модели таблица содержит по одной записи на страничный блок в реальной памяти, а не на страницу в виртуальном адресном пространстве. Например, при 64-разрядных виртуальных адресах, размере страниц 4 Кбайт и 256 Мбайт оперативной памяти инвертированная таблица страниц потребует всего лишь 65 536 записей. Каждая запись отслеживает, что (процесс, виртуальная страница) расположено в данном страничном блоке.

Хотя инвертированные таблицы страниц экономят значительное количество места, по крайней мере, когда виртуальное адресное пространство намного превышает физическую память, они имеют серьезный недостаток: перевод виртуального адреса в физический значительно усложняется. Когда процесс n обращается к виртуальной странице p, аппаратное обеспечение не может больше найти физическую страницу, отталкиваясь от номера p как от индекса в таблице страниц. Вместо этого оно должно производить поиск записи (n, p) во всей инвертированной таблице страниц. Более того, этот поиск должен выполняться при каждом обращении к памяти, а не только при страничном прерывании. Операция поиска в таблице размером 64 К при каждой ссылке к памяти вовсе не увеличит скорость вашей машины.

Выйти из этого затруднительного положения позволяет буфер быстрого преобразования адреса. Если TLB может содержать все часто используемые страницы, трансляция адреса будет происходить так же быстро, как и с обычными таблицами страниц. Но при промахе в TLB поиск в инвертированной таблице страниц должен выполняться программно. Один из возможных способов усовершенствовать его — поддерживать хеширование виртуальных адресов. Все виртуальные страницы, находящиеся в данный момент в памяти и имеющие одинаковое значение хеш-функции, сцепляются друг с другом. Если хеш-таблица состоит из такого же количества ячеек, сколько есть в машине физических страниц, средняя цепочка будет длиной только в одну запись, что значительно увеличит скорость отображения адресов. Как только найден номер страничного блока, новая пара (виртуальная, физическая) помещается в TLB.

Инвертированные таблицы страниц в настоящее время используются на некоторых рабочих станциях компаний IBM и Hewlett–Packard и будут встречаться все чаще, так как 64-разрядные машины получают все более широкое распространение.

Некоторые другие методы управления виртуальной памятью большого размера можно найти в [45, 78, 79].

4.4. Алгоритмы замещения страниц

Когда происходит страничное прерывание, операционная система должна выбрать страницу для удаления из памяти, чтобы освободить место для страницы, которую нужно туда перенести. Если удаляемая страница была изменена за время своего присутствия в памяти, ее необходимо переписать на диск, чтобы обновить копию, хранящуюся там. Однако если страница не была модифицирована (например, она содержит текст программы), копия на диске уже является самой новой и ее не надо переписывать. Тогда страница, которую надо прочитать, просто считывается поверх выгружаемой.

Хотя, в принципе, можно при каждом страничном прерывании выбирать случайную страницу для удаления, производительность системы заметно повышается, когда предпочтение отдается редко используемой странице. Если выгружается страница, обращения к которой происходят часто, велика вероятность, что вскоре опять потребуется ее возврат в память, что даст в результате дополнительные издержки. Теме разработки алгоритмов замены страницы было посвящено много работ, как теоретических, так и экспериментальных. Ниже мы опишем некоторые из наиболее важных алгоритмов.

4.4.1. Оптимальное замещение страниц

Наилучший из возможных алгоритмов замещения страниц легко описать, но невозможно реализовать. Он действует так. В тот момент, когда происходит страничное прерывание, в памяти находится некоторый набор страниц. К одной из этих страниц будет обращаться следующая команда процессора (к странице, содержащей требуемую команду). На другие страницы, возможно, не будет ссылок в течение следующих 10, 100 или даже 1000 команд. Каждая страница может быть помечена количеством команд, которые будут выполняться перед первым к ней обращением.

Оптимальный страничный алгоритм просто сообщает, что должна быть выгружена страница с наибольшей меткой. Если одна страница не будет использоваться в течение 8 млн команд, а другая — в течение 6 млн команд, удаление первой отодвинет в будущее на возможно максимальный срок страничное прерывание, которое вернет ее назад. Компьютеры, подобно людям, пытаются отложить неприятные события настолько, насколько это возможно.

С этим алгоритмом связана только одна проблема: он невыполним. В момент страничного прерывания операционная система не имеет возможности узнать,

когда произойдет следующее обращение к каждой странице. (Мы рассматривали аналогичную ситуацию раньше, когда обсуждали алгоритм планирования «кратчайшая задача — первая»: как система может сказать, какая из задач самая короткая?) Тем не менее, выполняя программу на модели и следя за всеми обращениями к страницам, оптимальную замену можно осуществить при *втором* запуске, опираясь на информацию о ссылках на страницы, собранную во время *первого* запуска.

В этом случае можно сравнивать производительность реализуемых алгоритмов с наилучшим. Если операционная система добивается производительности, скажем, всего на один процент ниже, чем при работе оптимального алгоритма, усилия, потраченные на поиск лучшего алгоритма, повысят продуктивность схемы максимум на 1 %.

Чтобы избежать возможных недоразумений, следует прояснить, что полученный протокол обращений к страницам относится только к одной хорошо спланированной программе и, кроме того, к определенным входным данным. Таким образом, алгоритм замещения страниц, выведенный из него, будет характерен только для этой программы с именно этими входными данными. Хотя такой метод полезен для оценки алгоритмов замещения страниц, он не используется в практических системах. Ниже мы изучим алгоритмы, которые *являются* применимыми в реальных ситуациях.

4.4.2. Алгоритм NRU — не использовавшаяся в последнее время страница

Чтобы дать возможность операционной системе собирать полезные статистические данные о том, какие страницы используются, а какие — нет, большинство компьютеров с виртуальной памятью поддерживают два статусных бита, связанных с каждой страницей. Бит R (от Referenced — обращения) устанавливается всякий раз, когда происходит обращение к странице (чтение или запись). Бит M (от Modified — изменение) устанавливается, когда страница записывается (то есть изменяется). Биты содержатся в каждом элементе таблицы страниц, как показано на рис. 4.11. Важно реализовать обновление этих битов при каждом обращении к памяти, поэтому необходимо, чтобы они задавались аппаратно. Если однажды бит был установлен в 1, то он остается равным 1 до тех пор, пока операционная система программно не вернет его в состояние 0.

Если аппаратное обеспечение не поддерживает эти биты, их можно смоделировать следующим образом. Когда процесс запускается, все его записи в таблице страниц помечаются как отсутствующие в памяти. Как только происходит обращение к странице, происходит страничное прерывание. Затем операционная система устанавливает бит R (в своих внутренних таблицах); изменяет запись в таблице страниц так, чтобы она указывала на корректную страницу с режимом «только для чтения», и перезапускает команду. Если страница позднее записывается, происходит другое страничное прерывание, позволяющее операционной системе установить бит M и изменить состояние страницы на «чтение/запись».

Биты R и M могут использоваться для построения простого алгоритма замещения страниц, описанного ниже. Когда процесс запускается, оба страничных бита для всех его страниц операционной системой сброшены в 0. Периодически (например, при каждом прерывании по таймеру) бит R очищается с целью отличить страницы, к которым давно не было обращения, от тех, на которые ссылки были.

При возбуждении страничного прерывания операционная система проверяет все страницы и делит их на четыре категории на основании текущих значений битов R и M:

- класс 0: не было обращений и изменений;
- класс 1: не было обращений, страница изменена;
- класс 2: было обращение, страница не изменена;
- класс 3: произошло и обращение, и изменение.

Хотя класс 1 на первый взгляд кажется невозможным, такое случается, когда у страницы из класса 3 бит R сбрасывается во время прерывания по таймеру. Прерывания по таймеру не затирают бит M, поскольку эта информация необходима для того, чтобы знать, нужно ли переписывать страницу на диске или нет. Поэтому если бит R устанавливается на ноль, а M остается нетронутым, страница попадает в класс 1.

Алгоритм *NRU* (Not Recently Used — не использовавшийся в последнее время) удаляет страницу с помощью случайного поиска в непустом классе с наименьшем номером. Подразумевается, что лучше выгрузить измененную страницу, к которой не было обращений, по крайней мере в течение одного тика системных часов (обычно 20 мс), чем стереть часто используемую. Привлекательность алгоритма NRU заключается в том, что он легок для понимания, умеренно сложен в реализации и дает производительность, которая, конечно, не оптимальна, но бывает вполне достаточной.

4.4.3. Алгоритм FIFO: первым прибыл — первым обслужен

Другим требующим небольших издержек алгоритмом является *FIFO* (First-In, First-Out — «первым прибыл — первым обслужен»). Чтобы проиллюстрировать его работу, рассмотрим универсам, на полках которого можно выставить ровно k различных продуктов. Он предлагает новую удобную пищу: растворимый, «глубоко замороженный», экологически чистый йогурт, который можно мгновенно приготовить в микроволновой печи. Покупатели тут же обратили внимание на этот продукт, и наш ограниченный в размерах супермаркет, для того чтобы продавать новинку, должен избавиться от одного из старых товаров.

Один из вариантов решения состоит в том, чтобы найти продукт, который супермаркет продает дольше всего (то есть что-нибудь, что завезли на реализацию 120 лет назад), и освободить от него магазин на том основании, что им никто больше не интересуется. В действительности хранится номенклатура всех прода-

ваемых в данный момент в супермаркете товаров, упорядоченная по времени их появления. Каждый новый продукт помещается в конец перечня, а из начала списка удаляется одно старое наименование.

Та же самая идея пригодна в качестве алгоритма замещения страниц. Операционная система поддерживает список всех страниц, находящихся в данный момент в памяти, в котором первая страница является старейшей, а страницы в хвосте списка попали в него совсем недавно. Когда происходит страничное прерывание, выгружается из памяти страница в голове списка, а новая страница добавляется в его конец. Применительно к тому же магазину: если удалить воск для усов согласно алгоритму FIFO, мы также можем лишиться и муки, соли или масла. Та же проблема встает и в отношении компьютеров. По этой причине алгоритм FIFO редко используется в своей исходной форме.

4.4.4. Алгоритм «вторая попытка»

В простейшем варианте алгоритма FIFO, который позволяет избежать проблемы вытеснения из памяти часто используемых страниц, у самой старейшей страницы изучается бит R. Если он равен 0, значит, страница не только находится в памяти долго, она вдобавок еще и не используется, поэтому немедленно заменяется новой. Если же бит R равен 1, ему присваивается значение 0, страница переносится в конец списка, а время ее загрузки обновляется, то есть считается, что страница только что попала в память. Затем процедура продолжается.

Работа этого алгоритма, называемого *«второй попыткой»* (second chance), показана на рис. 4.12, *а*. Здесь изображены страницы от A до H, хранящиеся в однонаправленном списке и отсортированные по времени их поступления в память. Числа над страницами обозначают их время загрузки в память.

Предположим, что в момент времени 20 происходит страничное прерывание. Самой старшей страницей является страница A, она была загружена в память в момент 0, когда начал работу процесс. Если бит R страницы A равен 0, она выгружается из памяти или путем записи на диск (если страница «грязная»), или просто удаляется (если она «чистая»). Во втором случае, если бит R равен 1, страница A передвигается в конец списка, а ее «загрузочное время» принимает текущее значение (20). При этом бит R сбрасывается. Поиск подходящей страницы продолжается; следующей проверяется страница B.

Алгоритм «вторая попытка» ищет в списке самую старую страницу, к которой не было обращений в предыдущем временном интервале. Если же происходили ссылки на все страницы, то «вторая попытка» превращается в обычный алгоритм FIFO. Представьте, что у всех страниц на рис. 4.12, *а* бит R равен 1. Одну за другой передвигает операционная система страницы в конец списка, очищая бит R каждый раз, когда она перемещает страницу в хвост. Наконец, она вернется к странице A, но теперь уже ее биту R присвоено значение 0. В этот момент страница A выгружается из памяти. Таким образом, алгоритм всегда успешно завершает свою работу.

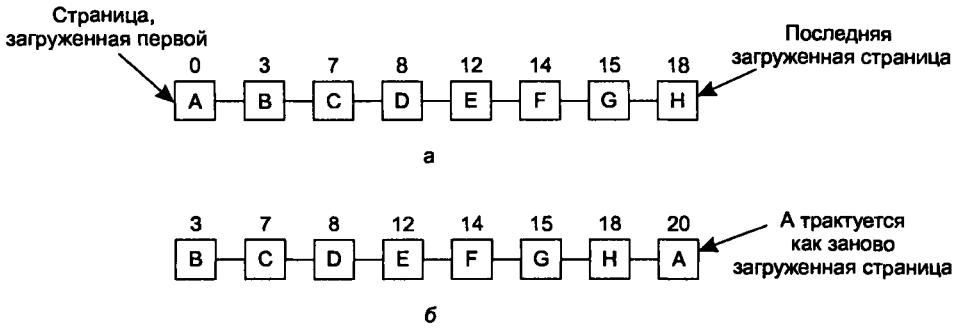


Рис. 4.12. Действие алгоритма «вторая попытка»: *а* — страницы, отсортированные в порядке очереди (FIFO); *б* — список страниц, если страничное прерывание произошло во время 20, а страница A имеет бит R, равный 0

4.4.5. Алгоритм «часы»

Хотя алгоритм дополнительного шанса является корректным, он слишком неэффективен, так как постоянно тасует страницы по списку. Поэтому лучше хранить все страничные блоки в кольцевом списке в форме часов, как показано на рис. 4.13. Стрелка указывает на старейшую страницу.

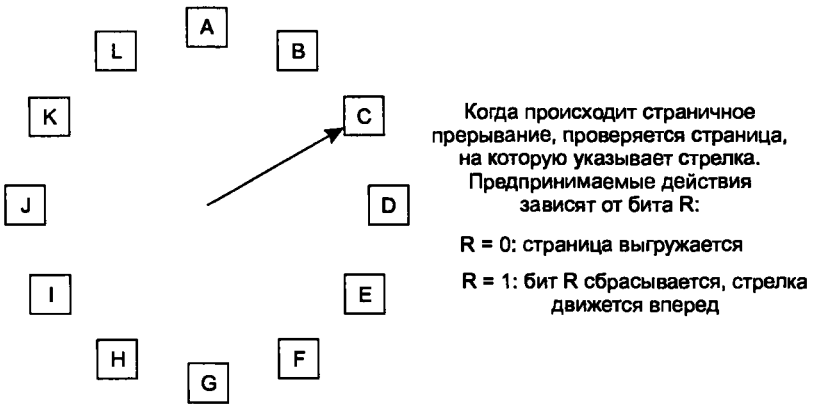


Рис. 4.13. Алгоритм замещения страниц «часы»

Когда происходит страничное прерывание, проверяется та страница, на которую направлена стрелка. Если ее бит R равен 0, страница выгружается, на ее место в часовой круг встает новая страница, а стрелка сдвигается вперед на одну позицию. Если бит R равен 1, он сбрасывается, стрелка перемещается к следующей странице. Этот процесс повторяется до тех пор, пока не находится та страница, у которой бит R = 0. Не удивительно, что алгоритм называется *«часами»*. Он отличается от алгоритма «вторая попытка» только своей реализацией.

4.4.6. Алгоритм LRU — страница, не использовавшаяся дольше всего

В основе этой неплохой аппроксимации оптимального алгоритма лежит наблюдение, что страницы, к которым наблюдалось многократное обращение в нескольких последних командах, вероятно, также будут часто востребованы в следующих инструкциях. И наоборот, можно полагать, что страницы, к которым ранее не возникало обращений, не будут нужны в течение долгого времени. Эта идея привела к следующему реализуемому алгоритму: когда происходит страничное прерывание, выгружается из памяти страница, которая не использовалась дольше всего. Такая стратегия замещения страниц называется *LRU* (Least Recently Used).

Хотя алгоритм LRU теоретически реализуем, он не является дешевым. Для полной реализации алгоритма LRU необходимо поддерживать список всех содержащихся в памяти страниц, такой, где последняя использовавшаяся страница находится в начале списка, а та, к которой дольше всего не было обращений, — в конце. Сложность заключается в том, что список должен обновляться при каждом обращении к памяти. Поиск страницы, ее удаление, а затем вставка в начало списка — это операции, поглощающие очень много времени, даже если они выполняются аппаратно (если предположить, что необходимое оборудование можно сконструировать).

Однако существуют другие способы реализации алгоритма LRU с помощью специального оборудования. Для первого метода требуется оснащение компьютера 64-разрядным аппаратным счетчиком C, который автоматически инкрементируется после каждой команды. Кроме того, каждая запись в таблице страниц должна иметь поле, достаточно большое для хранения значения счетчика. После каждого обращения к памяти текущая величина счетчика C запоминается в записи таблицы, соответствующей той странице, к которой произошла ссылка. А если возникает страничное прерывание, операционная система проверяет все значения счетчиков в таблице страниц и ищет наименьшее. Эта страница является не использовавшейся дольше всего.

Теперь рассмотрим второй вариант аппаратной реализации алгоритма LRU. На машине с n страничными блоками оборудование LRU может поддерживать матрицу $n \times n$ бит, изначально равных нулю. Всякий раз при доступе к страничному блоку k аппаратура сначала присваивает всем битам строки k значение 1, затем приравнивает нулю все биты столбца k. В любой момент времени строка, двоичное значение которой наименьшее, является не использовавшейся дольше всего. Работа этого алгоритма продемонстрирована на рис. 4.14, где рассматриваются четыре страничных блока и следующий порядок обращения к страницам:

0 1 2 3 2 1 0 3 2 3

После ссылки на страницу 0 мы получаем ситуацию, показанную на рис. 4.14, *а*; после обращения к странице 1 — рис. случай 4.14, *б* и т. д.

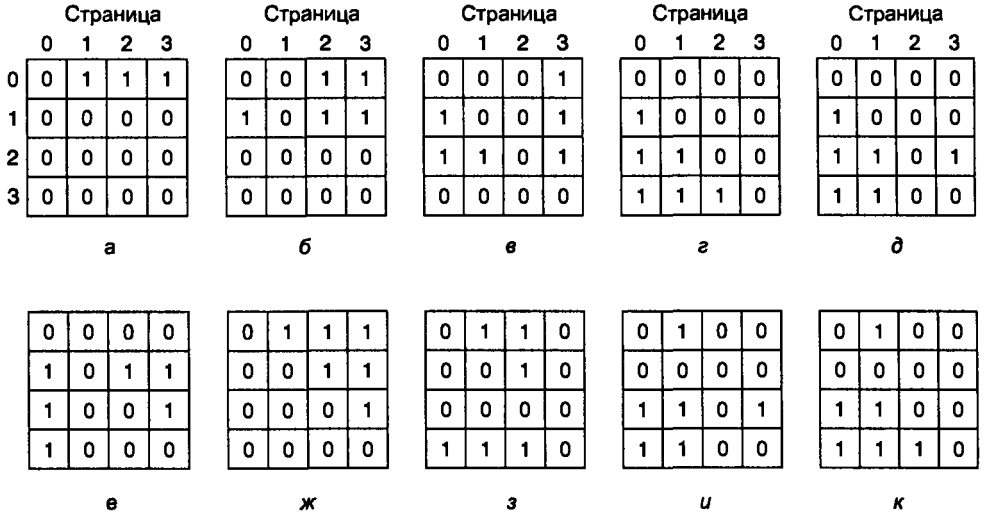


Рис. 4.14. Алгоритм LRU с привлечением матрицы. Обращения к страницам происходят в последовательности: 0, 1, 2, 3, 2, 1, 0, 3, 2, 3

4.4.7. Программное моделирование алгоритма LRU

Хотя оба описанных выше алгоритма LRU в принципе осуществимы, очень мало (если вообще такие есть) машин оснащено подобным оборудованием, в силу чего разработчики операционных систем для компьютеров, не имеющих такой аппаратуры, редко используют эти алгоритмы. Вместо них требуется программно реализуемое решение. Одна из разновидностей схемы LRU называется алгоритмом *NFU* (Not Frequently Used — редко использовавшаяся страница). Для него необходим программный счетчик, связанный с каждой страницей в памяти, изначально равный нулю. Во время каждого прерывания по таймеру операционная система исследует все страницы в памяти. Бит R каждой страницы (он равен 0 или 1) прибавляется к счетчику. В сущности, счетчики пытаются отследить, как часто имело место обращение к каждой странице. При страничном прерывании для замещения выбирается страница с наименьшим значением счетчика.

Основная проблема, возникающая при работе с алгоритмом NFU, заключается в том, что он никогда ничего не забывает. Например, в многопроходном компиляторе страницы, которые часто обрабатывались во время первого прохода, могут все еще иметь высокое значение счетчика на дальнейших проходах. Фактически, если случается так, что первый проход занимает самое долгое время выполнения из всех, страницы, содержащие программный код для следующих проходов, могут всегда иметь более низкое значение счетчика, чем страницы первого прохода. Следовательно, операционная система удалит полезные страницы вместо тех, которые больше не нужны.

К счастью, небольшая доработка алгоритма NFU делает его способным моделировать алгоритм LRU достаточно хорошо. Изменение сводится к двум модификациям. Во-первых, каждый счетчик сдвигается вправо на один разряд перед прибавлением бита R. Во-вторых, бит R вдвигается в крайний слева, а не в крайний справа разряд счетчика.

На рис. 4.15 продемонстрировано, как работает видоизмененный алгоритм, известный под названием *«старения»* (aging). Предположим, что после первого тика часов биты R для страниц от 0 до 5 имеют значения 1, 0, 1, 0, 1, 1 соответственно (у страницы 0 бит R равен 1, у страницы 1 — R = 0, у страницы 2 — R = 1 и т. д.). Другими словами, между тиком 0 и тиком 1 произошло обращение к страницам 0, 2, 4 и 5, их биты R приняли значение 1, остальные сохранили значение 0. После того как шесть соответствующих счетчиков сдвинулись на разряд и бит R занял крайнюю слева позицию, счетчики получили значения, показанные на рис. 4.15, *а*. Остальные четыре колонки рисунка изображают шесть счетчиков после следующих четырех тиков часов.

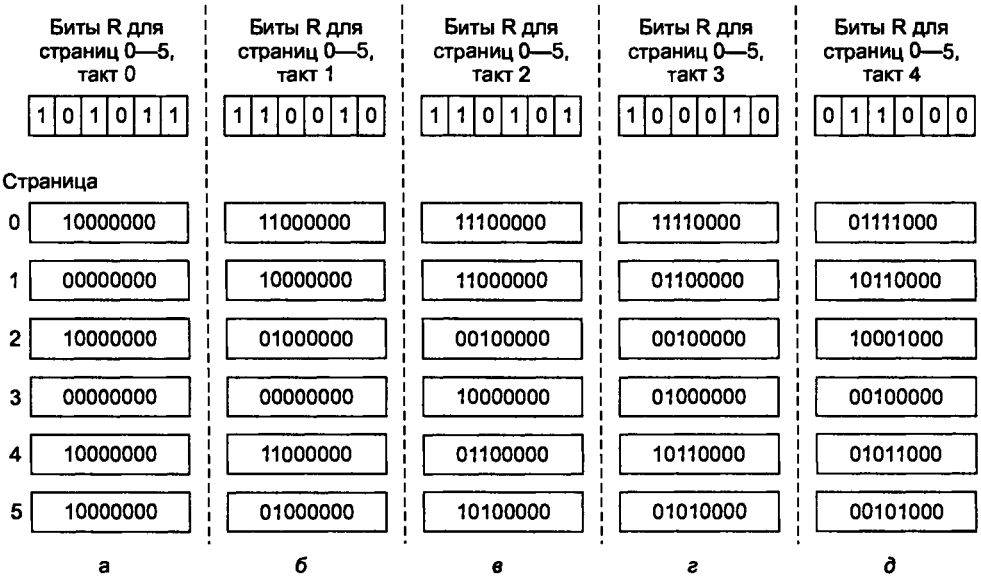


Рис. 4.15. Алгоритм старения программно моделирует алгоритм LRU. Здесь изображены шесть страниц после пяти тиков часов от (*а*) до (*д*)

Когда происходит страничное прерывание, удаляется та страница, счетчик которой имеет наименьшую величину. Ясно, что счетчик страницы, к которой не было обращений, скажем, за четыре тика, будет начинаться с четырех нулей и, таким образом, иметь более низкое значение, чем счетчик страницы, на которую не ссылались в течение только трех тиков часов.

Эта схема отличается от алгоритма LRU в двух случаях. Рассмотрим страницы 3 и 5 на рис. 4.15, *д*. Ни к одной из них не было обращений за последние

два тика, к обеим было обращение за предшествующий тик. Следуя алгоритму LRU, при удалении страницы из памяти мы должны выбрать одну из двух. Проблема в том, что мы не знаем, к какой из них позже имелось обращение в интервале времени между тиками 1 и 2. Записывая только один бит за промежуток времени, мы теряем возможность отличить более ранние от более поздних обращений в этом интервале времени. Все, что мы можем сделать, — это выгрузить страницу 3, так как к странице 5 также обращались двумя тиками раньше, а к странице 3 — нет.

Второе отличие между алгоритмами LRU и «старения» заключается в том, что в последнем счетчик имеет конечное число разрядов, например 8. Предположим, что каждая из двух страниц получила нулевое значение своего счетчика. В данной ситуации мы лишь случайным образом можем выбрать одну из них. На самом деле не исключено, что к одной странице в последний раз обращались 9 тиков назад, а к другой — 1000 тиков назад. И мы не имеем возможности увидеть это. На практике, однако, обычно достаточно 8 бит при тике системных часов около 20 мс. Если к странице не обращались в течение 160 мс, очень вероятно, что она не важна.

4.5. Разработка систем со страничной организацией памяти

В предыдущих разделах мы объяснили, как работает подкачка по страницам, представили несколько основных алгоритмов замещения страниц и показали, как их моделировать. Но знания голой механики недостаточно. Чтобы разработать хорошо работающую систему, вы должны вникнуть во все намного глубже. Разница такая же, как между человеком, который видел, как ходят ладья, конь, слон и другие шахматные фигуры, и хорошим шахматистом. Ниже мы рассмотрим другие вопросы, которые должны принимать во внимание разработчики операционных систем для того, чтобы получить достойную производительность системы со страничной организацией памяти.

4.5.1. Модель «рабочий набор»

В простейшей схеме страничной подкачки в момент запуска процессов нужные им страницы отсутствуют в памяти. Как только центральный процессор пытается выбрать первую команду, он получает страничное прерывание, побуждающее операционную систему перенести в память страницу, содержащую первую инструкцию. Обычно следом быстро происходят страничные прерывания для глобальных переменных и стека. Через некоторое время в памяти скапливается большинство необходимых процессу страниц, и он приступает к работе с относительно небольшим количеством ошибок из-за отсутствия страниц. Этот метод называется *замещением страниц по запросу* (demand paging), так как страницы загружаются в память по требованию, а не заранее.

Конечно, достаточно легко написать тестовую программу, систематически читающую все страницы в огромном адресном пространстве, что сопровождается таким количеством страничных прерываний, что будет не хватать памяти для их обработки. К счастью, большинство процессов не работают таким образом. Они характеризуются *локальностью обращений*, означающей, что во время выполнения любой своей фазы процесс имеет обращения только к сравнительно небольшой части собственных страниц. Многопроходный компилятор, например, обращается только к части от общего количества страниц на каждом проходе.

Множество страниц, которое процесс использует в данный момент, называется *рабочим набором* [22]. Если рабочий набор целиком находится в памяти, процесс будет выполняться, не вызывая большого количества ошибок, до тех пор пока он не перейдет к другой фазе выполнения (то есть к следующему проходу компилятора). Если доступная память слишком мала для того, чтобы содержать полный рабочий набор, процесс инициирует много страничных прерываний и будет работать медленнее, так как выполнение инструкции занимает несколько наносекунд, а чтение страницы с диска обычно требует 10 мс. При скорости одна или две команды на 10 мс для завершения программы понадобятся века. Говорят, что программа, вызывающая страничное прерывание каждые несколько команд, *пробуксовывает* (thrashing) [23].

В многозадачных системах процессы часто перемещаются на диск (то есть все их страницы удаляются из памяти), с целью позволить другим процессам получить доступ к центральному процессору. Возникает вопрос, что делать, когда процесс снова загружается в память. С формальной точки зрения делать ничего не нужно. Процесс будет вызывать одно за другим страничные прерывания до тех пор, пока не загрузится в память весь его рабочий набор. Проблема в том, что - наличие 20, 100 или даже 1000 страничных прерываний при каждой загрузке процесса сильно замедляет систему, и, кроме того, тратится впустую значительное количество времени центрального процессора, до нескольких миллисекунд, столько, сколько требует обработка страничного прерывания операционной системой.

Поэтому многие системы со страничной организацией пытаются отслеживать рабочий набор каждого процесса и обеспечивают его нахождение в памяти еще до запуска процесса. Такой подход носит название *модели рабочего набора* [24]. Он разработан для того, чтобы значительно снизить процент страничных прерываний. Загрузка страниц *перед* тем, как разрешить процессу работать, также называется *опережающей подкачкой страниц* (prepaging). Заметьте, что рабочий набор изменяется с течением времени.

Для реализации модели рабочего набора необходимо, чтобы операционная система отслеживала, какие страницы в нем находятся. Один из способов получить такую информацию — использовать описанный выше алгоритм старения. Пусть установленный старший бит счетчика у страницы означает, что она входит в рабочий набор. Если в течение n последовательных тактов часов к такой странице не было сделано обращений, она выбрасывается из рабочего набора. Параметр n придется определить экспериментальным путем, но производительность системы обычно не слишком чувствительна к его точному значению.

При помощи информации рабочего набора можно увеличить производительность алгоритма «часы». Обычно, когда «стрелка часов» показывает на страницу, у которой бит R равен нулю, эта страница удаляется. Чтобы улучшить алгоритм, можно дополнительно проверять, входит ли страница в рабочий набор текущего процесса, и если да, оставлять ее. Такой алгоритм называется *wsclock*.

4.5.2. Политики распределения памяти: локальная и глобальная

В предыдущих разделах мы обсудили несколько алгоритмов, выполняющих поиск страницы для замещения, когда происходит прерывание. Основной вопрос, связанный с этим выбором (который мы тщательно обходили до сих пор): как должна быть распределена память между параллельными конкурирующими работоспособными процессами?

Обратим внимание на рис. 4.16, *а*. Здесь три процесса, A, B и C, составляют набор работоспособных процессов. Предположим, процесс A вызвал страничное прерывание. Должен ли алгоритм замещения страниц пытаться найти наиболее давно использовавшуюся страницу, учитывая только шесть страниц, предоставленные в данный момент процессу A, или же он должен рассматривать все страницы памяти? Если алгоритм производит поиск только среди страниц процесса A, наименьший возраст имеет страница A5, и мы получаем ситуацию, изображенную на рис. 4.16, *б*.

С другой стороны, если удаляется страница с наименьшим возрастом, независимо от того, к какому процессу она относится, то будет выбрана страница B3, и система попадет в состояние, показанное на рис. 4.16, *в*. Алгоритм на рис. 4.16, *б* называется *локальным*, а про схему на рис. 4.16, *в* говорят, что это *глобальный* алгоритм замещения страниц. Локальные алгоритмы соответствуют размещению каждого процесса в фиксированной области памяти. Глобальные алгоритмы динамически распределяют страничные блоки между выполняющимися процессами. Таким образом, количество страничных блоков, предоставленных каждому процессу, изменяется со временем.

В целом глобальные алгоритмы работают лучше, особенно если размер рабочего набора может изменяться за время жизни процесса. Если используется локальный алгоритм и рабочий набор увеличивается в размере, в результате мы получим пробуксовку, даже когда в системе существует достаточное количество свободных страничных блоков. Когда рабочий набор уменьшается, в случае локального алгоритма часть памяти потратится впустую. Если же используется глобальный алгоритм, система должна непрерывно выносить вердикты о том, сколько страничных блоков нужно предоставить каждому процессу. Можно наблюдать за размером рабочего набора с помощью битов возраста страниц, но этот метод не всегда позволяет избежать буксования. Рабочий набор может изменяться в размере за микросекунды, тогда как возрастные биты являются грубым усреднением за тик часов.

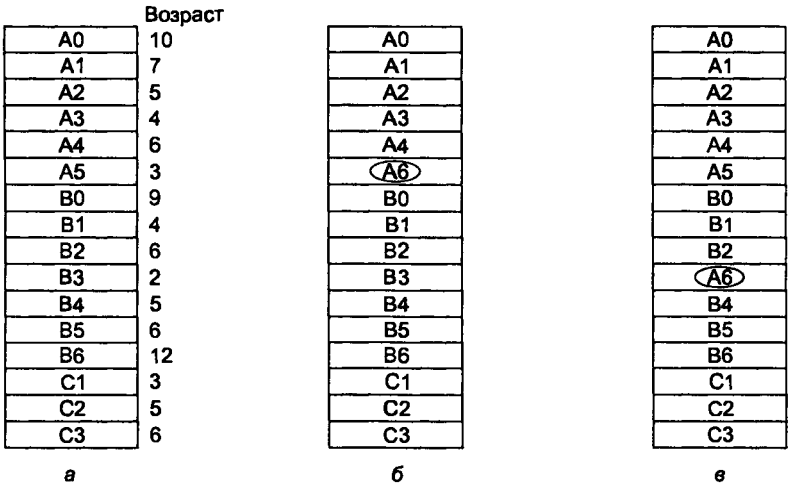


Рис. 4.16. Локальный алгоритм замещения страниц в сравнении с глобальным: *а* — исходная конфигурация; *б* — локальное замещение страниц; *в* — глобальное замещение страниц

Другой способ состоит в том, чтобы иметь в системе алгоритм для распределения страничных блоков между процессами. Например, можно периодически определять количество работающих процессов и предоставлять каждому равную часть памяти. Соответственно, при наличии доступных (то есть не принадлежащих операционной системе) 12 416 страничных блоков и 10 процессах каждый процесс получит 1241 блок. Оставшиеся шесть блоков поступают в резерв для использования в тот момент, когда происходит страничное прерывание.

Хотя этот метод кажется справедливым, существует небольшой шанс, что процессы размером 10 Кбайт и 300 Кбайт получат равные области памяти. Вместо этого можно предоставлять страницы пропорционально абсолютному размеру каждого процесса, тогда больший из этих двух процессов получит долю памяти в 30 раз больше, чем меньший процесс. Разумно отдавать каждому процессу некоторый минимум, чтобы он мог работать независимо от своего размера. На некоторых машинах, например, одиночная команда процессора, включающая в себя два операнда, может нуждаться в целых шести страницах, поскольку сама команда, операнд-источник и операнд-приемник могут располагаться на разных страницах. Если предоставить только пять страниц, программа, содержащая подобную инструкцию, вообще не сумеет выполниться.

Если используется глобальный алгоритм, допустимо запускать каждый процесс с некоторым количеством страниц, пропорциональным его размеру, но распределение памяти можно динамически изменять во время работы. Алгоритм *PFF* (Page Fault Frequency — частота страничных прерываний) предоставляет один из способов управления размещением процессов в памяти. Он говорит, когда увеличивать или уменьшать количество страниц, предоставленных процессу, но не упоминает о том, какую страницу замещать по прерыванию. Этот алгоритм только контролирует размер набора страниц, назначенных процессу.

Для большого класса схем замещения страниц, включая алгоритм LRU, известно, что частота прерываний уменьшается при увеличении числа предоставленных страниц, как мы обсуждали выше. Эта посылка лежит в основе алгоритма PFF. Данное свойство иллюстрирует рис. 4.17.

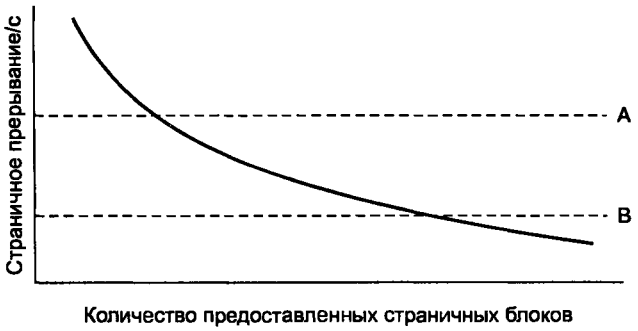


Рис. 4.17. Частота страничных прерываний как функция от количества предоставленных процессу страничных блоков

Прерывистая линия, обозначенная буквой A, соответствует частоте страничных прерываний, выше которой она недопустимо высока, поэтому увеличивается количество страничных блоков, предоставленных прерванному процессу, с целью уменьшения процента прерываний. Линия B соответствует очень низкой частоте страничных прерываний, позволяющей сделать вывод, что процесс занимает слишком много памяти. В этом случае у него можно забрать несколько страничных блоков. Таким образом, алгоритм PFF пытается сохранить частоту подкачки страниц для каждого процесса внутри допустимых границ.

Если в памяти оказывается так много процессов, что удержать их все выше линии A оказывается невозможно, некоторые процессы удаляются из памяти, а освободившееся пространство распределяется между остальными или заносится в банк свободных страниц для использования при последующих страничных прерываниях. Решение о том, какой процесс удалить из памяти, является одной из форм управления нагрузкой. Это показывает, что даже при страничной организации памяти бывает нужен свопинг, только здесь он служит во благо уменьшения потенциальной потребности в памяти, а не для возврата и немедленного использования блоков.

4.5.3. Размер страницы

Зачастую размер страницы является параметром, выбираемым операционной системой. Даже если аппаратное обеспечение предусматривает, например, размер страницы 512 байт, операционная система может просто рассматривать страницы 0 и 1, 2 и 3, 4 и 5 и т. д. как страницы размером 1 Кбайт, всегда предоставляя для них два последовательных страничных блока.

Определение наилучшего размера страниц требует равновесия нескольких параллельных факторов. Поэтому не существует абсолютного оптимального решения. Прежде всего, есть два довода в пользу маленького размера страниц. Случайно выбранный текст, данные или сегмент стека не заполняют целое количество страниц. В среднем половина последней страницы оказывается пустой, и это дополнительное пространство пропадает. Такие потери называют *внутренней фрагментацией*. Если в памяти n сегментов, а размер страницы равен p байтам, $np/2$ байт будет потрачено ни на что в результате внутренней фрагментации. Это разумный аргумент в пользу страниц небольшого размера.

Другой довод становится очевидным, если мы представим себе программу, выполняемую в восемь последовательных этапов, по 4 Кбайт каждый. При размере страницы 32 Кбайт программе должно быть постоянно выделено 32 Кбайт. При размере страницы 16 Кбайт ей необходимо только 16 Кбайт. При размере страницы 4 Кбайт или меньше программа требует всего лишь 4 Кбайт в любой момент времени. То есть большой размер страницы скорее, чем маленький, станет причиной того, что в памяти находится неиспользуемая часть страницы.

С другой стороны, небольшой размер страницы означает, что программам будет нужно их большое количество, следовательно — огромная таблица страниц. Программа размером 32 Кбайт требует всего четыре страницы по 8 Кбайт и 64 страницы по 512 байт. Как правило, страница за раз переносится на диск и с него, при этом большая часть времени уходит на поиск цилиндра и задержку вращения, так что перемещение маленькой страницы занимает почти столько же времени, сколько и большой. Может потребоваться 64 × 10 мс, чтобы загрузить 64 страницы размером 512 байт, и всего лишь 4 × 12 мс для загрузки четырех страниц по 8 Кбайт.

На некоторых машинах таблица страниц должна записываться в аппаратные регистры каждый раз, когда процессор переключается от одного процесса к другому. Если на таком компьютере страница имеет маленький размер, то время, требуемое для загрузки таблицы, будет увеличиваться пропорционально уменьшению размера страницы. Более того, пространство, занятое таблицей страниц, также возрастает с уменьшением страницы.

Этот последний момент можно проанализировать математически. Пусть средний размер процесса равен s байтам, а страницы — p байтам. Кроме того, предположим, что запись для каждой страницы требует e байтов. Тогда приблизительное количество страниц, необходимое для процесса, равно s/p, что соответствует se/p байтам для таблицы страниц. Потеря памяти в последней странице процесса вследствие внутренней фрагментации равна $p/2$. Таким образом, общие накладные расходы вследствие поддержки таблицы страниц и потери от внутренней фрагментации равны сумме этих двух составляющих:

$$расход = se/p + p/2.$$

Первое слагаемое (размер таблицы страниц) растет при уменьшении размера страницы. И наоборот, при увеличении размера страницы возрастает второе слагаемое (внутренняя фрагментация). Оптимальный вариант следует искать где-то посередине. Если взять первую производную по переменной p и приравнять ее к нулю, мы получим равенство:

$$-se/p^2 + 1/2 = 0.$$

Из этого равенства мы можем вывести формулу, дающую оптимально сбалансированный размер страниц (принимая во внимание только потери памяти на фрагментацию, а также величину таблицы страниц). В результате получится:

$$p = \sqrt{2se}.$$

Для среднего размера процесса s = 1 Мбайт и длины записи в таблице страниц e = 8 байт оптимальный размер страницы будет равен 4 Кбайт. В серийно выпускаемых компьютерах были приняты значения в диапазоне от 512 байт до 64 Кбайт.

4.5.4. Интерфейс виртуальной памяти

До сих пор в наших рассуждениях предполагалось, что виртуальная память прозрачна для процессов и программистов, то есть все, что они видят — это огромное виртуальное адресное пространство на компьютере с небольшой (или меньшей) физической памятью. Обычно это соответствует истине, но в некоторых прогрессивных системах программистам предоставлена определенная свобода управления картой памяти, которую они могут направить на улучшение поведения программы нетрадиционными способами. В этом разделе мы кратко рассмотрим некоторые из них.

Одной из причин предоставления программистам контроля над картой памяти является желание позволить двум и более процессам совместно использовать одну и ту же память. Если программисты сами будут давать названия областям памяти, один процесс сможет дать другому процессу имя области памяти, и этот второй процесс также сможет ей пользоваться. Если два (или больше) процессов разделяют страницы памяти, становится реальной высокая пропускная способность совместного доступа — один процесс пишет в разделяемую память, а другой читает из нее.

Совместное использование страниц также находит применение для высокопроизводительных систем передачи сообщений. Когда передается сообщение, данные обычно копируются из одного адресного пространства в другое, а это значительные издержки. Если процессы будут управлять своей картой страниц, можно передавать сообщения с помощью процесса-отправителя, убирающего из карты страницу (страницы) с сообщением, и процесса-получателя, помещающего ее (их) в карту. При этом должны копироваться только имена страниц вместо всех данных.

Еще одна современная техника управления памятью носит название *распределенной памяти совместного доступа* [32, 55, 89]. Она позволяет нескольким процессам в сети совместно использовать набор страниц, возможно (но не обязательно) как единственное разделяемое линейное адресное пространство. Когда процесс обращается к странице, не отображаемой в данный момент, он инициирует страничное прерывание. Обработчик страничных прерываний, находящийся в ядре или в пользовательском пространстве, определяет машину, содержащую страницу, и посылает ей сообщение с просьбой выгрузить страницу

и послать ее по сети. Когда страница прибывает, она попадает в карту, и прерванная команда перезапускается.

4.6. Сегментация

Обсуждавшаяся до сих пор виртуальная память представляет собой одномерное пространство, потому что виртуальные адреса идут один за другим от 0 до некоторого максимума. Для многих задач наличие двух и более отдельных виртуальных адресных пространств может оказаться намного лучше, чем всего одно. Например, у компилятора есть много таблиц, которые формируются по мере трансляции, возможно, включая в себя:

1. Исходный текст, сохраненный для печати листинга (в пакетных системах).
2. Символьную таблицу, содержащую имена и атрибуты переменных.
3. Таблицу, содержащую все используемые константы: целые и с плавающей точкой.
4. Дерево грамматического разбора, содержащее синтаксический анализ программы.
5. Стек, используемый для процедурных вызовов внутри компилятора.

Во время компиляции каждая из первых четырех таблиц непрерывно растет. Последняя таблица при компиляции непредсказуемо увеличивается или уменьшается. В одномерной памяти эти пять таблиц должны были бы размещаться в смежных частях виртуального адресного пространства, как на рис. 4.18.

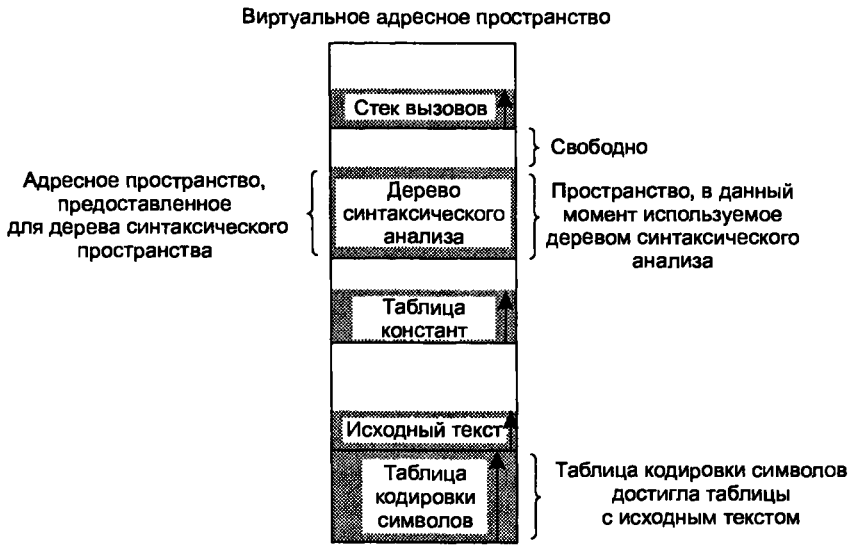


Рис. 4.18. В одномерном адресном пространстве при росте таблиц одна может упереться в другую

Рассмотрим, что происходит, если программа имеет исключительно большое число переменных, но нормальное количество всего остального. Участок адресного пространства, предоставленный для таблицы кодировки символов, может заполниться, но в других таблицах, скорее всего, останется пустым множество ячеек. Конечно, компилятор может просто создать сообщение о том, что компиляция не может продолжаться вследствие слишком большого количества переменных, но нам кажется, что такое решение проблемы не спортивно, когда в других таблицах осталась масса неиспользованного места.

При другом варианте можно поиграть в Робин Гуда, забирая пространство из таблиц с излишеством ячеек и передавая их таблицам с их недостатком. Такая перетасовка реализуема, но она аналогична управлению собственными оверлеями, что представляет собой маленькое неудобство в лучшем случае и большое количество скучной и неоплачиваемой работы в худшем случае.

На самом деле необходим метод, освобождающий программиста от управления расширяющимися и сокращающимися таблицами тем же способом, которым виртуальная память устраняет беспокойство организации программ с оверлеями.

Простое и предельно общее решение заключается в том, чтобы обеспечить машину множеством полностью независимых адресных пространств, называемых **сегментами**. Каждый сегмент содержит линейную последовательность адресов от 0 до некоторого максимума. Длина каждого сегмента может быть любой от нуля до разрешенного максимума. Различные сегменты могут быть различной длины. Более того, длины сегментов могут изменяться во время выполнения. Длина сегмента стека может увеличиваться всякий раз, когда что-либо помещается в стек, и уменьшаться при выборке данных из стека.

Поскольку каждый сегмент составляет отдельное адресное пространство, разные сегменты могут расти или сокращаться независимо друг от друга. Если стек, находящийся в определенном сегменте, нуждается в большем количестве адресного пространства для роста, он может получить его, потому что в его адресном пространстве нет больше ничего, с чем можно столкнуться. Конечно, сегмент может заполниться, но сегменты обычно очень большие, поэтому такие инциденты редки. Чтобы определить адрес в такой сегментированной или двумерной памяти, программа должна указать адрес, состоящий из двух частей: номер сегмента и адрес внутри сегмента.

Рисунок 4.19 иллюстрирует сегментированную память, использующуюся для обсуждавшихся ранее таблиц компилятора. Здесь показаны пять независимых сегментов.

Стоит подчеркнуть, что сегмент — это логический объект, о чем программист знает и поэтому использует его как логический объект. Сегмент может иметь в составе процедуру, массив, стек или набор скалярных переменных, но обычно он не содержит смеси различных типов.

Помимо простоты управления увеличивающимися или сокращающимися структурами данных, сегментированная память обладает и другими преимуществами. Если каждая процедура занимает отдельный сегмент и адрес 0 — это ее начальный адрес, компоновка отдельно скомпилированных процедур происходит намного проще. После того как все процедуры, составляющие программу, будут

скомпилированы и скомпонованы, для адресации слова 0 (начальной точки) обращение к процедуре в сегменте *n* будет использовать адрес, состоящий из двух частей (*n*, 0).

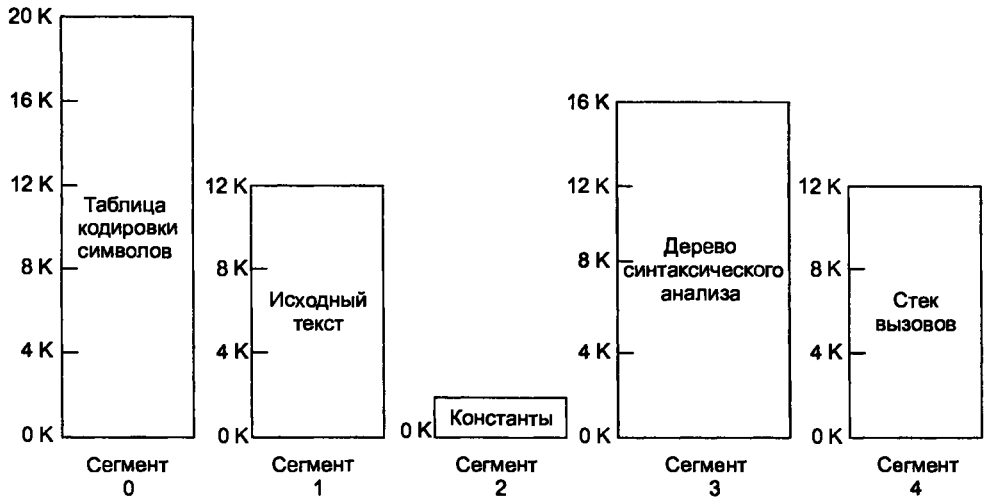


Рис. 4.19. Сегментированная память позволяет каждой таблице расти или уменьшаться независимо от других таблиц

Если потом процедура в сегменте *n* модифицируется и компилируется заново, не нужно изменять другие процедуры (потому что начальный адрес остался тем же), даже если новая версия больше предыдущей. В одномерной памяти процедуры упакованы одна к другой, и между ними нет свободного адресного пространства. В результате изменение размера одной процедуры может повлиять на начальный адрес другой, не имеющей отношения к первой процедуре. Это, в свою очередь, требует модификации всех процедур, вызывающих любую из передвинутых процедур, чтобы поместить в них новые начальные адреса. Если программа содержит сотни процедур, такой процесс очень дорог.

Сегментация также облегчает совместное использование процедур и данных несколькими процессами. Общим примером является **библиотека совместного доступа**. Современные рабочие станции, работающие с передовыми оконными системами, часто имеют крайне большие графические библиотеки, являющиеся составляющими практически каждой программы. В сегментированных системах графические библиотеки могут располагаться в отдельном сегменте и совместно использоваться несколькими процессами, что устраняет необходимость их присутствия в адресном пространстве каждого процесса. В принципе в системах с чистой страничной организацией памяти также можно иметь совместно используемые библиотеки, но это намного сложнее в реализации. Поэтому такие системы предоставляют совместный доступ путем моделирования сегментации.

Поскольку каждый сегмент формирует логический объект (такой как процедура, массив или стек), с которым общается программист, у различных сегмен-

тов могут быть разные виды защиты. Сегмент процедуры может быть определен как только исполняемый, что запрещает попытки чтения из него или сохранения в него. Для массива чисел с плавающей точкой можно разрешить режим доступа чтение/запись, но не исполнение, чтобы отлавливать попытки передачи управления по адресам, на которых располагается массив. Такая защита полезна при обнаружении ошибок программирования.

Вы должны попытаться понять, почему защита имеет смысл в сегментированной памяти, а не в одномерной страничной памяти. В сегментированной памяти пользователь осведомлен о том, что представляет собой каждый сегмент. В обычном случае сегмент не может содержать, например, и процедуру и стек, а только либо первое, либо второе. Так как каждый сегмент содержит только один тип объектов, он может иметь защиту, соответствующую этому конкретному типу. Страничная организация памяти и сегментация сравниваются в табл. 4.2.

Таблица 4.2. Сравнение страничной организации памяти и сегментации

Вопрос	Страничная память	Сегментация
Нужно ли программисту знать о том, что используется эта техника?	Нет	Да
Сколько в системе линейных адресных пространств?	1	Много
Может ли суммарное адресное пространство превышать размеры физической памяти?	Да	Да
Возможно ли разделение процедур и данных, а также раздельная защита для них?	Нет	Да
Легко ли размещаются таблицы с непостоянными размерами?	Нет	Да
Облегчен ли совместный доступ пользователей к процедурам?	Нет	Да
Зачем была придумана эта техника?	Чтобы получить большое линейное адресное пространство без дополнительных затрат на физическую память	Для возможности разбиения программ и данных на логически независимые адресные пространства, облегчения совместного доступа и защиты

Содержимое страниц в известной степени случайно. Программист не осведомлен даже о том факте, что происходит страничная подкачка. Хотя добавление нескольких битов в каждую запись таблицы страниц для определения разрешенного доступа в принципе возможно, но чтобы использовать это свойство, программист должен был бы отслеживать, где находятся границы страниц в его адресном пространстве. Это представляет собой в точности тот вид администрирования, для устранения которого была придумана страничная подкачка. По-

скольку пользователь сегментированной памяти имеет дело с иллюзией постоянного нахождения всех сегментов в оперативной памяти — то есть он может адресоваться к ним так, как будто они существуют, — он может защищать сегменты по отдельности, не заботясь об управлении их загрузкой в память.

4.6.1. Реализация сегментации

Реализация сегментации существенно отличается от страничной организации памяти: страницы имеют фиксированный размер, а сегменты — нет. На рис. 4.20, *а* показан пример физической памяти, изначально содержащей пять сегментов. Теперь взглянем, что произойдет, если удаляется сегмент 1, а на его место помещается сегмент 7 меньшего размера. Мы получим конфигурацию памяти, изображенную на рис. 4.20, *б*. Между сегментом 7 и сегментом 2 расположена неиспользуемая область, то есть дыра. Затем сегмент 4 замещается сегментом 5 (рис. 4.20, *в*), а сегмент 3 заменяется сегментом 6, как на рис. 4.20, *г*. После того как система поработает какое-то время, память разделится на некоторое количество участков, часть которых содержит сегменты, а остальные свободны. Этот феномен разделения памяти на маленькие свободные участки называется *поклеточной разбивкой* или *внешней фрагментацией*. С внешней фрагментацией можно бороться с помощью уплотнения, как показано на рис. 4.20, *д*.

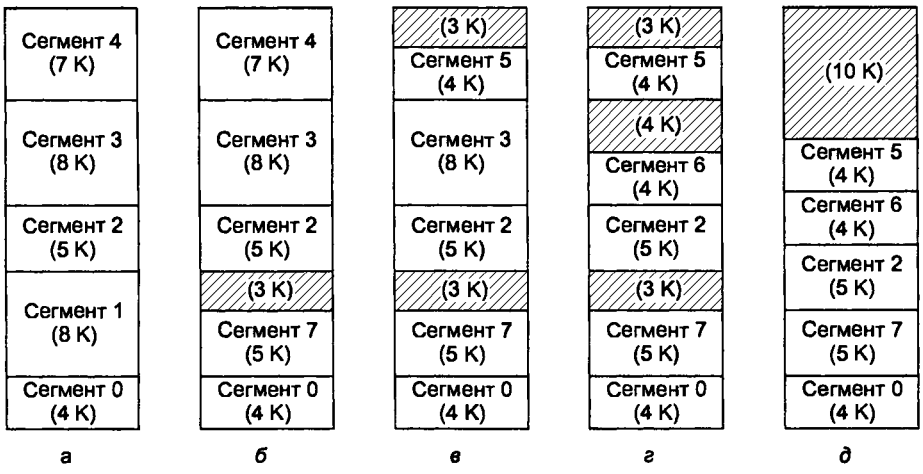


Рис. 4.20. *а–г* — развитие внешней фрагментации; *д* — устранение фрагментации с помощью уплотнения

4.6.2. Сегментация с использованием страниц: MULTICS

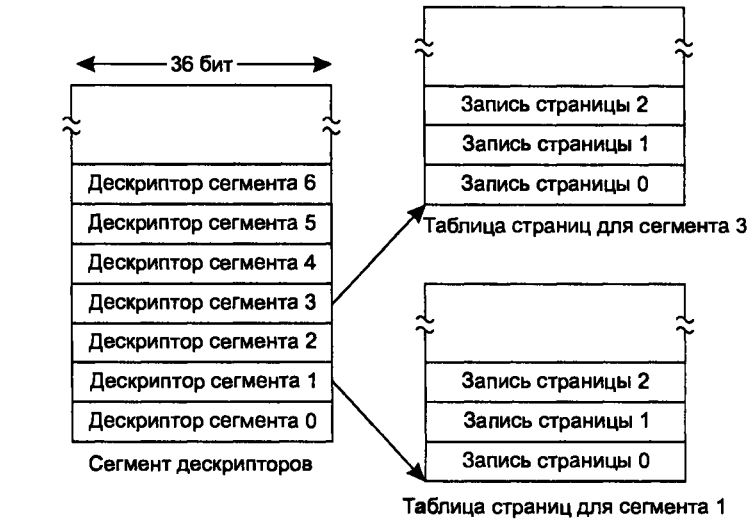
При большом размере сегментов может быть неудобно или даже невозможно хранить их в оперативной памяти целиком. Это приводит к идее их страничной организации, чтобы рядом располагались только те страницы, которые нужны на

самом деле. Страничные сегменты поддерживались несколькими важными для нас системами. В этом разделе мы будем описывать первую из них: MULTICS. В следующем разделе мы обратимся к более современной системе Intel Pentium.

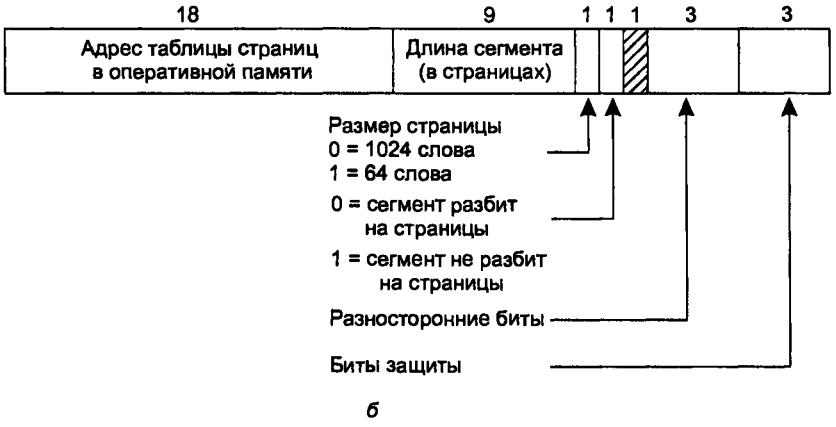
Система MULTICS работала на компьютерах Honeywell 6000 и их потомках и обеспечивала каждую программу виртуальной памятью размером вплоть до 2^{18} сегментов (более 250 000), с длинами до 65 536 (36-разрядных) слов. Чтобы добиться этого, разработчики системы MULTICS решили трактовать каждый сегмент как виртуальную память и разбить его на страницы, комбинируя преимущества страничной организации памяти (постоянный размер страницы и отсутствие необходимости хранения целого сегмента в памяти, если используется только его часть) с выгодой сегментации (упрощение программирования, модульности, защиты и совместного доступа).

Каждая программа в MULTICS имеет таблицу сегментов с одним дескриптором на сегмент. Так как записей в таблице потенциально больше четверти миллиона, таблица сегментов сама является сегментом и разбита на страницы. Дескриптор сегмента содержит индикатор того, находится ли сегмент в памяти или нет. Если какая-то часть сегмента присутствует в памяти, считается, что и сегмент в памяти, его таблица страниц также будет в памяти. Если сегмент находится в памяти, то его дескриптор содержит 18-разрядный указатель на его таблицу страниц (рис. 4.21, *а*). Поскольку физические адреса 24-разрядные, а страницы выравниваются по 64-байтовым границам (предполагается, что шесть битов низших разрядов адреса страницы — это 000000), необходимо только 18 бит в дескрипторе для хранения адреса таблицы страниц. Дескриптор также содержит размер сегмента, биты защиты и несколько других полей. Рисунок 4.21, *б* демонстрирует дескриптор сегмента в системе MULTICS. Адрес сегмента во вспомогательной памяти не включен в дескриптор сегмента, но в другой таблице используется обработчиком сегментных прерываний.

1. Каждый сегмент представляет собой обыкновенное адресное пространство и поделен на страницы точно так, как и несегментированная страничная память, описанная ранее в этой главе. Нормальный размер страницы равен 1024 словам (хотя несколько меньшие сегменты, используемые MULTICS, не разбиты на страницы или же все-таки разделены на блоки по 64 слова).
2. Адрес в системе MULTICS состоит из двух частей: сегмента и адреса внутри сегмента. Последний, в свою очередь, делится на номер страницы и слово внутри страницы, как показано на рис. 4.22. Когда происходит обращение к памяти, выполняется следующий алгоритм:
   1) по номеру сегмента находится дескриптор сегмента;
   2) проверяется, находится ли таблица страниц сегмента в памяти. Если да, определяется расположение таблицы. Если нет, генерируется сегментное прерывание. При нарушении защиты происходит прерывание;
   3) изучается запись в таблице страниц для запрашиваемой виртуальной страницы. Если страница не находится в памяти, возбуждается страничное прерывание. Если она в памяти, из записи таблицы страниц извлекается адрес начала страницы в оперативной памяти;



а



б

Рис. 4.21. Виртуальная память системы MULTICS: *а* — дескрипторы указывает на таблицы страниц; *б* — дескриптор сегмента. Числа означают длину полей

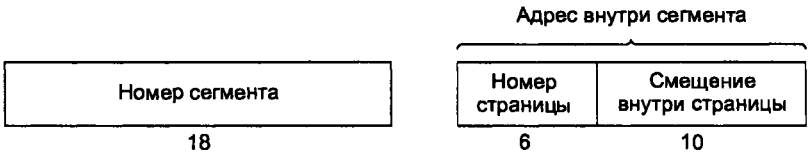


Рис. 4.22. 32-разрядный виртуальный адрес в системе MULTICS

4) к адресу начала страницы прибавляется смещение, что дает в результате адрес в оперативной памяти, где расположено нужное слово;

5) наконец, происходит чтение или сохранение.

Этот процесс продемонстрирован на рис. 4.23. Для простоты был опущен тот факт, что сегмент дескрипторов сам имеет страничное строение. Реально происходит следующее: регистр (основной регистр дескриптора) используется для определения расположения таблицы страниц сегмента дескрипторов, которая, в свою очередь, указывает на страницы сегмента дескрипторов. Как только дескриптор для необходимого сегмента найден, адресация продолжается согласно рис. 4.23.

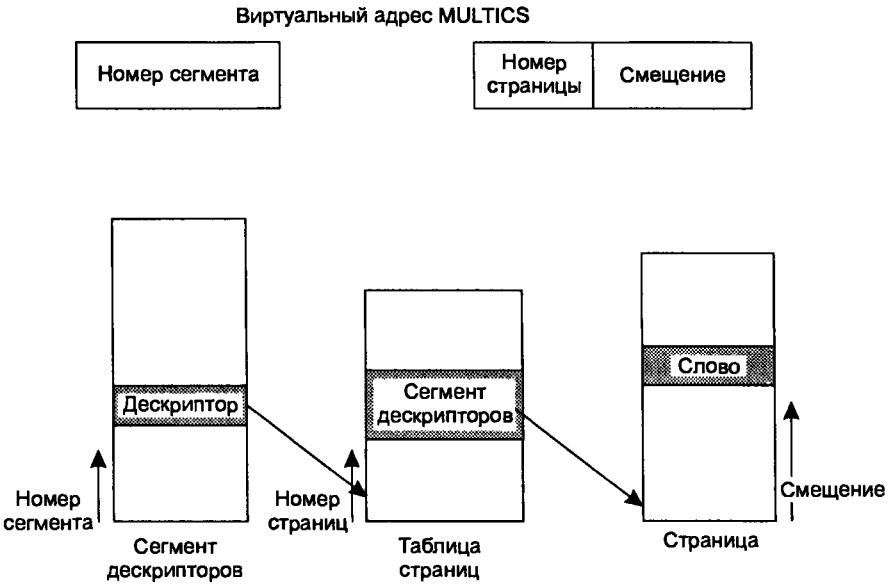


Рис. 4.23. Преобразование в системе MULTICS адреса, состоящего из двух частей, в адрес в оперативной памяти

Как вы, без сомнения, теперь догадались, если бы на практике предыдущий алгоритм выполнялся операционной системой для каждой команды процессора, программы были бы не слишком быстры. В действительности аппаратура для MULTICS содержит высокоскоростной буфер быстрого преобразования адреса (TLB) размером в 16 слов, способный производить поиск параллельно по всем своим записям для заданного ключа. Это проиллюстрировано на рис. 4.24. Когда компьютер получает адрес, аппаратура адресации сначала проверяет наличие виртуального адреса в TLB. При его наличии она получает номер страничного блока прямиком из TLB и формирует фактический адрес слова, к которому происходит обращение, не выполняя поиск в сегменте дескрипторов или таблице страниц.

Адреса 16 самых часто востребуемых страниц хранятся в буфере (TLB). Программы, у которых рабочий набор меньше размера буфера, будут хранить адреса

всего рабочего набора в TLB и, следовательно, работать эффективно. Если страница не находится в буфере быстрого преобразования адреса, фактически происходит обращение к дескриптору и таблице страниц, чтобы найти адрес страничного блока, и TLB обновляется для включения этой страницы. Тут же выгружается страница, не дольше других ожидавшая обращений. Поле «возраста» хранит информацию о том, какая из записей использовалась наиболее давно. Причиной для применения TLB служит обеспечиваемое им параллельное сравнение сегментов и номеров страниц всех записей.

Поле сравнения					Эта запись используется?
Номер сегмента	Виртуальная страница	Страничный блок	Защита	Возраст	
4	1	7	Чтение/запись	13	1
6	0	2	Только чтение	10	1
12	3	1	Чтение/запись	2	1
					0
2	1	0	Только выполнение	7	1
2	2	12	Только выполнение	9	1

Рис. 4.24. Простейший вариант буфера быстрого преобразования адреса в системе MULTICS. Два разных размера страниц делают фактическое строение TLB более сложным

4.6.3. Сегментация с использованием страниц: Intel Pentium

Виртуальная память на компьютере Pentium во многих отношениях аналогична памяти в системе MULTICS, включая наличие и сегментации, и страничной организации. В то время как MULTICS имеет 256 K независимых сегментов, каждый до 64 K 36-разрядных слов, система Pentium поддерживает 16 K независимых сегментов, каждый до 1 млрд 32-разрядных слов. Хотя в последней системе меньше сегментов, их увеличенный размер намного более важен, так как программы редко нуждаются более чем в 1000 сегментах, но многим программам необходимы сегменты значительного размера.

Основа виртуальной памяти Pentium состоит их двух таблиц: *локальной таблицы дескрипторов LDT* (Local Descriptor Table) и *глобальной таблицы дескрипторов GDT* (Global Descriptor Table). У каждой программы есть своя собственная таблица LDT, но глобальная таблица дескрипторов одна, ее совместно используют все программы в компьютере. Таблица LDT описывает сегменты, локальные для каждой программы, — ее код, данные, стек и т. д., тогда как таблица GDT несет информацию о системных сегментах, включая саму операционную систему.

Чтобы получить доступ к сегменту, программа системы Pentium сначала загружает селектор для этого сегмента в один из шести сегментных регистров машины. Во время выполнения регистр CS содержит селектор для сегмента кода, а регистр DS хранит селектор для сегмента данных. Каждый селектор представляет собой 16-разрядный номер (рис. 4.25).

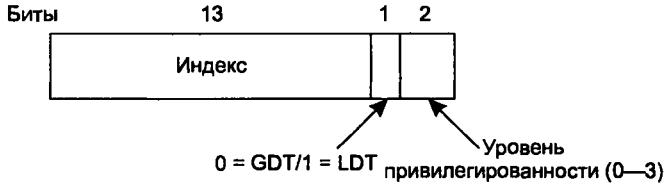


Рис. 4.25. Селектор в системе Pentium

Один из битов селектора говорит, является ли данный сегмент локальным или глобальным (то есть находится ли он в локальной или глобальной таблице дескрипторов). Следующие тринадцать битов определяют номер записи в таблице дескрипторов, поэтому эти таблицы ограничены: каждая содержит 8 К сегментных дескрипторов. Остальные два бита относятся к проблемам защиты и будут описаны позже. Дескриптор 0 является запрещенным — его можно безопасно загрузить в сегментный регистр, чтобы обозначить, что сегментный регистр в данный момент недоступен, но при попытке его использовать вырабатывается прерывание.

Во время загрузки селектора в сегментный регистр соответствующий элемент извлекается из локальной или глобальной таблицы дескрипторов и сохраняется в микропрограммных регистрах, что обеспечивает к нему быстрый доступ. Дескриптор состоит из 8 байт, в которые входят базовый адрес сегмента, размер и другая информация (рис. 4.26).

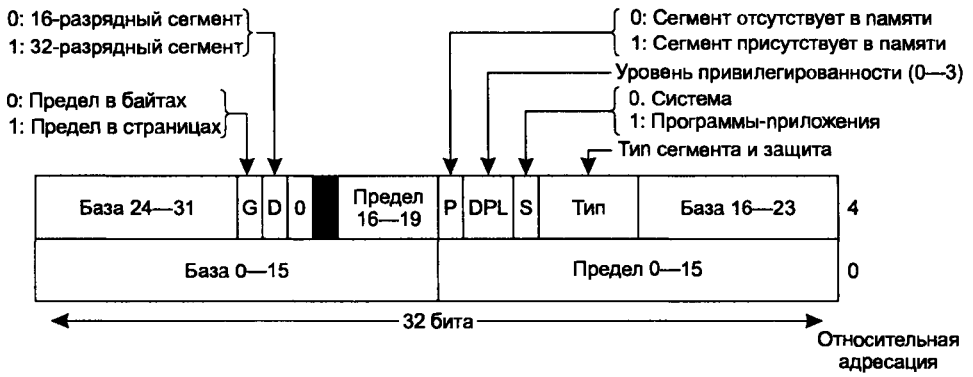


Рис. 4.26. Дескриптор программного сегмента в системе Pentium. Сегменты данных немного отличаются от программных сегментов

Формат селектора сознательно продуман так, чтобы упростить определение местоположения дескриптора. Сначала выбирается локальная или глобальная таблица дескрипторов, основываясь на бите 2 селектора. Затем селектор копируется во внутренний рабочий регистр, и три младших бита сбрасываются в нули. Наконец, к нему прибавляется адрес одной из таблиц, с целью получить прямой указатель на дескриптор. Например, селектор 72 ссылается на запись 9 в глобальной таблице дескрипторов, расположенную по адресу в таблице GDT+72.

Теперь проследим шаги, с помощью которых пара селектор-смещение преобразуется в физический адрес. Как только микропрограмма узнает, какой сегментный регистр используется, она может найти в своих внутренних регистрах полный дескриптор, соответствующий этому селектору. Если сегмент не существует (селектор равен 0) или в данный момент выгружен, происходит прерывание.

Затем микропрограмма проверяет, выходит ли смещение за пределы сегмента, в случае чего также возникает прерывание. Логически, в дескрипторе просто должно существовать 32-разрядное поле, дающее размер сегмента, но там доступны только 20 бит, поэтому действует другая схема. Если поле Gbit (granularity — глубина детализации) равно 0, то поле Limit (предел) содержит точный размер сегмента, до 1 Мбайт. Если гранулярность равна 1, поле Limit дает размер сегмента в страницах вместо байтов. Размер страницы в системе Pentium фиксирован на величине 4 Кбайт, поэтому 20 бит достаточно для сегментов размером до 2^{32} байт.

Предположим, что сегмент находится в памяти, а смещение попало в нужный интервал. Тогда Pentium прибавляет 32-разрядное поле Base (база) в дескрипторе к смещению, формируя то, что называется *линейным адресом*, как показано на рис. 4.27. Поле Base разбито на три части, которые разбросаны по дескриптору для совместимости с процессором Intel 80286, где поле Base имеет только 24 бита. В сущности, поле Base позволяет каждому сегменту начинаться в произвольном месте внутри 32-разрядного линейного адресного пространства.

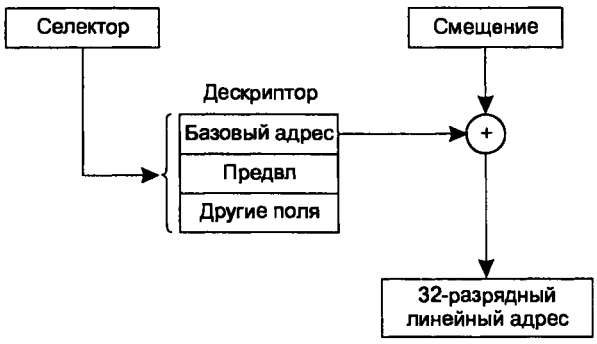


Рис. 4.27. Преобразование пары селектор-смещение в физический адрес

Если разбиение на страницы блокировано (с помощью бита в глобальном управляющем регистре), линейный адрес интерпретируется как физический адрес

и посылается в память для чтения или записи. Таким образом, при отключенной страничной схеме памяти мы получаем чистую схему сегментации с базовым адресом каждого сегмента, выдаваемым его дескриптором. Сегментам разрешено перекрываться случайным образом, возможно потому, что контроль за тем, чтобы они не пересекались, мог бы причинить достаточно хлопот и занял бы слишком много времени.

С другой стороны, если доступна страничная подкачка, линейный адрес интерпретируется как виртуальный адрес и отображается на физический адрес с помощью таблицы страниц практически так же, как в наших предыдущих примерах. Единственная серьезная трудность заключается в том, что при 32-разрядном виртуальном адресе и странице размером 4 Кбайт сегмент может содержать 1 млн страниц, поэтому используется двухуровневое отображение с целью уменьшения размера таблицы страниц для маленьких сегментов.

У каждой работающей программы есть *страничный каталог*, состоящий из 1024 32-разрядных записей. Он расположен по адресу, хранящемуся в глобальном регистре. Каждая запись в каталоге ссылается на таблицу страниц, также содержащую 1024 32-разрядных записей. Записи в таблицах страниц, в свою очередь, указывают на страничные блоки. Эта организация продемонстрирована на рис. 4.28.

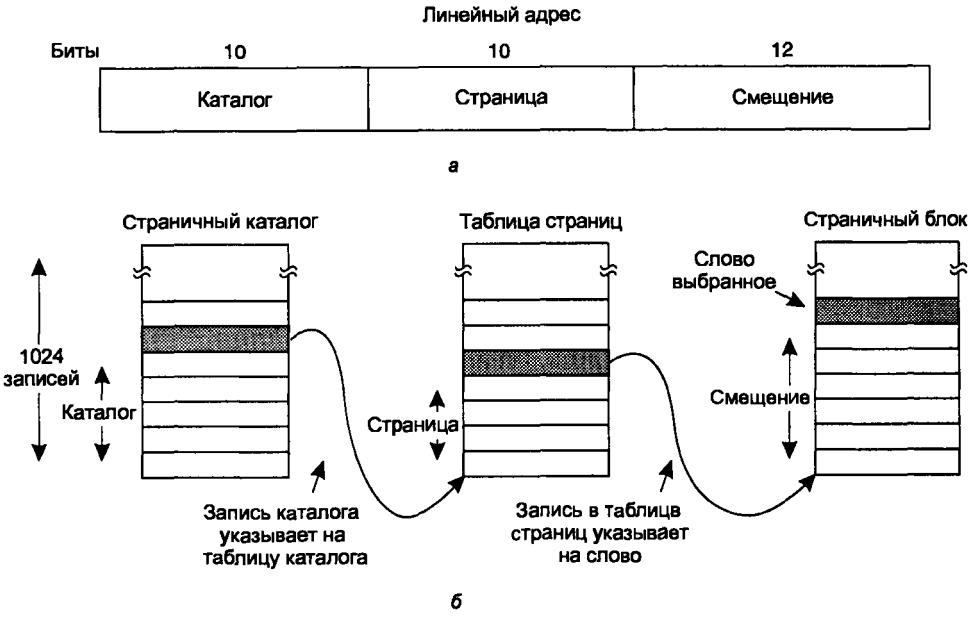


Рис. 4.28. Отображение линейного адреса на физический адрес

На рис. 4.28, *а* мы видим линейный адрес, разделенный на три поля: Каталог, Страница и Смещение. Поле Каталог используется как индекс в страничном каталоге, определяющий расположение указателя на правильную таблицу страниц.

Затем обрабатывается поле Страница в качестве индекса в таблице страниц, с целью найти физический адрес страничного блока. И наконец, чтобы получить физический адрес требуемого байта или слова, к адресу страничного блока прибавляется последнее поле Смещение.

Каждая запись в таблице имеет размер 32 бита, двадцать из которых содержат номер страничного блока. Остальные биты — это биты доступа и «грязный» бит, задаваемые аппаратурой для операционной системы, биты защиты и другие полезные биты.

Каждая таблица страниц включает в себя записи для 1024 страничных блоков размером по 4 Кбайт, таким образом, одна таблица страниц управляет четырьмя мегабайтами памяти. Сегмент, длина которого меньше 4 М, будет иметь страничный каталог с единственной записью — указателем на его единственную таблицу страниц. Следовательно, в случае короткого сегмента на поддержку таблиц страниц расходуется только две страницы вместо миллиона, который был бы нужен в одноуровневой таблице страниц.

Чтобы избежать повторных обращений к памяти, система Pentium, как и MULTICS, имеет небольшой буфер быстрого преобразования адреса (TLB), который напрямую отображает наиболее часто используемые комбинации Каталог-Страница на физический адрес страничного блока. Только когда текущая комбинация отсутствует в TLB, действительно выполняется механизм, показанный на рис. 4.28, и TLB обновляется. Система обладает хорошей производительностью до тех пор, пока обращения к отсутствующим страницам в TLB происходят относительно редко.

Также следует отметить, что когда некоторые приложения не требуют сегментации, а довольствуются единым, разбитым на страницы 32-разрядным адресным пространством, эта модель все равно работает. Все сегментные регистры могут быть настроены тем же самым селектором, в дескрипторе которого поле Base = 0 и поле Limit установлено на максимум. Тогда, при использовании единственного адресного пространства, смещение команды будет линейным адресом — в сущности, это обычная страничная организация памяти. Фактически все современные операционные системы для компьютера Pentium работают таким образом. Система OS/2 была единственной, которая использовала всю мощь архитектуры диспетчера памяти (MMU) фирмы Intel.

В конце концов, кто-то должен похвалить разработчиков системы Pentium. При поставленных перед ними противоречивых задачах — реализовать чистую страничную организацию памяти, чистое сегментирование и страничные сегменты, и в то же время обеспечить совместимость с 286-м процессором, а кроме того, сделать все это эффективно, — результирующая структура удивительно проста и понятна.

Мы, хотя и кратко, но целиком описали полную архитектуру виртуальной памяти системы Pentium, и теперь следует сказать несколько слов о защите, так как эта тема тесно связана с виртуальной памятью. Как схема виртуальной памяти, так и система защиты на Pentium близка по модели к системе MULTICS. Pentium поддерживает четыре уровня защиты, где уровень 0 является наиболее привилегированным, а уровень 3 — наименее привилегированным. Они показаны

на рис. 4.29. В каждый момент времени работающая программа находится на определенном уровне, что отмечается 2-битовым полем в его регистре слова состояния программы (PSW). Каждый сегмент в системе также имеет свой уровень.

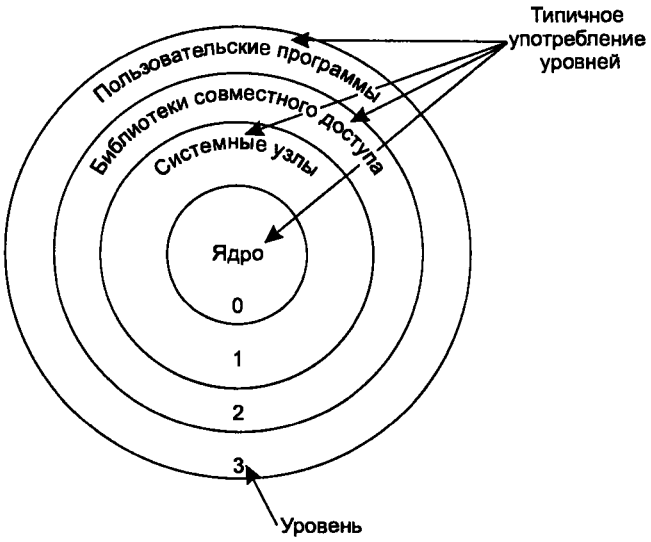


Рис. 4.29. Защита в системе Pentium

До тех пор пока программа сама ограничивает использование сегментов на своем собственном уровне, система прекрасно работает. Разрешаются попытки получения доступа к данным высшего уровня. Попытки доступа к данным более низкого уровня запрещены и вызывают прерывания. Попытки вызвать процедуры различного уровня (более высокого или низкого) позволяются, но тщательно контролируемым образом. Чтобы сделать межуровневый вызов, инструкция CALL должна содержать селектор вместо адреса. Этот селектор определяет дескриптор, называемый *шлюзом вызова* (call gate) и передающий адрес вызываемой процедуры. Таким образом, перепрыгнуть в середину произвольного сегмента кода другого уровня невозможно, открыты лишь официальные точки входа. Концепция уровней защиты и схем вызова впервые появилась в системе MULTICS, где они представали в виде *колец защиты*.

Типичное использование этого механизма представлено на рис. 4.29. На уровне 0 мы находим ядро операционной системы, занимающееся обработкой операций ввода/вывода, управлением памятью и другими первоочередными вопросами. На уровне 1 находится обработчик системных вызовов. Пользовательские программы этого уровня могут обращаться к процедурам для выполнения системных вызовов, но только к определенному и защищенному списку процедур. Уровень 2 содержит библиотечные процедуры, возможно, совместно используемые несколькими работающими программами. Пользовательские программы

вправе вызывать эти процедуры и читать их данные, но не могут их изменять. И наконец, пользовательские программы работают на уровне 3, который имеет наименьшую степень защиты.

В эмулируемые и аппаратные прерывания заложен механизм, аналогичный шлюзам вызовов. Они тоже обращаются к дескрипторам, а не к абсолютным адресам, эти дескрипторы указывают на определенные процедуры. Поле Тип на рис. 4.26 позволяет различать программные сегменты, сегменты данных и шлюзы вызовов различных видов.

4.7. Обзор управления памятью в MINIX

В MINIX управление памятью реализовано просто: ни свопинг, ни страничная организация не используются. Менеджер памяти поддерживает список свободных участков («дыр»), отсортированный по адресам. Когда процессу вследствие вызова fork или exec требуется память, элементы списка перебираются до тех пор, пока не будет найдена первая достаточно большая дыра. Расположившись в памяти, процесс всегда остается на прежнем месте до своего завершения. Он никогда не выгружается на диск и не перемещается по другому адресу. Кроме того, он никогда не увеличивает и не уменьшает выделенную ему область памяти.

Такая стратегия требует некоторых пояснений. Ее определяют три фактора: (1) MINIX предназначена для персональных компьютеров, а не для больших систем с разделением времени, (2) MINIX должна работать на всех PC, и (3) система должна быть простой, чтобы ее можно было реализовать на всех малых компьютерах.

Первый фактор означает, что в среднем количество работающих процессов будет небольшим и памяти с избытком хватит для всех. Тогда свопинг вообще не требуется. А так как свопинг усложняет систему, отказ от него означает более простой код.

Желание заставить MINIX работать на всех IBM-совместимых компьютерах также оказало большое влияние на выбор стратегии менеджера памяти. Простейшим в этом семействе является процессор 8086, с очень примитивной архитектурой. Он ни в какой форме не поддерживает виртуальную память и даже не обнаруживает переполнение стека. В более поздних процессорах, 80386, 80486 и Pentium, таких ограничений нет, но если взять на вооружение их возможности, MINIX станет несовместима с 8086.

С позиций переносимости, схема управления памятью должна быть до нельзя простой. Если бы MINIX опиралась на страничную организацию или сегментацию, перенести систему на машины без таких возможностей было бы сложно. Чем меньше предположений о потенциале оборудования закладывается в систему, тем шире становится круг платформ, на которые ее можно перенести.

Еще одной необычной особенностью MINIX является способ реализации менеджера памяти. Последний не является частью ядра. Управлением памятью занимается отдельный процесс, работающий в адресном пространстве пользова-

теля и взаимодействующий с ядром при помощи стандартного механизма сообщений. На рис. 2.24 менеджер памяти находится на уровне серверов.

Исключение менеджера памяти из ядра — пример разделения *политики* и *механизма*. Решение о том, какая область памяти будет отведена каждому из процессов (политика), принимается менеджером памяти. Реальное же обслуживание карт памяти для процессов (то есть механизм) выполняется задачей системы в ядре. Подобное разделение позволяет легко изменить политику управления памятью (алгоритмы и т. д.), не затрагивая нижние уровни операционной системы.

Большая часть кода менеджера памяти в MINIX посвящена обработке системных вызовов, связанных с управлением памятью (в основном это fork и exec), а не манипулированию списками процессов и свободных блоков памяти. В следующем разделе мы изучим распределение памяти, а еще дальше последует обзор того, как в MINIX выполняются системные вызовы, обрабатываемые менеджером памяти.

4.7.1. Распределение памяти

В MINIX у простых процессов адресные пространства кода и данных объединяются — распределяются и реализуются как один блок. Для большей ясности сначала мы рассмотрим выделение памяти в простой модели. Процессы с разделенными адресными пространствами кода и данных в состоянии более эффективно использовать память, но это вносит дополнительные сложности, их мы рассмотрим после того, как разберемся с простыми вещами.

В MINIX память выделяется в двух ситуациях. Во-первых, когда процесс разветвляется, дочернему процессу предоставляется необходимая ему память. Во-вторых, когда процесс при помощи системного вызова exec заменяет свой образ, старый образ памяти процесса возвращается в список свободных блоков памяти, а новая память выделяется на новом месте. Положение нового образа процесса в памяти будет зависеть от того, где найден первый подходящий по размеру блок. Кроме того, когда процесс завершается (самостоятельно или принудительно, по сигналу), занятая им память освобождается.

На рис. 4.30 показаны оба варианта выделения памяти. На рис. 4.30, *а* мы видим в памяти два процесса, A и B. Если процесс A разветвляется, мы попадаем в ситуацию на рис. 4.30, *б*. Дочерний процесс является точной копией процесса A. Если теперь дочерний процесс выполнит файл C, память придет в состояние, показанное на рис. 4.30, *в*. Образ дочернего процесса был заменен образом C.

Обратите внимание, что область памяти, занимаемая потомком, освобождается перед тем, как выделить память для нового образа, поэтому C может занять память, в которой раньше располагался потомок. Таким образом, после выполнения серии пар вызовов fork и exec все процессы будут соседствовать в памяти впритирку друг к другу. Если бы память для нового образа выделялась первой, между процессами обязательно зияли бы дыры.

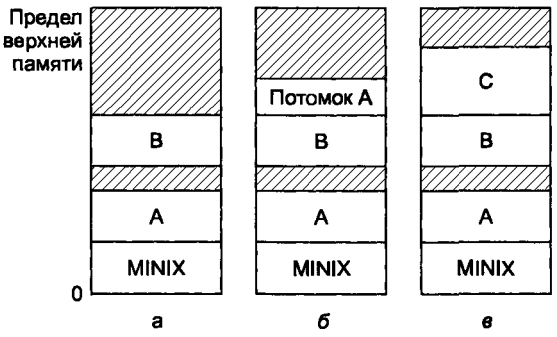


Рис. 4.30. Выделение памяти: *а* — исходное состояние; *б* — после вызова fork; *в* — дочерний процесс сделал вызов exec. Неиспользованные области памяти отмечены штриховкой. У процесса код и данные находятся в одном адресном пространстве

Когда память распределяется по вызову fork или exec, некоторое ее количество отдается новому процессу. В первом случае запросы дочернего и родительского процессов удовлетворяются в равной мере. Во втором — менеджер памяти оперирует значением из заголовка исполняемого файла. После того как память была раз выделена, ни при каких условиях процесс не сможет посягнуть на дополнительную.

Сказанное выше относится к программам, которые были скомпилированы с объединенными адресными пространствами кода и данных. Программы, у которых эти пространства разделены, пользуются улучшенным вариантом управления памятью, который называется *разделяемым кодом* (shared text). Когда подобный процесс делает fork, память выделяется только под стек и данные дочернего процесса. Потомок получает в наследство исполняемый код, который используется родительским процессом. Когда делается exec, в таблице процессов ищется процесс, уже использующий требуемый код. Если такой процесс обнаруживается, память, опять же, выделяется только для данных и стека. Но совместное использование кода усложняет завершение процесса. При завершении процесса всегда освобождается память, занимаемая его данными и стеком. Память же, занимаемая сегментом текста, освобождается только и только тогда, если поиск в таблице процессов показал, что ни один другой процесс этот код больше не интересует. Может получиться так, что процессу при запуске выделяется больше памяти, чем освобождается при его завершении. Мы говорим про случай, если при запуске он загрузил собственный текст, а по окончании оказалось, что этот текст уже используется другими процессами.

На рис. 4.31 показано, как программа хранится в виде файла на диске и как она располагается в памяти, когда MINIX запускает процесс. Информация о размерах различных частей образа процесса находится в заголовке файла, так же как и сведения о его полном размере. Для программы с объединенными адресными пространствами кода и данных в заголовке указывается суммарный размер двух частей, они копируются напрямую в образ в памяти. Для сегмента данных образа при копировании в память выделяется больший объем. Размер дополни-

тельной области памяти в байтах хранится в поле bss заголовка. Эта область заполняется нулями и используется для неинициализированных статических данных. Общий объем памяти, который будет выделен процессу, задается полем заголовка total. Если, например, у программы код имеет длину 4 Кбайт, данные плюс bss занимают 2 Кбайт, и 1 Кбайт стек, а в заголовке указано выделить всего 40 Кбайт, то неиспользуемый промежуток памяти между сегментами данных и стека будет иметь размер 33 Кбайт. Кроме того, программа на диске может содержать таблицу символов. Она требуется для отладки и не загружается в память.

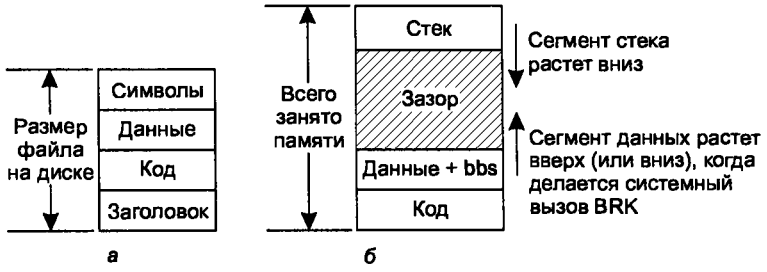


Рис. 4.31. *а* — программа хранится на диске в виде файла; *б* — внутреннее распределение памяти для одного процесса. На обеих частях рисунка в нижней части расположены нижние адреса

Если программист знает, что общий объем памяти, необходимый для стека и данных программы в файле a.out, не превышает 10 Кбайт, то при помощи команды

```
chmem = 10240 a.out
```

он может изменить поле заголовка исполняемого файла, после чего вызов exec будет выделять только 10 240 байт дополнительной памяти. Так, для описанного выше примера будет выделено 16 Кбайт памяти. Из этой памяти верхний килобайт будет отведен под стек, а 9 Кбайт образуют свободную область для будущего роста стека и/или области данных.

Для программ с раздельными адресными пространствами кода и данных (у которых компоновщик устанавливает специальный бит в заголовке программы), поле общего размера в заголовке относится только к стеку и данным. Так, при загрузке программы с 4 Кбайт кода, 2 Кбайт данных, 1 Кбайт стека и суммарным размером 64 Кбайт будет выделено 68 Кбайт памяти (4 Кбайт на код и 64 Кбайт образуют адресное пространство данных, а 61 Кбайт останется для роста стека и данных). Граница сегмента данных может быть изменена только при помощи системного вызова brk. При выполнении этого вызова проверяется, не упирается ли новая граница сегмента данных в нижнюю границу стека, и соответствующие изменения вносятся во внутренние таблицы. Это делается исключительно на уровне менеджера памяти, так как у системы не запрашивается никаких новых блоков памяти. Если же после изменения границы сегмент данных будет пересекаться со стеком, вызов завершается ошибкой.

Подобная стратегия была выбрана для того, чтобы MINIX могла работать на IBM PC с процессором 8086, который не делает аппаратной проверки переполнения стека. Пользовательская программа может поместить в стек сколь угодно много слов, не ставя об этом в известность операционную систему. На системах с более сложным механизмом управления памятью под стек сначала выделяется некоторый объем памяти. Если делается попытка превысить квоту, генерируется прерывание и система, если это возможно, выделяет под стек дополнительное пространство. У процессоров 8086 такой механизм отсутствует, поэтому в их компании опасно оставлять стек соседствовать с чем-либо помимо большого блока незадействуемой памяти, так как стек может расти быстро и без предупреждения. MINIX написана так, чтобы при реализации на компьютере с улучшенным управлением памятью менеджер памяти было бы легко заменить.

Тут нужно упомянуть небольшую семантическую сложность. Когда мы говорим «сегмент», мы имеем в виду область памяти, определяемую операционной системой. Но у процессоров Intel 80x86 есть специальные внутренние «сегментные регистры» и (у более новых из них) «таблицы дескрипторов сегментов», обеспечивающие аппаратную поддержку «сегментов». Концепция сегмента в аппаратной архитектуре Intel сходна с тем, как сегменты определяются в MINIX, но это не одно и то же. Все ссылки на сегменты в этом тексте следует рассматривать в контексте их определения в MINIX. Когда мы будем говорить об аппаратных сегментах, мы будем явно упоминать сегментные регистры и дескрипторы сегментов.

Это предупреждение может быть обобщено. Разработчики аппаратного обеспечения часто стараются обеспечить целенаправленную поддержку операционных систем, под оборудование. Поэтому терминология описания регистров и других аспектов архитектуры процессора обычно отражает понимание того, как все это будет использоваться. Подобные возможности часто полезны для разработчиков операционной системы, но практика не обязательно соответствует тому, что предвидел производитель оборудования. В результате звучащие одинаково термины, но имеющие разное значение при описании аспектов операционной системы и оборудования, могут привести к непониманию.

4.7.2. Обработка сообщений

Как и все остальные компоненты ОС MINIX, менеджер памяти управляется сообщениями. После инициализации системы менеджер памяти входит в свой главный цикл, в котором сообщение принимается, выполняется содержащийся в сообщении запрос и отправляется ответное сообщение. Допустимые типы сообщений, их параметры и отправляемые в ответ значения перечислены в табл. 4.3.

Вызовы fork, exit, wait, waitpid, brk и exec тесно связаны с выделением и освобождением памяти. Вызовы kill, alarm и pause связаны с сигналами, такими как SIGALARM, SIGSUSPEND, SIGPENDING, SIGMARK и SIGRETURN. Они оказывают влияние и на память, так как при завершении процесса по сигналу занимаемая этим процессом память высвобождается. Вызов reboot воздействует на всю операционную систему, но прежде всего он отправляет сигналы, чтобы корректно завершить все

процессы, поэтому менеджер памяти — вполне подходящее место для выполнения этого вызова. Семь вызовов get/set вообще не имеют никакого отношения к управлению памятью. Они были помещены в код менеджера просто потому, что код файловой системы и без того достаточно велик. Вызов ptrace, применяемый при отладке, попал сюда по той же причине.

Таблица 4.3. Типы и параметры сообщений для менеджера памяти и ответные значения

Тип сообщения	Параметры	Ответное значение
FORK	(Нет)	PID потомка
EXIT	Код возврата	(При успехе ответа нет)
WAIT	(Нет)	Состояние
WAITPID	(Нет)	Состояние
BRK	Новый размер	Новый размер
EXEC	Указатель на исходный стек	(При успехе ответа нет)
KILL	Идентификатор процесса и сигнал	Состояние
ALARM	Время ожидания в секундах	Оставшееся время
PAUSE	(Нет)	(При успехе ответа нет)
SIGACTION	Номер сигнала, код операции, старый код операции	Состояние
SIGSUSPEND	Маска сигналов	(При успехе ответа нет)
SIGPENDING	(Нет)	Состояние
SIGMASK	Как изменять, новый набор, старый набор	Состояние
SIGRETURN	Контекст	Состояние
GETUID	(Нет)	UID и его действующее значение
GETGID	(Нет)	GID и его действующее значение
GETPID	(Нет)	PID и его действующее значение
SETUID	Новый UID	Состояние
SETGID	Новый GID	Состояние
SETSID	Новый SID	Группа процессов
GETPGRP	Новый GID	Группа процессов
PTRACE	Запрос, PID, адрес, данные	Состояние
REBOOT	Действие (останов, перезагрузка, паника)	(При успехе ответа нет)
KSIG	Ячейка процесса в таблице и сигналы	(Нет ответа)

Последнее сообщение, KSIG, не связано с системным вызовом. Этот тип сообщений используется ядром для того, чтобы информировать его о сигнале, исходящем из ядра. Примерами таких сигналов являются SIGINT, SIGQUIT и SIGALARM.

Несмотря на существование библиотечного вызова sbrk, системного вызова sbrk нет. Эта подпрограмма вычисляет объем необходимой памяти, уменьшая

или увеличивая текущее значение на величину переданного ей аргумента, и делает вызов brk, чтобы изменить размер области. Точно так же нет отдельных системных вызовов для geteuid и getegid. Вызовы getuid и getgid возвращают как действующие, так и реальные идентификаторы пользователя и группы. Аналогично, вызов getpid возвращает идентификаторы как самого процесса, так и его родителя.

Ключевой структурой данных для обработки сообщений является таблица call_vec, определяемая в файле table.c. Она содержит указатели на подпрограммы, обрабатывающие все различные типы сообщений. Когда сообщение попадает в менеджер памяти, команды в главном цикле извлекают из него тип сообщения и записывают его в глобальную переменную mm_call. Позже это значение используется как индекс в таблице callvec, и по нему находится адрес подпрограммы для обработки прибывшего сообщения. Затем эта подпрограмма отрабатывает, выполняя системный вызов. Значение, которое возвращает обработчик, отправляется обратно, чтобы сообщить о выполнении или ошибке. Механизм подобен механизму обработки системных вызовов, но только в адресном пространстве пользователя, а не ядра.

4.7.3. Структуры данных и алгоритмы менеджера памяти

У менеджера памяти есть две ключевые структуры данных: таблица процессов и таблица свободных блоков памяти («дыр»).

Из табл. 2.1 видно, что одни поля таблицы процессов нужны для управления процессами, другие для управления памятью, а третьи требуются файловой системе. В MINIX каждая из трех частей операционной системы поддерживает собственную таблицу процессов, содержащую только те поля, которые интересны ей. Записи всех трех таблиц соответствуют друг другу, чтобы не усложнять дело. Так, ячейка *k* таблицы процессов менеджера памяти соответствует тому же процессу, что и ячейка *k* из таблицы процессов файловой системы. При создании или уничтожении процесса необходимо обновлять записи во всех трех таблицах, чтобы поддерживать их в синхронизированном состоянии.

Таблица процессов менеджера памяти называется mproc, она определена в файле /usr/src/mm/mproc.h. В этой таблице содержатся поля, связанные с выделением памяти, а также некоторые дополнительные сведения. Самым важным полем является mp_seg, у которого есть три записи, для сегмента текста (кода), данных и стека. Все эти величины измеряются не в байтах, а в кликах. Размер клика (минимального блока памяти) зависит от реализации. Стандартным значением для MINIX является 256 байт. Все сегменты начинаются на границе клика и содержат целое число кликов.

Способ записи информации о выделении памяти проиллюстрирован рис. 4.32. На рисунке показан процесс с такой структурой: 3 Кбайт текста, 4 Кбайт данных, зазор имеет величину 1 Кбайт, а после него следует стек объемом 2 Кбайт. Всего выделено 10 Кбайт памяти. На рис. 4.32, *б* показано, что хранится в полях длины и виртуального и физического адреса для этих сегментов, при условии, что про-

цесс не разделяет адресные пространства кода и данных. В этой модели размер сегмента текста всегда считается равным нулю, а сегмент данных содержит и данные, и код. Когда показанный на рисунке процесс обращается к виртуальному адресу 0 или совершает переход по этому адресу, произойдет обращение к физическому адресу 0x32000 (в десятичной записи 200 Кбайт). В кликах этот адрес записывается как 0x320.

Нужно отметить, что виртуальный адрес, с которого начинается стек, зависит от общего объема памяти, выделенной процессу. Так, если при помощи команды chmem модифицировать заголовок исполняемого файла, чтобы при запуске для программы резервировалось больше динамически распределяемой памяти (промежуток между сегментами стека и данных), то при следующем запуске программы начало стека будет расположено выше. Если стек вырастет на один клик, его запись *должна* измениться с тройки (0x20, 0x340, 0x8) на тройку (0x1F, 0x33F, 0x9).

Аппаратное обеспечение процессоров 8088 не поддерживает прерывание переполнения стека, и в MINIX стек задается так, чтобы на 32-разрядных процессорах не инициировать прерывание до тех пор, пока сегмент стека не пересечется с сегментом данных. Таким образом, информация о стеке будет обновлена только при следующем системном вызове brk, при этом операционная система явно считывает значение SP (указатель стека) и пересчитывает параметры стека. На машинах, поддерживающих аппаратный контроль переполнения стека, информация о стеке должна обновляться как только стек переполнит свой сегмент. По причинам, которые мы далее обсудим, в 32-битной версии MINIX для процессоров Intel этого не делается.

а

	Виртуальный	Физический	Длина
Стек	0x20	0x340	0x8
Данные	0	0x320	0x1c
Код	0	0x320	0

б

	Виртуальный	Физический	Длина
Стек	0x14	0x340	0x8
Данные	0	0x32c	0x10
Код	0	0x320	0xc

в

Рис. 4.32. *а* — процесс в памяти; *б* — его представление с единым адресным пространством данных и кода; *в* — с раздельными пространствами данных и кода

Ранее мы уже упоминали, что усилия разработчиков аппаратного обеспечения не всегда приводят к тем результатам, которые нужны программистам. Даже работая в защищенном режиме Pentium, MINIX не отслеживает ситуацию, когда

стек переполняет свой сегмент. Хотя в этом режиме MINIX обрабатывает попытки обратиться к памяти за пределами сегмента (параметры которого определены дескриптором сегмента, см. рис. 4.26), в MINIX дескрипторы сегмента данных и сегмента стека всегда идентичны. В MINIX стек и данные находятся в едином адресном пространстве, и благодаря этому они могут расширяться за счет межсегментного зазора. Но это не более чем внутреннее представление MINIX. С точки зрения процессора, при обращении к памяти между сегментами нет никакой ошибки, так как она является частью общего сегмента данных и стека. Аппаратное обеспечение помогает отслеживать серьезные ошибки, например попытки обращения к памяти за пределами комбинированной области «данные-зазор-стек», благодаря чему один процесс можно защитить от ошибок другого процесса, но уберечь процесс самого от себя таким образом нельзя.

MINIX это делает сознательно. Если у вас есть аргументы за то, чтобы отказаться от разделяемого аппаратного сегмента и зазора, мы не будем спорить. Альтернативой могут быть два раздельных аппаратных сегмента для стека и данных. Такой подход обеспечивает большую безопасность в отношении некоторых ошибок, но это превратило бы MINIX в пожирающее память чудовище. Исходные коды системы открыты для всех, кто желает поэкспериментировать с таким подходом.

На рис. 4.33, *в* видно, как будут определены сегменты в случае раздельных адресных пространств кода и данных. Здесь уже и сегмент стека, и сегмент данных имеют ненулевую длину. Показанный на рис. 4.33, *б* и *в* массив mp_seg большей частью применяется для преобразования виртуальных адресов в физические. Зная виртуальный адрес и адресное пространство, которому он принадлежит, несложно проверить, является этот адрес допустимым (то есть попадает ли он в сегмент), и рассчитать, какому физическому адресу он соответствует. Например, такое преобразование осуществляет вспомогательная подпрограмма umap, которую задачи ввода/вывода привлекают для обмена данными с пользовательскими процессами.

При выполнении процесса содержимое его областей данных и стека может меняться, но код не меняется никогда. Часто случается, что несколько процессов выполняют одну и ту же программу, например несколько пользователей могут пользоваться одной оболочкой. Поэтому разделяемый текст повышает эффективность памяти. Когда системный вызов **exec** собирается загрузить процесс, он открывает файл с образом этого процесса и считывает заголовок. Если у процесса раздельные адресные пространства кода и данных, среди всех ячеек таблицы mproc осуществляется поиск по полям mp_dev, mp_ino, mp_ctime. Эти поля содержат информацию о номере *i*-узла и времени модификации образов, исполняемых другими процессами. Если обнаруживается процесс, который уже выполняет нужную программу, то выделять память под еще одну копию кода не нужно. Вместо этого в карту памяти нового процесса в поле mp_seg[T] записывается указатель на ту область памяти, где уже хранится код, а память выделяется только под данные и стек. Это продемонстрировано на рис. 4.34. Если загруженного образа не было найдено или адресные пространства кода и данных объединены, память выделяется согласно рис. 4.33 и заполняется данными с диска.

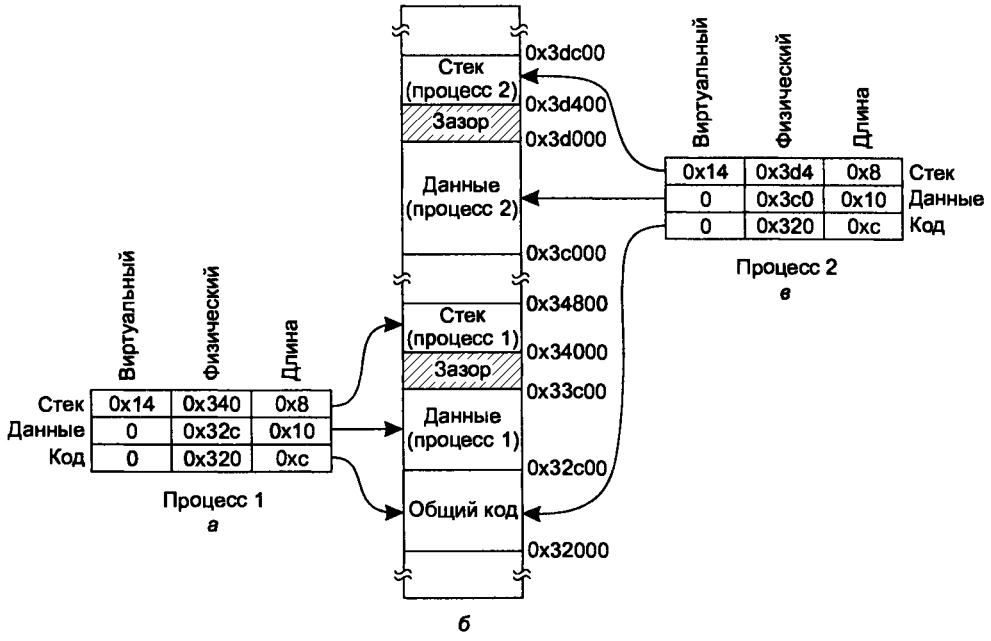


Рис. 4.33. *а* — карта памяти для раздельных адресных пространств кода и данных, как на предыдущем рисунке; *б* — распределение памяти после запуска второго процесса, выполняющего тот же код; *в* — карта памяти второго процесса

Помимо информации о сегментах, mproc хранит идентификатор (PID) самого процесса и его родителя, идентификаторы пользователя и группы (реальное и эффективное значения), информацию о сигналах и код возврата, если процесс уже завершился, но его родитель еще не завершил вызов wait.

Таблица свободных участков памяти, hole, заданная в файле alloc.c, является другой важнейшей таблицей менеджера памяти. Записи в этой таблице расположены в порядке возрастания адресов памяти. Промежутки между сегментами стека и данных зарезервированы для определенного процесса и не считаются незанятыми участками памяти, поэтому они не входят в эту таблицу. Каждая запись таблицы «дыр» содержит три поля: базовый адрес свободного блока памяти, в кликах, длину блока, тоже в кликах, и указатель на следующую запись в списке. Список является однонаправленным, другими словами, зная одну запись, легко найти любую следующую, но если необходимо найти предшествующую запись, придется перебирать список с самого начала.

Причина, по которой положение сегментов и свободных блоков измеряется в кликах, а не в байтах, проста: это более эффективно. В 16-битном режиме для записи адресов памяти используются 16-битные целые числа, что позволяется при размере клика 256 байт поддерживать до 16 Мбайт памяти. В 32-разрядном режиме таким образом может адресоваться до 2^{40} байт, что составляет 1024 Гбайт.

Основными действиями над списком «дыр» являются выделение блока заданного размера и освобождение ранее выделенного блока. При выделении блока

список просматривается в порядке возрастания адресов, пока не обнаруживается достаточно большой свободный блок. Затем «дыра» уменьшается на величину выделенного сегмента, или же, в том редком случае, когда размер «дыры» равен затребованному размеру блока, «дыра» вовсе исключается из списка. Это быстрый и простой механизм, но он страдает как от внутренней фрагментации (при выделении блока может быть впустую израсходовано до 255 байт, так как всегда выделяется целое число кликов), так и от внешней фрагментации.

Когда процесс завершает свою работу и удаляется из памяти, память, выделенная под данные и стек, возвращается в список свободных блоков. Если адресные пространства кода и данных процесса объединены, это означает, что возвращается вся занятая процессом память, поскольку отдельный блок под код здесь не предусмотрен. Если адресные пространства разделены и обнаруживается, что код процесса больше никем не используется, то занятая кодом память также высвобождается. При возврате блока в список «дыр» он, если это возможно, объединяется с соседними свободными блоками, соответственно, двух смежных дыр никогда не возникает. Таким образом, при работе системы постоянно меняются количество, положение и размер свободных участков памяти. Когда все пользовательские процессы завершатся, вся свободная память вновь будет готова к распределению. Она не обязательно будет образовывать один сплошной блок, так как физическая память может прерываться областями, недоступными даже операционной системе. Например, в IBM PC-совместимых системах доступна память ниже 640 Кбайт и выше 1 Мбайт, а промежуток между этими адресами занят ПЗУ и памятью, зарезервированной для операций ввода/вывода.

4.7.4. Системные вызовы fork, exit и wait

Когда создаются или уничтожаются процессы, необходимо выделять и освобождать память. Кроме того, нужно обновлять таблицу процессов, в том числе и те ее части, которые поддерживаются ядром и файловой системой. Эту деятельность координирует менеджер памяти. За создание процесса отвечает вызов fork, выполняющийся за несколько шагов. Эта последовательность такова.

1. Проверить, заполнена ли таблица процессов.
2. Попытаться выделить память для данных и стека дочернего процесса.
3. Скопировать содержимое данных и стека родительского процесса в память потомка.
4. Найти свободную ячейку в таблице процессов и скопировать в нее запись родительского процесса.
5. Поставить на учет карту памяти дочернего процесса в таблице процессов.
6. Выбрать для дочернего процесса PID.
7. Передать информацию о потомке ядру и файловой системе.
8. Сообщить ядру сведения о карте памяти потомка.
9. Отправить дочернему и родительскому процессам ответное сообщение.

Останавливать выполнение fork на полпути сложно и неудобно, поэтому менеджер памяти, чтобы всегда знать, есть ли свободные ячейки в таблице процессов, поддерживает счетчик существующих процессов. Если таблица еще не заполнена, делается попытка выделить память под данные и стек дочернего процесса. Для процессов с разделенными пространствами кода и данных запрашивается только память, достаточная для размещения стека и данных. Если этот шаг пройден успешно, fork гарантированно выполнится. Затем выделенная область памяти заполняется, в таблице процессов находится и заполняется ячейка нового процесса, для него выбирается PID, и другие части системы информируются о создании нового процесса.

Процесс полностью завершается по факту совокупности следующих событий: (1) процесс сам выполнил выход (или был завершен по сигналу), и (2) родительский процесс, чтобы выяснить, чем все закончилось, выполнил системный вызов wait. Процесс, прекративший выполнение или завершенный по сигналу, при условии, что его родитель еще не выполнил wait, попадает в своего рода замороженное состояние, такой процесс называют *зомби*. Он исключается из планирования, и сигнальный таймер у него отключается (если был включен), но он остается в таблице процессов. Память процесса при этом освобождается. В состоянии зомби процесс находится временно, и оно редко длится долго. Когда родительский процесс наконец выполняет wait, занятая ячейка в таблице процессов освобождается, и файловая система и ядро уведомляются об этом.

Проблема возникает в том случае, если родитель завершающегося процесса сам уже мертв. Если не предпринять никаких действий, потомок останется уже полноправным зомби. Поэтому таблицы загодя меняются так, чтобы процесс стал потомком процесса init. При запуске системы init считывает файл с информацией о всех терминалах /etc/ttytab и для обслуживания каждого терминала ответвляет от себя дочерний процесс. Затем он входит в состояние блокировки, ожидая уведомлений о завершении. Таким образом, осиротевшие зомби быстро добиваются.

4.7.5. Системный вызов exec

Когда с терминала поступает команда, оболочка ответвляет новый процесс, который выполняет запрошенную команду. Этим мог бы заняться один системный вызов, который бы одновременно решал задачи fork и exec, но они разделены по одной очень веской причине: чтобы упростить реализацию перенаправления ввода/вывода. Если стандартный ввод перенаправлен и оболочка выполняет ветвление, то потомок, перед тем как выполнить команду, закрывает стандартный ввод, а затем открывает новый. Таким образом, запущенный процесс наследует перенаправление стандартного ввода. То же относится и к стандартному выводу.

Системный вызов exec является самым сложным в MINIX. Он должен заместить текущий образ процесса новым и, в том числе, установить новый стек. Этапы выполнения этого системного вызова:

1. Проверить, является ли файл исполняемым.

2. Считать из заголовка файла размеры сегментов и общий требуемый объем памяти.
3. Узнать у выполнившего запуск процесса аргументы и переменные окружения.
4. Выделить участок памяти и освободить старую память.
5. Копировать в новый образ стек.
6. Записать в новый образ в памяти данные (и, возможно, код).
7. Проверить и, если они установлены, обработать биты setuid, setgid.
8. Подправить запись в таблице процессов.
9. Сообщить ядру, что процесс готов к запуску.

Каждый из шагов сам, в свою очередь, состоит из серии более мелких шагов, некоторые из них могут завершиться неудачей. Например, доступной памяти может не хватить для запуска процесса. Порядок, в котором производятся проверки, продуман так, чтобы гарантировать, что старый образ процесса в памяти остается до тех пор, пока не будет уверенности, что вызов exec завершится успехом. Это делается во избежание ситуации, когда новый образ сформировать невозможно, а к старому уже не вернуться. Как правило, вызов exec не передает управление обратно, но если вызов «провалился», процесс вновь получает управление и уведомляется об ошибке.

Некоторые из перечисленных выше шагов нуждаются в небольшом пояснении. Прежде всего, это вопрос о том, достаточно ли для нового процесса места. После того как выяснена потребность в памяти (с возможными проверками, есть ли уже запущенные процессы с тем же кодом), то, чтобы узнать, достаточно ли памяти, просматривается список свободных блоков. Это делается *до* того, как старая память освобождается, так как в противном случае, если памяти не хватает, вернуть старый образ было бы проблематично.

Тем не менее проверка чересчур строга. Иногда отклоняется вызов exec, который, фактически, мог бы быть выполнен. Например, предположим, что выполняющий exec процесс занимает 20 Кбайт и его текст не разделяют другие процессы. Далее, представим, что есть свободный блок объемом 30 Кбайт, а новый образ требует 50 Кбайт памяти. Выполняя проверку до освобождения памяти, мы обнаружим, что доступно только 30 Кбайт памяти и вызов не будет выполнен. Если бы память сначала высвобождалась, то вызов мог бы быть выполнен, в зависимости от того, сольются ли при освобождении памяти имеющийся свободный блок размером 30 Кбайт с новым, размером 20 Кбайт. Обрабатывать подобные ситуации лучше мог бы более сложный алгоритм.

Еще одно потенциальное улучшение — искать два свободных блока, один для сегмента кода и один для сегмента данных, если адресные пространства запускаемого процесса разделены. Сегменты не должны быть расположены непрерывно.

Более тонкий нюанс — умещается ли новый процесс в *виртуальном* адресном пространстве. Суть проблемы в том, что память выделяется не байтами, а 256-байтовыми кликами. Каждый клик должен целиком принадлежать одному сег-

менту и не может, например, быть наполовину занят данными, наполовину стеком, так как все управление памятью производится в кликах.

Чтобы стало очевиднее, как такое поведение приводит к проблемам, заметим, что на 16-битных системах (8088 и 80286) адресное пространство ограничено 64 Кбайт, что составляет 256 кликов. Представим теперь, что программе с раздельными адресными пространствами кода и данных требуется 40 000 байт под код, 32 770 байт под данные и 32 760 байт под стек. Тогда сегмент данных займет 129 кликов, из которых последний будет использоваться только частично (при этом он все равно целиком будет принадлежать сегменту). Под стек потребуется 128 кликов. Для данных и стека вместе нужно более 256 кликов, таким образом, они не смогут сосуществовать, хотя необходимое количество байтов умещается в виртуальной памяти (едва-едва). В теории, эта проблема относится ко всем машинам, у которых размер клика более одного байта, но на практике для процессоров класса Pentium она возникает исключительно редко, так как на таких машинах допустимы большие сегменты (более 4 Гбайт).

Другой важный вопрос — начальная установка стека. Библиотечный вызов, обычно используемый для выполнения **exec**, выглядит так:

```
execve(name, argv, envp);
```

Здесь **name** — указатель на имя загружаемого файла, **argv** ссылается на массив указателей на аргументы, а указатель **envp** содержит адрес массива указателей, ссылающихся на строки переменных окружения.

Достаточно просто реализовать **exec**, передав эти три указателя в сообщении менеджеру памяти и позволив ему самостоятельно извлечь имя файла и адреса двух массивов. Но для этого потребовалось бы работать со строками по одной, и на каждый аргумент пришлось отправлять как минимум одно сообщение задаче системы, а возможно, и больше, так как менеджер памяти не знает, какого размера каждая последующая порция.

Чтобы уменьшить накладные расходы, связанные со считыванием всех этих кусочков, была выбрана совершенно иная стратегия. Библиотечная процедура **execve** сама строит новый стек и передает менеджеру памяти его базовый адрес и размер. Создавать новый стек в пользовательском пространстве гораздо эффективнее, поскольку ссылки на аргументы будут локальными указателями, а не ссылками на другое адресное пространство.

Чтобы яснее понять этот механизм, рассмотрим пример. Когда пользователь вводит в оболочке команду

```
ls -l f.c g.c
```

оболочка интерпретирует ее и делает вызов библиотечной процедуры:

```
execve("/bin/ls", argv, envp);
```

Содержимое двух массивов указателей показано на рис. 4.34, *а*. Затем процедура **execve**, работая в пространстве пользователя, строит новый стек, как показано на рис. 4.34, *б*. В конечном итоге, стек при выполнении менеджером памяти системного вызова **exec** копируется без изменений.

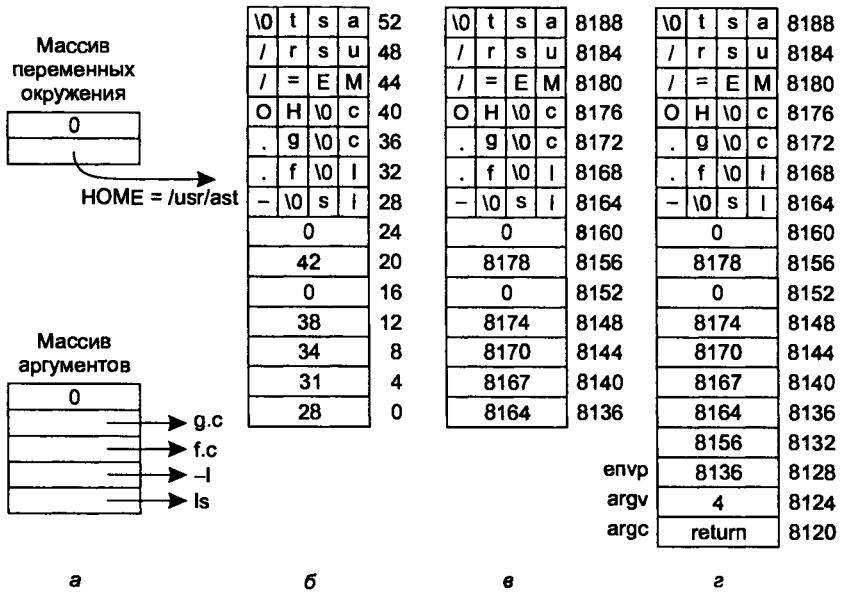


Рис. 4.34. а — массивы, передаваемые подпрограмме *execve*; *б* — стек, построенный *execve*; *в* — стек после перемещения менеджером памяти; *г* — стек, который процедура увидит в начале выполнения

Когда стек в конце концов попадает в пользовательский процесс, он не помещается по нулевому виртуальному адресу. Вместо этого он записывается в конец выделенной области памяти, размер которой определяется по заголовку исполняемого файла. Для примера, предположим, что общий размер составляет 8192 байт, то есть последний доступный программе байт имеет адрес 8191. Тогда менеджер памяти так располагает в стеке указатели, что стек будет выглядеть, как показано на рис. 4.34, *в*.

Когда системный вызов **exec** завершится и программа начнет работу, стек придет в состояние, показанное на рис. 4.34, *в*, а указатель стека станет содержать значение 8136. Тем не менее остается другая нерешенная проблема. Главная процедура исполняемого файла, скорее всего, объявлена примерно так:

```
main (argc, argv, envp).
```

Поскольку рассматривается компилятор C, **main** является всего лишь одной из функций. Компилятор не знает о ее особой роли, поэтому строит ее код так, как если бы аргументы передавались ей по стандартному соглашению вызова языка C, когда последний аргумент идет в стеке первым. Поскольку значениями аргументов являются одно целое число и два указателя, ожидается, что они займут три слова, предшествующие адресу возврата. Естественно, показанный на рис. 4.34, *в* стек выглядит совершенно иначе.

Решение в том, чтобы выполнение программы не начиналось с **main**. Вместо нее управление сначала получает маленькая стартовая ассемблерная подпро-

грамма crtso, которую компоновщик помещает в сегмент кода по «нулевому» адресу. Эта подпрограмма называется runtime, и ее назначение в том, чтобы поместить в стек три требующихся слова и вызвать main, пользуясь стандартным соглашением вызова. Благодаря этой хитрости main может думать, что она вызвана обычном образом (хотя в действительности это не трюк, она *действительно* вызывается совершенно обычно).

Если программист в конце кода main не сделал вызов exit, то после ее завершения управление передается обратно в C runtime. Опять же, с точки зрения компилятора, main — обычная функция, и для возврата из нее компилятор генерирует стандартный код. Большая часть кода 32-битной версии crtso приведена в листинге 4.1. Комментарии поясняют выполняемые действия. Не вошел в этот листинг лишь код, загружающий из стека помещенные туда регистры, и несколько строк, устанавливающие флаг наличия/отсутствия математического сопроцессора.

Листинг 4.1. Ключевая часть C runtime, стартовой подпрограммы

```
push ecx                                ; push environ
push edx                                ; push argv
push eax                                ; push argc
call _main                              ; main(argc, argv, envp)
push eax                                ; push exit_status
call _exit
hlt                                     ; Если вызов exit не удался, вызывается прерывание
```

4.7.6. Системный вызов brk

При помощи библиотечных подпрограмм brk и sbrk изменяется положение верхней границы сегмента данных. Первая из них берет абсолютное значение нового адреса (в байтах) и передает его системному вызову brk. Вторая вычисляет положительное или отрицательное приращение текущего положения, вычисляет новый размер и вызывает brk. Отдельного системного вызова sbrk в действительности нет.

Интересен вопрос: как sbrk определяет текущий размер, чтобы вычислить новый? Ответом на него является переменная brksize, в которой всегда хранится размер и откуда brk и sbrk всегда могут его считать. Эта переменная инициализируется генерируемым компилятором символом, определяющим либо суммарный размер кода и данных (общее адресное пространство), либо только размер данных (раздельные адресные пространства). Имя и даже само существование такого символа привязаны к компилятору, поэтому вы не найдете его обозначение ни в одном из заголовочных файлов исходных кодов. Он задается в библиотеке, в файле brksize.s. Где именно он расположен, зависит от системы, но он будет в том же каталоге, что и crto.s.

Для менеджера памяти несложно выполнить вызов brk. Все, что ему нужно сделать, — это проверить, что сегмент умещается в адресном пространстве, обновить таблицы и уведомить ядро.

4.7.7. Обработка сигналов

В главе 1 сигналы были определены как механизм передачи информации процессу, который не обязательно ждет ввода. Задается набор сигналов, и у каждого сигнала есть действие по умолчанию: либо завершить процесс, которому сигнал адресован, либо игнорировать сигнал. Если бы других альтернатив не было, обработку сигналов было бы просто понять и реализовать. Но при помощи системных вызовов процессы способны менять это поведение. Процесс может потребовать, чтобы любой сигнал (за исключением особого случая — SIGKILL) игнорировался. Более того, процесс может «поймать» сигнал, предписав, чтобы вместо действия по умолчанию был вызван указанный им *обработчик сигнала* (опять же, это не относится к SIGKILL). Таким образом, с точки зрения программиста есть два этапа работы с сигналами: подготовительная фаза, когда определяется ответная реакция на будущий сигнал, и ответная — когда сигнал генерируется и обрабатывается. Ответным действием может быть выполнение собственной подпрограммы-обработчика. В действительности, есть и третья фаза. Когда пользовательский обработчик завершается, специальный системный вызов восстанавливает нормальную работу получившего сигнал процесса. Программисту об этой третьей фазе знать не нужно, он пишет обработчики сигналов как обычные функции, а заботу о вызове и завершении обработчиков и управлении стеком берет на себя операционная система.

В подготовительной фазе программа вправе в любой момент изменить реакцию на сигнал при помощи нескольких системных вызовов. Самый общий из них — вызов sigaction, посредством которого можно указать, чтобы сигнал игнорировался, обрабатывался (при этом вместо действия по умолчанию для такого сигнала выполняется некоторый заданный пользователем код из самого процесса), или же восставить ответную реакцию по умолчанию. При помощи другого системного вызова, sigprocmask, сигнал можно заблокировать, тогда он будет поставлен в очередь и обработан только тогда, когда процесс разблокирует сигналы этого типа. Эти вызовы можно делать в любой момент даже из самой функции-обработчика. В MINIX действия подготовительной стадии осуществляются исключительно в менеджере памяти, так как все необходимые структуры данных расположены в его части таблицы процессов. Для каждого процесса в этой таблице имеется несколько переменных типа sigset_t, в которых за каждый сигнал отвечает определенный бит. Одна из переменных хранит информацию о том, какие сигналы необходимо игнорировать, другая — какие сигналы обрабатываются и т. д. Кроме того, у каждого процесса есть массив структур sigaction, по одной на каждый сигнал. В этой структуре присутствует переменная, хранящая адрес пользовательского обработчика сигнала, а также дополнительное поле типа sigset_t, где запоминается информация о сигналах, заблокированных во время исполнения другого обработчика. В поле адреса обработчика вместо адреса пользовательской функции могут храниться специальные данные, обозначающие, что данный сигнал должен быть игнорирован или обработан по умолчанию.

В генерации сигнала участвуют многие компоненты операционной системы MINIX. Начинается все с менеджера памяти, который решает, какой процесс

должен при помощи упомянутых выше структур получить сигнал. Если данный сигнал обрабатывается, его необходимо доставить процессу. Для этого требуется сначала сохранить информацию о процессе, чтобы после обработки восстановить его нормальное состояние. Эта информация сохраняется в стеке процесса, причем предварительно делается проверка наличия достаточного места в стеке. Стеком заведует менеджер памяти, который и выполняет этот контроль, а затем, чтобы поместить информацию в стек, вызывает задачу системы, которая также изменяет значение счетчика команд процесса, чтобы выполнился код обработчика. Когда обработчик завершается, делается системный вызов sigreturn. Посредством этого вызова менеджер памяти и ядро участвуют в восстановлении контекста — сигналов и регистров процесса, возвращая его к обычному режиму работы. Если сигнал не обрабатывается, выполняется действие по умолчанию. При этом, если необходимо сбросить дамп ядра (то есть записать образ процесса в файл для последующего анализа под отладчиком), может быть затронута файловая система, а если процесс должен быть прекращен, затрагивается как менеджер памяти, так и файловая система и ядро. Наконец, если сигнал адресован группе процессов, менеджер памяти может предписать повторить эти действия несколько раз.

Сигналы, которые знает MINIX, определяются в файле /usr/include/signal.h, согласно стандарту POSIX. Они перечислены в табл. 4.4. В MINIX объявлены все требуемые POSIX сигналы, но пока что не все они поддерживаются. Например, стандартом определен набор сигналов для управления задачами, при помощи которых можно переводить задачу в фоновый режим и возвращать ее обратно. В MINIX не поддержано управление задачами, но программы, которые генерируют такие сигналы, могут быть перенесены под MINIX. Неподдерживаемые сигналы будут просто игнорироваться. Кроме того, MINIX определяет ряд сигналов, не имеющихся в POSIX, а также несколько синонимов для POSIX-имен, с целью обеспечить совместимость с более старым исходным кодом.

Сигналы генерируются либо ядром, либо при помощи системного вызова kill. Ядро MINIX всегда может генерировать сигналы SIGINT, SIGQUIT и SIGALARM, возможность генерировать другие сигналы зависит от аппаратной поддержки. Например, процессоры 8086 и 8088 не умеют обнаруживать недопустимые инструкции, а процессоры 286 и старше при попытке выполнить такую инструкцию уже вызывают прерывание. Это обусловлено возможностями аппаратного обеспечения. Чтобы в ответ на прерывание генерировать сигнал, разработчики операционной системы должны заготовить для этого код. В главе 2 мы видели, что в файле kernel/exception.c как раз содержится нужный код, для разных сочетаний условий. Таким образом, когда MINIX работает на машине с процессором 286 или старше, недопустимая инструкция приведет к появлению сигнала SIGKILL, но на компьютере с процессором 8088 такого никогда не произойдет.

То, что оборудование способно генерировать аппаратное прерывание при возникновении некоторой ситуации, не означает, что разработчики операционной системы могут во всем понадеяться на эту возможность. Так, все процессоры Intel, начиная с 286, распознают несколько типов нарушений целостности памяти, вызывающих исключения. Код в файле kernel/exception.h преобразует эти исключения в сигналы SIGSEGV.

Таблица 4.4. Сигналы MINIX, регламентированные POSIX. Знак (*) означает, что сигнал зависит от аппаратной поддержки, сигналы с пометкой (M) не исходят из POSIX и введены в MINIX для поддержания совместимости со старым кодом. В таблицу не попали несколько устаревших имен и синонимов

Сигнал	Описание	Источник
SIGHUP	Отбой	Системный вызов kill
SIGINT	Прерывание	Ядро
SIGQUIT	Завершение	Ядро
SIGILL	Недопустимая инструкция	Ядро (*)
SIGTRAP	Прерывание трассировки	Ядро (M)
SIGABRT	Аномальное завершение	Ядро
SIGFPE	Ошибка при вычислениях с плавающей запятой	Ядро (*)
SIGKILL	Принудительное завершение (сигнал не может быть игнорирован или обработан)	Системный вызов kill
SIGUSR1	Задаваемый пользователем сигнал № 1	Не поддерживается
SIGSEGV	Нарушение целостности памяти	Ядро (*)
SIGUSR2	Задаваемый пользователем сигнал № 2	Не поддерживается
SIGPIPE	Запись в канал, из которого никто не читает	Ядро
SIGALRM	Сигнал таймера, истечение тайм-аута	Ядро
SIGTERM	Сигнал завершения программы	Системный вызов kill
SIGCHLD	Дочерний процесс завершил работу или остановился	Не поддерживается
SIGCONT	Продолжить работу после останова	Не поддерживается
SIGSTOP	Остановить выполнение процесса	Не поддерживается
SIGTSTP	Интерактивный сигнал останова	Не поддерживается
SIGTTIN	Фоновый процесс пытается выполнить ввод	Не поддерживается
SIGTTOU	Фоновый процесс пытается выполнить вывод	Не поддерживается

Нарушениям аппаратных границ стека и границ других сегментов соответствуют разные типы исключений, в расчете на различную их обработку. Тем не менее, в силу особенностей работы MINIX с памятью, не все возникающие нарушения могут быть обнаружены. Аппаратные регистры задают базовый адрес сегмента и его длину. В MINIX базовый адрес аппаратного сегмента стека совпадает с базовым адресом аппаратного сегмента данных, но аппаратно заданный размер сегмента данных превышает контролируемую программно границу. Другими словами, контролируемый аппаратно размер сегмента данных соответствует ситуации, когда стек уменьшился до нуля. Аналогично, заданный аппаратно объем стека соответствует нулевому объему данных. Таким образом, хотя некоторые из нарушений и поддаются аппаратному обнаружению, наиболее вероятная ошибка — повреждение области данных из-за переполнения стека — не де-

тектируется. Это следствие того, что с точки зрения аппаратных регистров и таблиц дескрипторов сегменты данных и стека пересекаются.

Потенциально возможно добавить в ядро код, который бы проверял регистры процесса всякий раз после того, как процесс получает управление, и в случае нарушения программно заданных границ сегментов данных или стека генерировал бы сигнал SIGSEGV. Но не совсем понятно, стоит ли это делать, ведь аппаратные ловушки в силах обнаружить сбой по доступу сразу же после того, как он произошел. Программная же проверка может случиться спустя много тысяч инструкций после момента переполнения, когда обработчик прерывания уже мало на что годится и восстановление неосуществимо.

Где бы ни брали свое начало сигналы, менеджер памяти обрабатывает их одинаково. Сначала для каждого процесса-получателя делается набор проверок, с целью убедиться, можно ли передать ему сигнал. Один процесс может передавать другому сигнал в том случае, если первый принадлежит суперпользователю, либо если его эффективный UID равен реальному или эффективному UID второго. Но существует еще несколько условий, вмешивающихся в отправку сигнала. Например, нельзя передавать сигнал зомби. Также процессу нельзя передавать сигнал, если он явно сделал вызов sigaction, чтобы игнорировать возможный сигнал, или вызвал sigprocmask, чтобы его блокировать. Блокировка сигнала и пренебрежение им — разные вещи. Заблокированный сигнал запоминается и передается процессу, когда тот снимет блокировку (если он ее вообще снимет). Наконец, если у процесса-получателя недостаточно места в стеке, этот процесс принудительно завершается.

Когда все тесты пройдены, сигнал можно отправлять. Если процесс не сделал ничего, чтобы обработать сигнал, то и никакой информации передавать ему не требуется. В таком случае менеджер памяти выполняет обработку сигнала по умолчанию, обычно это означает либо завершение процесса, либо сброс дампа памяти в файл. Несколько сигналов по умолчанию игнорируются. Сигналы, которые в табл. 4.4 обозначены как не поддерживаемые, рекомендованы стандартом POSIX, но в MINIX игнорируются.

Обработка сигнала процессом заключается в исполнении заданного пользователем кода обработчика, адрес обработчика сохранен в структуре sigaction в таблице процессов. В главе 2 мы видели, как в кадр стека в пределах таблицы процессов записывается информация, необходимая для восстановления процесса после его прерывания. Путем модификации кадра стека процесса-получателя сигнала можно добиться того, чтобы в следующий раз, когда процесс получит управление, он начал бы исполнять код обработчика сигнала. Посредством манипуляций с собственным стеком процесса, в пользовательском пространстве, делается так, чтобы по завершении обработчика произошел системный вызов sigreturn. Явно эта подпрограмма из пользовательского кода никогда не вызывается. Вызов исполняется за счет того, что ядро помещает его адрес в стек, то есть здесь это адрес возврата, на который осуществляется переход после завершения обработчика. Вызов sigreturn восстанавливает исходный кадр стека получившего сигнал процесса, чтобы тот мог продолжить свое выполнение с момента, на котором его застал сигнал.

Теперь, несмотря на то что финальная стадия отправки сигнала происходит в задаче системы, пора подытожить то, что мы изучили. Когда сигнал должен быть обработан, необходимо действие, во многом сходное с обычным переключением контекстов, которое происходит, когда один процесс переводится в ожидание, а вместо него начинает выполняться другой. Но в таблице процессов есть только одно место, где можно сохранить все регистры процессора, необходимые для восстановления исходного состояния процесса. Достаточно ли этого? Для ответа на вопрос посмотрим на рис. 4.35, *а*. Здесь изображен в упрощенном виде стек процесса и часть его записи в таблице процессов в тот момент, когда процесс только что был приостановлен по факту прерывания. На время бездействия содержимое всех регистров копируется в запись stackframe этого процесса в части таблицы процессов, принадлежащей ядру. Данная ситуация будет соответствовать времени генерации сигнала, пока источник и приемник сигнала — разные процессы.

В процессе подготовки к обработке сигнала содержимое кадра стека копируется из таблицы процессов в собственный стек процесса в виде структуры sigcontext, тем самым сохраняется информация о состоянии. Затем в стек помещается структура sigframe, содержащая информацию, которая станет нужной sigreturn после завершения обработчика. Кроме того, в ней хранится и адрес самой библиотечной подпрограммы sigrtetrun, это ret addr1, и еще один адрес возврата, ret addr2, — адрес, с которого необходимо продолжить выполнение прерванной программы. Как мы увидим в дальнейшем, при нормальной работе второй адрес использоваться не должен.

Хотя обработчик и пишется программистом как обычная функция, он не вызывается инструкцией call. Чтобы началось исполнение кода обработчика, изменяется значение поля счетчика команд в кадре стека в таблице процессов, в результате, когда restart переводит процесс в состояние исполнения, начинает выполняться обработчик. На рис. 4.35, *б* показана ситуация, когда все эти приготовления завершены и обработчик уже занимается своим делом. Ну, а поскольку обработчик все-таки является обычной процедурой, когда он завершает работу, из стека извлекается адрес ret addr1 и запускается sigreturn.

Ситуация, когда исполняется sigreturn, показана на рис. 4.35, *в*. Оставшаяся часть структуры sigframe используется как локальные переменные sigreturn. Одно из действий этого системного вызова направлено на такое изменение собственного указателя стека, когда при обычном возврате осуществляется переход на адрес ret addr2. Но в действительности sigreturn не завершается как обычные функции. Он, как и другие системные вызовы, передает право решать, какой процесс станет активным, планировщику в ядре. В конечном итоге, в какой-то момент времени получивший сигнал процесс будет выбран на выполнение. Оно продолжится с этого адреса, продублированного в оригинальном кадре стека процесса. Данный адрес помещается в стек для того, чтобы отладчик, проходя по программе, не испытывал проблем со стеком при трассировке обработчика сигнала. Благодаря таким манипуляциям на каждом этапе выполнения стек выглядит как обычный стек процесса, с собственными локальными переменными после адреса возврата.

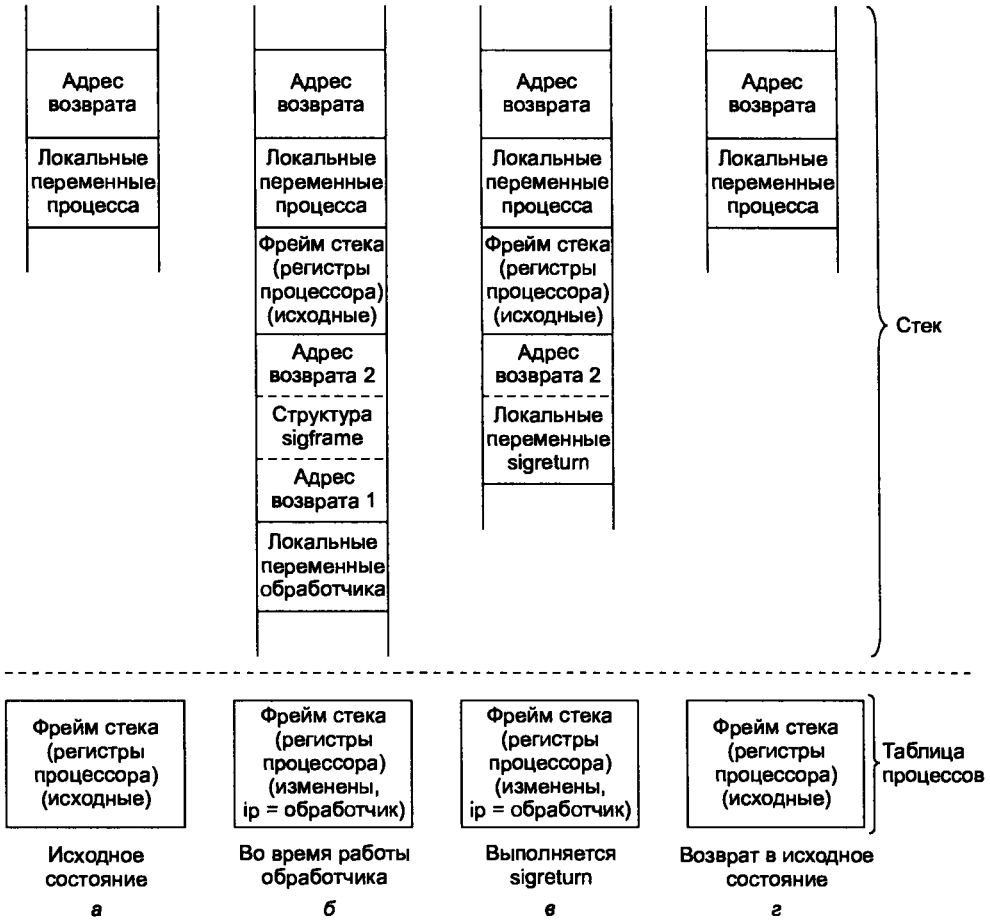


Рис. 4.35. Стек процесса (сверху) и кадр стека (снизу), соответствующие различным фазам обработки сигнала: *а* — состояние, в котором процесс приостанавливается; *б* — состояние на начало исполнения обработчика; *в* — состояние во время исполнения sigretrun; *г* — состояние после того, как sigretrun завершил работу

Главное, что должен сделать вызов sigreturn, — это вернуть все на круги своя, к моменту получения сигнала. Важнее всего восстановить кадр стека процесса, для чего и используется его копия, сохраненная в собственном стеке процесса. В результате после завершения sigreturn процесс возвращается в состояние, соответствующее рис. 4.35, *г*.

По умолчанию большинство сигналов приводят к завершению процесса. Заботится об этом менеджер памяти, завершающий процесс, если сигнал по умолчанию не игнорируется и если процесс-приемник не требовал блокировать, обрабатывать или игнорировать данный сигнал. Если завершения процесса ожидает его родитель, процесс завершается, и соответствующая информация удаляет-

ся из таблицы процессов. Если родитель пренебрегает своим долгом, процесс становится зомби. Кроме того, для некоторых типов сигналов (например, для SIGQUIT) менеджер памяти записывает в текущий каталог дамп памяти процесса.

Легко может случиться так, что получивший сигнал процесс находится в состоянии блокировки, скажем, читает с терминала при помощи read, когда на терминал не поступает данных. Если процесс не указал, что сигнал должен обрабатываться, он просто завершается обычным образом. Но если сигнал обрабатывается, закономерен вопрос: что должен делать процесс после того, как прерывание сигнала обработано. Должен ли он вернуться в состояние ожидания или продолжить работу со следующей инструкции?

В MINIX делается следующее: системный вызов завершается с кодом возврата EINTR, поэтому процесс может понять, что он был прерван сигналом. Узнать о том, что процесс заблокирован на системном вызове, не так просто. Менеджеру памяти придется спрашивать об этом файловую систему.

Такое поведение предполагается стандартом POSIX, который позволяет вызову read возвращать байты, считанные до прихода сигнала, хотя и не жестко предписано. Кроме того, возврат с кодом EINTR позволяет легко реализовать таймер и обработать SIGALARM, например, чтобы завершить операцию login и повесить трубку, если пользователь некоторое время (тайм-аут) не отвечает. При помощи синхронных часов эта задача решается с меньшими издержками, но синхронный таймер — новшество MINIX, и не столь поддается переносу, как сигналы. Также он доступен только серверам, обычные пользовательские процессы не вправе им пользоваться.

4.7.8. Прочие системные вызовы

Менеджер памяти отвечает еще за несколько простых системных вызовов. Две библиотечные функции, getuid и geteuid, пользуются одним и тем же системным вызовом getuid, который в ответном сообщении возвращает оба значения. Аналогично, системный вызов getuid также возвращает как реальное, так и действующее значения идентификаторов, нужные соответственно функциям getgid и getegid. Подобно работает и вызов getpid, возвращающий идентификатор самого процесса и его родителя, а при помощи вызовов setuid и setgid можно устанавливать как реальное, так и эффективное значения сразу, одним вызовом. В этой группе есть два дополнительных системных вызова, getpgrp и setsid. Первый возвращает идентификатор группы процессов, а второй устанавливает его равным текущему идентификатору процесса (PID). Эти семь функций — самые простые системные вызовы в MINIX.

Вызовы ptrace и reboot также выполняются в менеджере памяти. Первый из них помогает отлаживать программы. Второй оказывает влияние на многие аспекты системы. Первым действием этот вызов отправляет сигналы, чтобы завершить все процессы, кроме init, поэтому его код помещен именно в менеджер памяти. Чтобы завершить работу после того, как отправлены сигналы, в процесс выполнения вовлекаются файловая система и системная задача.

4.8. Реализация управления памятью в MINIX

Вооружившись общим пониманием того, как работает менеджер памяти, давайте обратимся к самому коду. Менеджер памяти полностью написан на C, его код прямолинеен и снабжен значительным количеством комментариев, посему по большей части мы не будем слишком глубоко вдаваться в детали. Сначала мы изучим заголовочные файлы, затем главную программу, а потом файлы с кодом описанных ранее системных вызовов.

4.8.1. Заголовочные файлы и структуры данных

В каталоге исходных файлов менеджера памяти есть несколько заголовочных файлов, имена которых совпадают с именами файлов ядра. Эти же имена мы встретим еще раз при изучении файловой системы. Такие файлы имеют «одноименные» функции (в своем контексте). Параллельная структура была выбрана для того, чтобы упростить понимание устройства MINIX в целом. Помимо означенных, к менеджеру памяти относятся несколько заголовочных файлов с уникальными именами. Как и в других частях системы, место для хранения глобальных переменных выделяется в файле table.c. Его рассмотрением, а также сопутствующих заголовочных файлов мы займемся в этом разделе.

Как и у всех остальных основополагающих компонентов системы MINIX, у диспетчера памяти есть свой главный заголовочный файл, mmh.h. Он включается в каждый файл с кодом и, в свою очередь, включает в себя все общесистемные заголовочные файлы из каталога /usr/include и его подкаталогов, нужные каждому объектному модулю. С ним присоединяется большая часть файлов, включаемых в kernel/kernrl.h, а также собственные версии const.h, type.h, proto.h и glo.h.

Файл const.h задает ряд необходимых менеджеру памяти констант. Строка

```
#define printf printk
```

переопределяет имя printf так, что вызовы этой функции будут при компиляции превращаться в вызовы printk. Эта функция аналогична той, что мы видели в ядре, и объявлена она по той же причине: чтобы менеджер памяти мог выводить отладочные сообщения и сообщения об ошибках, не обращаясь за помощью к файловой системе.

Файл type.h в текущей версии не используется и содержит только скелет, для того лишь, чтобы структура менеджера памяти была такой же, как и прочих частей системы. Файл proto.h предназначен для того, чтобы в одном месте собрать прототипы всех необходимых менеджеру памяти функций.

Глобальные переменные менеджера памяти декларируются в файле glo.h. Здесь применен тот же самый трюк с EXTERN, что и в ядре. А именно, макрос EXTERN разворачивается в ключевое слово extern во всех файлах, за исключением table.c, где он порождает пустую строку. В результате в файле table.c резервируется память под хранение глобальных переменных.

Первая из глобальных переменных, mp, является указателем на структуру mproc. Эта структура описывает ту часть таблицы процессов, которая принадлежит менеджеру памяти, а переменная mp ссылается на процесс, системный вызов которого обрабатывается в текущий момент. Вторая переменная, dont_reply, по прибытии каждого запроса инициализируется значением FALSE, но если в процессе обработки запроса обнаруживается, что отправлять ответное сообщение не нужно, ей может быть присвоено значение TRUE. В качестве примера вызова, при выполнении которого не требуется отправлять ответное сообщение, можно привести успешный вызов exec. Третья переменная, procs_in_use, служит для подсчета имеющихся в текущий момент процессов, эта цифра помогает выяснить, выполним ли вызов fork.

Буферы сообщений mm_in и mm_out предназначены соответственно для запросов и ответных сообщений. Переменная who содержит индекс текущего процесса. Она связана с переменой mp следующим соотношением:

```
mp=&mproc[who];
```

Когда получен системный вызов, его номер извлекается из сообщения и помещается в переменную mm_call.

Три переменные, err_code, result2 и resp_ptr, служат для хранения значений, возвращаемых в ответном сообщении процессу, сделавшему вызов. Важнейшей из этой тройки для нас является переменная err_code, которая, если нет ошибки, устанавливается в значение OK. Две последние переменные нужны тогда, когда обнаруживается ошибка. Если процесс завершается аномально, MINIX сбрасывает в файл образ процесса. Имя этого файла задается переменной core_name, а битовая карта core__sset описывает, какие сигналы должны приводить к дампу памяти.

Та часть таблицы процессов, которой заведует менеджер памяти, описана в следующем файле, mproc.h. По большей части, назначение полей этой таблицы объясняется комментариями. Несколько полей связаны с обработкой сигналов. Поля mp_ignore, mp_catch, mp_sigmask, mp_sigmask2 и mp_sigpending представляют собой битовые карты, в которых каждый бит означает один из сигналов, разрешенный к отправке процессу. Эти поля имеют тип sigset_t, являющийся 32-разрядным целым числом, следовательно, MINIX может поддерживать до 32 сигналов, но сейчас определено только 16 сигналов, причем сигналу № 1 соответствует наименее значимый (самый правый) бит карты. Впрочем, согласно POSIX, требуются специальные функции для добавления и удаления сигналов в эти наборы, поэтому при манипуляциях с битовыми картами программисту не нужно вдаваться в такие детали. А вот массив mp_sigact важен для обработки сигналов. В нем имеется по одной структуре типа sigaction (файл include/signal.h) на каждый возможный тип сигнала. Эта структура составлена из трех полей:

- sa_handler — определяет, будет ли для сигнала выполнено действие по умолчанию, специальное действие по обработке, или же сигнал будет игнорирован;
- sa_mask — является битовой картой, в которой отмечается, какие сигналы были заблокированы во время выполнения пользовательского обработчика;
- sa_flags — содержит набор флагов сигнала.

Благодаря массиву mp_sigact обеспечивается большая гибкость при обработке сигналов.

Поле mp_flags, как показано в конце файла, необходимо для хранения разнообразных битовых сочетаний. Хранимое в нем — это что-либо беззнаковое целое, 16-разрядное для самых старых процессоров и 32-разрядное для процессоров 386 и старше. Поле используется далеко не в меру своей емкости, на машинах 8088 задействованы только девять из битов.

Последнее поле таблицы процессов называется mp_procargs. Когда запускается новый процесс, строится стек, подобный показанному на рис. 4.35, и указатель на массив argv нового процесса сохраняется в этой переменной. Например, на рис. 4.35 в поле будет сохранено значение 8164, благодаря чему, пока выполняется команда ls, команда ps может вывести содержимое командной строки:

```
ls -l f.c g.c
```

Следующий файл, param.h, содержит макросы, помогающие формировать сообщения для многих системных вызовов. Кроме того, здесь же описаны четыре макроса для формирования ответных сообщений. Когда в любом файле, куда включен param.h, появляется такое выражение:

```
k = pid;
```

перед компиляцией препроцессор преобразует его следующим образом:

```
k = mm_in.m1_i1;
```

Давайте еще раз взглянем на файл table.c. При его компиляции получается объектный файл, в котором выделяется место для хранения глобальных переменных и структур, объявленных с директивой EXTERN. Такие структуры мы видели в glo.h и mproc.h. Благодаря тому, что в этом файле присутствует выражение

```
#define _TABLE
```

макрос EXTERN разворачивается в пустую строку. Это тот же самый механизм, который мы могли наблюдать в коде ядра.

Еще один важный элемент файла table.c — массив call_vec. С его помощью номер системного вызова преобразуется в адрес выполняющей его функции, номер вызова играет роль индекса в массиве. Если в сообщении указан несуществующий номер системного вызова, управление передается функции no_sys, которая просто возвращает код ошибки. Несмотря на то что при объявлении массива call_vec применяется макрос _PROTOTYPE, в действительности это не описание прототипа, а описание инициализированного массива. Но так как он же массив указателей на функции, проще всего получить код, компилируемый как классическим (Керниган и Ричи), так и стандартным компилятором C при помощи макроса _PROTOTYPE.

4.8.2. Основная программа

Менеджер памяти компилируется и компонуется независимо от ядра и файловой системы. Следовательно, у него имеется своя собственная главная процедура,

которая запускается после того, как ядро закончит свою инициализацию. Она расположена в файле main.c. В ней менеджер памяти сначала выполняет собственную инициализацию (функция mm_init), а затем входит в цикл. В этом цикле сначала, чтобы дождаться входящего сообщения, делается вызов get_work. Затем вызывается одна из функций do_XXX, адрес которой берется из массива call_vec, а далее, при необходимости, отправляется ответное сообщение. Такая конструкция должна быть вам уже знакома, по тому же принципу работают задачи ввода/вывода.

Соответственно, две следующие функции, get_work и reply, отвечают за реальные прием и отправку сообщений.

Последняя процедура в main.c — mm_init, она инициализирует менеджер памяти. После того как система заработала, эта процедура больше не нужна. При помощи вызова get_map в mm_init запрашивается информация об использовании памяти ядром. В следующем далее цикле заполняются все записи задач и серверов в таблице процессов, после чего подготавливается запись, соответствующая процессу init. Затем менеджер памяти ждет сообщения от файловой системы. Как уже говорилось при обсуждении тупиков, это единственный случай, когда файловая система отправляет сообщение без предварительного запроса. Сообщение говорит о том, сколько памяти было задействовано под RAM-диск. Дальше при помощи вызова mem_init инициализируется список свободных блоков памяти («дыр»), и вот теперь все готово к нормальной работе с памятью. Последний вызов также присваивает значения переменным total_clicks и free_clicks, после чего менеджер памяти может напечатать сообщение с информацией об общем объеме памяти, использовании памяти ядром, размере RAM-диска и объеме свободной памяти. Сделав это, менеджер памяти отправляет файловой системе ответ, выводя ее из состояния останова. В завершение задаче памяти передается адрес принадлежащей ядру части таблицы процессов, для того чтобы без проблем пользоваться командой ps.

4.8.3. Системные вызовы fork, exit и wait

Системные вызовы fork, exit и wait реализуются процедурами do_fork, do_mm_exit и do_wait соответственно, из состава файла forkexit.c. Процедура do_fork руководствуется последовательностью действий, перечисленных в разделе 4.7.4. Обратите внимание, что второй вызов procs_in_use резервирует для суперпользователя несколько последних ячеек в таблице процессов. При вычислении необходимой потомку памяти в сумму включается промежуток между данными и стеком, но не сегмент кода. Так делается потому, что сегмент кода либо разделяемый, либо, если адресные пространства кода и данных процесса объединены, его размер равен нулю. После того как нужный объем памяти подсчитан, чтобы получить ее, делается вызов alloc_mem. Если память успешно выделена, базовые адреса родителя и потомка преобразуются из кликов в байты и, чтобы выполнить копирование, вызывается sys_copy, которая отправляет сообщение задаче системы.

Теперь, когда память предоставлена, в таблице процессов ищется свободная ячейка. Более ранняя проверка, затрагивающая переменную procs_in_use, гаран-

тирует, что такая ячейка найдется. Найденная ячейка заполняется, для этого в нее сначала копируются данные родительского процесса, а затем обновляются поля mp_parent, mp_flags, mp_seg, mp_exitstatus и mp_sigstatus. Некоторые из этих полей требуют специальной подготовки. Так, в поле mp_flags обнуляется бит TRACED, поскольку потомок не наследует от родителя состояние трассировки. Поле mp_seg является массивом, элементы которого соответствуют сегментам кода, данных и стека, и если обнаруживается, что у процесса адресные пространства разделены, ячейка, соответствующая сегменту кода, не меняется и продолжает указывать на код родительского процесса.

На следующем шаге процессу назначается идентификатор — PID. В выборе PID система отталкивается от значения переменной next_pid, но тем не менее в обозримой вероятности не исключается одна проблема. После того как некому очень долгоживущему процессу назначен идентификатор (скажем, 20), может быть создано и уничтожено более 30 000 процессов, в результате чего next_pid вновь примет значение 20. Назначение процессу занятого идентификатора привело бы к сбою (представим, что позже кто-нибудь попытается отправить процессу под номером 20 сигнал), поэтому, чтобы удостовериться, что выбранный PID не занят, сканируется вся таблица процессов.

Вызовы sys_fork и tell_fs информируют ядро и файловую систему о рождении нового процесса, чтобы они могли обновить свои структуры таблицы процессов. (Все процедуры, имена которых начинаются с sys_, служат для отправки задаче системы сообщений, запрашивающих различные услуги согласно табл. 3.20.) Создание или убиение процесса всегда инициируется менеджером памяти и только потом в число участников этого таинства допускаются ядро и файловая система.

Ответное сообщение процессу-потомку отправляется точно в конце кода do_fork. Ответ родителю, содержащий идентификатор нового процесса, посылается из главного цикла в main, как обычный ответ на запрос.

Следующим системным вызовом, который выполняется менеджером памяти, является exit. Вызов принимает процедура do_mm_exit, но большую часть черновой работы делает mm_exit, несколькими строками позже. Причина такого разделения труда в том, что функция mm_exit также востребована, когда необходимо позаботиться о процессе, завершаемом по сигналу. Действия в этом случае те же самые, но другие аргументы, и такое разделение делается, по сути, для удобства.

Прежде всего mm_exit останавливает таймер, если у процесса он активен. Затем файловой системе и ядру сообщается, что этот процесс более не может быть запущен на выполнение. Вызов функции sys_xit отправляет задаче системы сообщение, по получении которого она помечает процесс, исключая его из планирования. Затем освобождается память. Функция find_share определяет, разделяется ли сегмент кода с другими программами. Если нет, сегмент кода освобождается вызовом free_mem. Следом за этим аналогичный вызов освобождает память, занимаемую стеком и данными. Иногда всю память можно освободить одним вызовом free_mem, но выгода того не стоит. Если родитель процесса ожидает, то, чтобы освободить ячейку, вызывается cleanup. Если нет, процесс становится зомби, это индицируется флагом HANGING в поле mp_flags. Полностью уничтожив

процесс или превратив его в зомби, менеджер памяти ищет в таблице процессов его потомков. Если таковые обнаруживаются, они делаются потомками процесса init. Когда init в ожидании и один из его потомков становится зомби, для этого потомка вызывается cleanup. Таким образом обрабатываются ситуации, подобные показанной на рис. 4.36, *а*. На этом рисунке мы видим процесс 12, который собирается завершиться, и его родителя, процесс 7, находящегося в ожидании. Чтобы избавиться от процесса 12, для него будет вызвана cleanup, в результате процессы 52 и 53 станут потомками init. Эта ситуация продемонстрирована на рис. 4.36, *б*. В результате оказывается, что процесс 53, который уже завершился, является потомком процесса, выполняющего wait. Следовательно, записи о нем будут корректно очищены.

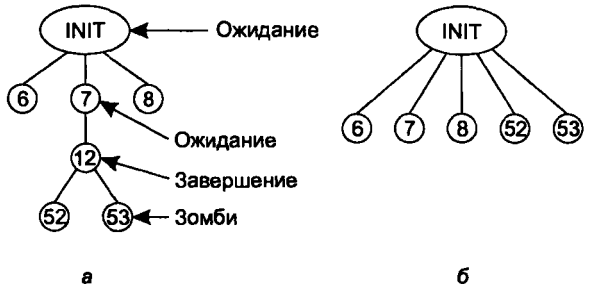


Рис. 4.36. *а* — процесс 12 собирается завершиться; *б* — ситуация после его завершения

Когда родитель делает вызов wait или waitpid, управление передается следующей функции, do_waitpid. Параметры этих двух системных вызовов различны, ожидаемые действия также различны, но благодаря специальной настройке значений двух внутренних переменных (options и pidarg) функция do_waitpid может выполнять оба вызова. После того как присвоены значения внутренним переменным, стартует цикл, в котором просматривается вся таблица процессов, с целью выяснить, есть ли вообще у процесса потомки. Если есть, проверяется, есть ли среди них зомби, которых теперь можно добить. Когда обнаруживается зомбированный процесс, он уничтожается и do_waitpid возвращает управление (второй оператор if внутри цикла). Так как ответное сообщение отправляется из функции cleanup, а не из цикла в main, устанавливается флаг dont_reply. Если обнаруживается трассируемый потомок, do_waitpid передает управление назад, отправив предварительно ответное сообщение, говорящее, что процесс остановлен. Флаг dont_reply при этом также устанавливается, ради предотвращения отправки второго ответного сообщения из main.

Если оказалось, что у вызвавшего wait процесса нет потомков, возвращается код ошибки (последний оператор return в функции). Если потомки есть, но среди них нет ни зомби, ни отслеживаемых процессов, проверяется, хотел ли сделавший вызов процесс дожидаться завершения работы потомков. Если да (это обычный случай), устанавливается бит, означающий, что процесс находится в ожидании, и родитель приостанавливается до тех пор, пока один из его потомков не завершится.

Когда процесс заканчивает выполнение, а его родитель ожидает его завершения (в каком бы порядке это ни произошло), для исполнения последних церемоний вызывается функция cleanup. У нее не слишком много работы. Родительский процесс, прохлаждающийся в ожидании в результате вызова wait или waitpid, пробуждается, ему передается PID завершившегося потомка, а также код выхода и состояние сигналов. Память потомка уже освобождена файловой системой, а ядро исключило его из планирования, поэтому все, что осталось сделать для завершения ритуала, — это очистить занимаемую процессом ячейку в таблице процессов.

4.8.4. Реализация системного вызова exec

Код системного вызова exec соответствует алгоритму из раздела 4.7.5. Этот код находится в функции do_exec. Сделав несколько простых проверок правильности данных, менеджер памяти извлекает из пользовательского адресного пространства имя файла программы, которая будет запущена как новый процесс. Перед тем как работать с файлом, файловой системе отправляется специальное сообщение, меняющее текущий каталог, чтобы полученный путь интерпретировался относительно рабочего каталога пользователя, а не каталога менеджера памяти.

Если файл существует и разрешено его исполнение, менеджер памяти считывает его заголовок и извлекает из него размеры сегментов. Затем из пользовательского пространства извлекается стек, определяется, будет ли новый процесс разделять код с уже работающими, выделяется память для нового образа, подправляются значения указателей (сравните рис. 4.35, *б* и *в*) и считываются сегменты кода (по необходимости) и данных. В завершение вызов выполняет специальные действия, если установлены биты setuid или setgid, обновляет запись в таблице процессов и сообщает ядру о том, что вызов завершен и процесс можно планировать.

Хотя все шаги управляются функцией do_exec, многие действия вынесены во вспомогательные процедуры в файле exec.c. Например, функция read_header не только считывает заголовок и возвращает размеры сегментов, но и проверяет, является ли файл нормальным исполняемым файлом MINIX для данного типа процессора. Тип процессора учитывается в директивах условной компиляции при компиляции менеджера памяти. Кроме того, read_header проверяет, умещаются ли сегменты в оперативной памяти.

Процедура new_mem смотрит, достаточно ли доступной памяти для загрузки нового образа. Для этого она ищет свободный блок, достаточно большой для размещения данных и стека в том случае, если код разделяется, а если код не разделяется, ищется блок для размещения текста, данных и стека в совокупности. Этот алгоритм можно улучшить, если отдельно искать свободное место для кода и для стека с данными, так как эти сегменты не обязаны быть расположены вместе. Данное требование было обязательным для ранних версий MINIX. Если нужная память найдена, ранее занятая память освобождается и выделяется новый блок. Если памяти недостаточно, вызов exec завершается неудачей. Выделив

память, new_mem обновляет карту памяти (mp_seg) и передает ее ядру при помощи вызова библиотечной процедуры sys_newmap.

Оставшаяся часть кода new_mem связана с обнулением области неинициализированных глобальных переменных (сегмент bss), области зазора и сегмента стека. Многие компиляторы сами генерируют обнуляющий код, но благодаря тому, что очистка выполняется менеджером памяти, MINIX может работать с компиляторами, которые так не поступают. Обнуляется и участок между сегментами данных и стека, чтобы при расширении области данных посредством brk выделяемая память уже содержала нули. Это не только дополнительное удобство для программиста, который может рассчитывать на то, что новые переменные инициализируются нулем, это еще и средство обеспечения защиты информации в многопользовательских системах, так как процесс, ранее занимавший память, мог содержать данные, которые нельзя показывать другим процессам.

Следующая процедура называется patch_ptr, ее задача в исправлении значений указателей из формы, показанной на рис. 4.35, *б*, в форму на рис. 4.35, *в*. Алгоритм ее работы прост: найти в стеке все указатели и добавить к ним значение базового смещения.

Процедура load_seg вызывается при исполнении exec один или два раза, для загрузки сегмента данных и иногда для загрузки сегмента текста (кода). Вместо того чтобы считывать сегмент блок за блоком и копировать блоки в адресное пространство пользователя, здесь используется трюк, при помощи которого файловая система может целиком загрузить сегмент напрямую. В результате вызов интерпретируется файловой системой несколько необычным образом, как будто чтение всего сегмента выполняется самим пользовательским процессом. Знают о том, что здесь что-то неладно, только несколько первых строк кода процедуры чтения из файловой системы. Благодаря такому маневру загрузка заметно ускоряется.

Последняя подпрограмма в файле exec.c называется find_share. Она по значению *i*-узла, номеру устройства и времени модификации ищет в таблице процессов процесс, с которым можно разделить код. Это простой последовательный поиск подходящего поля в таблице mproc. Конечно, при просмотре необходимо игнорировать сам процесс, для которого поиск заказан.

4.8.5. Реализация вызова brk

Как мы видели, модель памяти в MINIX довольно проста: каждому процессу при его создании выделяется один непрерывный участок памяти для данных и стека. Процесс никогда не перемещается в памяти, никогда из нее не выгружается, не растет и не уменьшается. Могут произойти только два важных события: область данных может израсходовать резерв и достигнуть области стека, и, наоборот, стек может разрастись на всю область зазора и наложиться на область данных. С учетом этих обстоятельств, реализация системного вызова brk (файл brk.c) относительно проста. При его выполнении сначала просто проверяется, что указанные размеры допустимы, а затем изменения вносятся в таблицы.

Выполняет вызов подпрограмма do_brk, но большая часть работы делается в процедуре adjust. Последняя проверяет, не пересеклись ли сегмент данных и стек. Если да, вызов brk завершается с ошибкой, но процесс не уничтожается немедленно. При выполнении сравнения к верхней границе области данных добавляется значение фактора безопасности, SAFETY_BYTES, поэтому остается надежда на то, что в стеке осталось еще немного места и процесс может поработать еще немного. Управление возвращается процессу, чтобы он хотя бы напечатал соответствующее сообщение и правильно завершился.

Обратите внимание: значение SAFETY_BYTES задано в середине процедуры при помощи директивы #define. Это довольно необычно, традиционно подобные объявления размещаются в начале файла или в отдельных заголовочных файлах. Комментарий рядом поясняет, что программист нашел сложным выбор значения фактора безопасности. Без сомнения, описание было расположено таким необычным образом для привлечения внимания и, возможно, чтобы стимулировать дальнейшие эксперименты.

Базовый адрес сегмента данных не меняется, поэтому adjust требуется обновлять только длину сегмента. Стек растет вниз с фиксированного конечного адреса, поэтому если adjust обнаруживает, что указатель стека (он передается в виде параметра) вышел за пределы области стека (то есть достиг более низких адресов), обновляются как адрес начала сегмента стека, так и его длина.

Последняя подпрограмма, size_ok, проверяет, помещаются ли сегменты заданных размеров в адресном пространстве, как в кликах, так и в байтах. Чтобы не делать отдельную версию функции для 16-битного варианта MINIX, используются директивы условной компиляции. Эта функция вызывается только в двух местах, и замена этих вызовов на первую строку кода из 32-битного варианта даст более компактную программу, поскольку некоторые передаваемые в функцию аргументы не нужны для 32-битной версии.

4.8.6. Реализация сигналов

С сигналами связаны восемь системных вызовов, перечисленных в табл. 4.5. Как сами сигналы, так и эти системные вызовы обрабатываются кодом из файла signal.c. Там же находится код еще одного системного вызова, reboot, из-за того что этот вызов использует сигналы, чтобы завершить все процессы.

Таблица 4.5. Системные вызовы, относящиеся к сигналам

Системный вызов	Назначение
sigaction	Изменяется реакция на будущий сигнал
sigprocmask	Модифицируется набор блокируемых сигналов
kill	Отправка сигнала другому процессу
alarm	Отправка сигнала ALRM самому себе после задержки
pause	Приостановить работу до следующего сигнала
sigsuspend	Набор блокируемых сигналов изменяется, затем вызывается pause
sigpending	Определить набор текущих (то есть заблокированных) сигналов
sigreturn	Восстановление после завершения обработчика сигнала

Системный вызов sigaction поддерживает две функции, sigaction и signal, позволяющие процессу изменять свою реакцию на сигнал. Функция sigaction регламентирована стандартом POSIX и является предпочтительным способом для всех целей, но библиотечная функция signal удовлетворяет ANSI C, и, если программа должна быть переносима на не-POSIX системы, она должна использовать вторую функцию. Код do_sigaction начинается с проверок правильности номера сигнала и того, что не делается попытки изменить реакцию на сигнал KILL. (Сигнал SIGKILL нельзя ни игнорировать, ни блокировать, ни обрабатывать. Это исключительно то средство, при помощи которого пользователь может контролировать собственные процессы, а системный оператор может контролировать пользователей.) При вызове sigaction передаются указатели на структуру типа sigaction, это sig_osa, куда помещаются старые значения атрибутов, и sig_nsa, где хранится новый набор атрибутов.

На первом шаге, чтобы скопировать текущие значения атрибутов по указателю sig_osa, вызывается системная задача. Далее, при вызове sigaction указатель sig_nsa может иметь значение NULL. Это означает, что требуется считать текущие значения атрибутов, не меняя их. В таком случае sigaction немедленно возвращает управление. Если указатель sig_nsa не равен NULL, в пространство менеджера памяти копируется новая структура, описывающая действие сигнала. Затем модифицируются битовые карты mp_catch, mp_ignore и mp_sigpending, чтобы указанный сигнал либо игнорировался, либо обрабатывался по умолчанию, либо вызывал обработчик. При этом используются библиотечные функции sigaddset и sigdelset, хотя те же действия можно было реализовать и при помощи макроса, как и другие простые манипуляции с битами. Тем не менее эти функции требуются по стандарту POSIX с целью упростить перенос программ на другие системы, в том числе и на те, в которых общее число сигналов превышает число битов в целом значении. Использование библиотечных функций упрощает и перенос самой MINIX на различные архитектуры.

В завершение заполняются другие относящиеся к сигналам поля той части таблицы процессов, которая принадлежит менеджеру памяти. Для каждого потенциального сигнала здесь есть битовая карта, sa_mask, определяющая, какие сигналы будут блокироваться при выполнении его обработчика. Кроме того, для каждого сигнала хранится указатель, sa_handler. Он может содержать либо указатель на функцию-обработчик, либо специальное значение, означающее, что сигнал должен быть блокирован или обработан по умолчанию. Адрес библиотечной функции, по завершении обработчика запускающей системный вызов sigreturn, хранится в поле mp_sigretrun. Этот адрес передается в одном из полей сообщения, которое получает менеджер памяти.

POSIX позволяет процессу изменять способ обработки получаемых им сигналов даже внутри обработчика. Благодаря этому можно, например, изменить реакцию на сигнал на время его обработки, а затем восстановить нормальный отклик. Следующая группа системных вызовов и служит средствами подобной манипуляции сигналами. Вызов sigpending выполняется функцией do_sigpending, возвращающей битовую карту mp_sigpending, по которой процесс может определить, имеются ли задержанные сигналы. Вызов sigprocmask, обслуживаемый

функцией do_sigprocmask, возвращает набор блокируемых сигналов. Кроме того, с его помощью можно изменить блокировку либо одного сигнала из набора, либо сразу установить новый набор. В тот момент, когда с сигнала снимается блокировка, стоит проверить, есть ли задержанные сигналы, для этого в конце вариантов SIG_SETMASK и SIG_UNBLOCK вызывается check_pending. Функция do_sigsuspend выполняет системный вызов sigsuspend. Этот вызов приостанавливает выполнение процесса до тех пор, пока не будет получен сигнал. Как и другие обсуждаемые здесь функции, do_sigsuspend манипулирует битовыми картами. Помимо того, она устанавливает в поле mp_flags бит SIGSUSPENDED, что и предотвращает выполнение процесса. Опять же, в завершение здесь стоит вызвать check_pending. Наконец, функция do_sigretrun обслуживает системный вызов sigreturn, используемый для возврата из пользовательского обработчика сигнала. Эта функция восстанавливает контекст сигналов, предшествующий входу в обработчик, а также вызывает check_pending.

Некоторые сигналы, такие как SIGINT, исходят из самого ядра. Такие сигналы обслуживаются подобно тому, как обслуживаются сигналы, генерируемые пользовательскими процессами при вызове kill. Поэтому две процедуры, do_kill и do_ksig, основаны на сходных идеях. Обе они заставляют менеджер памяти отправить сигнал. Так как при помощи одного вызова kill можно отправить сигнал группе процессов, do_kill просто вызывает функцию check_sig, которая ищет во всей таблице процессов подходящих получателей. Функция do_ksing вызывается тогда, когда получено сообщение от ядра. Это сообщение содержит битовую карту, при помощи которой ядро может генерировать несколько сигналов одним сообщением. Как и в случае kill, каждый из сигналов доставляется как одному процессу, та и их группе. Цикл внутри do_ksig бит за битом обрабатывает переданную битовую карту. Отдельным сигналам требуется особое внимание: в некоторых случаях меняется идентификатор процесса с той целью, чтобы сигнал был доставлен группе процессов (варианты SIGQUIT и SIGKILL), а SIGALRM, если он не был запрошен, игнорируется. За исключением этих специальных случаев, каждый установленный бит приводит к вызову check_sig, как и в do_kill.

Системный вызов alarm выполняется функцией do_alarm. Она вызывает следующую функцию, set_alarm, которая отправляет задаче часов сообщение, запуская таймер. Код set_alarm вынесен в отдельную функцию потому, что он также используется для отключения таймера, когда процесс завершается, а таймер еще работает. Когда заданный промежуток времени истекает, ядро сообщает этот факт менеджеру памяти, отправляя ему сообщение типа KSIG, которое, как говорилось выше, влечет за собой запуск do_ksig. По умолчанию сигнал SIGALRM приводит к завершению процесса. Если он должен обрабатываться, вызовом sigaction задается соответствующий обработчик. Вся последовательность событий обработки сигнала SIGALRM показана на рис. 4.37. Здесь представлены три последовательности сообщений. При помощи сообщений 1, 2 и 3 пользователь делает системный вызов alarm (сообщение менеджеру памяти), менеджер памяти отправляет запрос задаче часов, а задача часов передает подтверждение. В сообщениях 4, 5 и 6 задача часов докладывает менеджеру памяти о срабатывании таймера, менеджер памяти подает прошение системной задаче приготовить стек процесса к выпол-

нению обработчика (как на рис. 4.37), и задача системы отправляет ответ. Сообщение 7 — это вызов sigretrun, который происходит после завершения обработчика. В ответ менеджер памяти отправляет системной задаче сообщение 8, чтобы она закончила обработку, а задача системы отвечает 9. Сообщение 6 само по себе не приводит к запуску обработчика, но он будет стартован позже, так как задаче часов, благодаря принятой в MINIX системе планирования с приоритетами, будет позволено завершить свою работу. Обработчик является частью пользовательского процесса, поэтому не может быть запущен на выполнение до того, как системная задача не закончит свои дела.

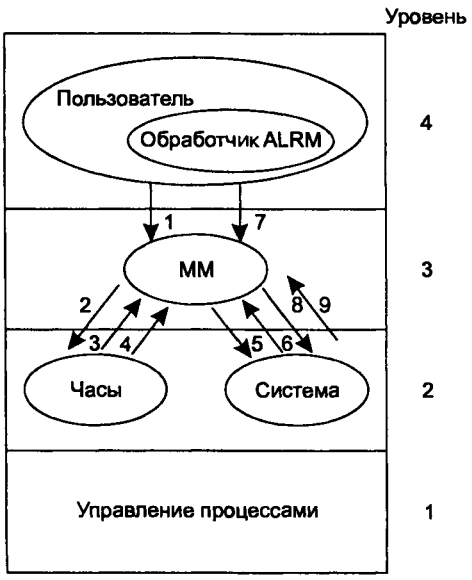


Рис. 4.37. Сообщения при работе таймера. Самые главные из них: *1* — пользователь делает вызов alarm; *4* — после истечения заданного интервала времени прибывает сигнал; *7* — обработчик завершается вызовом sigreturn. Подробности смотрите в тексте

Функция do_pause заботится о системном вызове pause. Все, что ей нужно сделать, — это установить бит и не отвечать, удерживая сделавший вызов процесс заблокированным. Не нужно даже информировать ядро, так как оно уже знает, что процесс заблокирован.

Последний системный вызов, код которого расположен в signal.c, — это вызов reboot. Он делается только специализированными программами, под руководством суперпользователя, ввиду важности возложенной на этот вызов роли. Он проверяет, что все процессы корректно завершены и файловая система синхронизирована, прежде чем передать задаче системы указание завершить работу системы. Для завершения процессов при помощи check_sig всем процессам, за исключением init, отправляется сигнал SIGKILL. Именно поэтому код reboot помещен в данном файле.

Выше мы упомянули несколько вспомогательных функций. Теперь рассмотрим их подробно. Важнейшей из них является sig_proc, которая и отвечает за отправку сигнала. Сначала она делает несколько проверок. Попытка послать сигнал завершившемуся или «потустороннему» процессу является признаком серьезной ошибки и приводит к панике системы. Если процесс трассируется, то при получении сигнала он останавливается. Если же сигнал должен игнорироваться, работа sig_proc завершается. Это действие принято по умолчанию для ряда сигналов, например для тех, которые задает стандарт POSIX, но не поддерживает MINIX. Если сигнал блокируется, нужно только установить соответствующий бит в битовой карте mp_sigpending процесса. Ключевое значение имеет проверка, разделяющая процессы на тех, кому разрешено обрабатывать сигнал, от тех, кому нет (6-й оператор if в этой функции). С этого момента все прочие варианты исключаются, и процессы, не удостоившиеся права обработать сигнал, будут завершены.

Следующая часть функции отвечает за сигналы, которые процессу позволено обработать. Сначала формируется сообщение для ядра, некоторые части сообщения заполняются копиями информации из принадлежащей менеджеру памяти части таблицы процессов. Если ранее процесс был приостановлен сигналом SIGSUSPEND, в сообщение будет включена маска сигналов, сохраненная в момент остановки, если нет, туда будет вставлена текущая маска. Дополнительно в сообщение включаются некоторые адреса из адресного пространства процесса-получателя сигнала: адрес обработчика, адрес библиотечной подпрограммы sigretrun, подлежащей вызову после завершения обработчика, и текущее значение указателя стека.

Затем выделяется место в стеке процесса. Структура, которая помещается в стек, показана на рис. 4.38. В нее входит и структура sigcontext одного из полей таблицы процессов. Она сохраняется в стеке потому, что ее значение в таблице процессов меняется при подготовке к запуску обработчика. Часть, названная sigframe, содержит адрес возврата и данные, необходимые вызову sigreturn для восстановления состояния процесса. Адрес возврата и указатель кадра стека в действительности не используются нигде в MINIX. Они нужны для того, чтобы обмануть отладчик при трассировке обработчика событий.

Структура, помещаемая в стек, довольно объемна. Для нее выделяется место (декремент переменной new_sp), а затем, чтобы проверить, достаточно ли байтов в стеке, вызывается adjust. Если места не хватает, происходит переход по метке doterminate (здесь вы можете увидеть редко используемый и всеми гонимый оператор языка C goto), и процесс завершается.

Существует потенциальная проблема при вызове adjust. При обсуждении реализации системного вызова brk говорилось, что adjust начинает сообщать об ошибке тогда, когда стек приблизился к области данных на расстояние SAFETY_BYTES. Дополнительный зазор вводится потому, что программно стек может проверяться лишь время от времени. В данном случае, вероятно, такой запас прочности излишен, так как достоверно известно, какая часть стека требуется для сигнала, а дополнительное место необходимо только обработчику, который предположительно является относительно простой функцией. Поэтому возможно, что неко-

торые процессы вследствие чересчур строгой проверки в adjust будут заживо похоронены, но это все же лучше, чем наблюдать таинственные сбои в работе в другое время. Таким образом, данные проверки еще можно улучшить.

Если в стеке достаточно места, проверяются значения еще двух флагов. Флаг SA_NODEFER означает, что дальнейшие сигналы должны быть блокированы на время выполнения обработчика. Флаг SA_RESETHAND — признак того, что при получении сигнала обработчик должен быть снят. (В результате получается честная имитация устаревшего вызова signal. Хотя такое поведение обычно рассматривалось как недостаток, при поддержке устаревших функций необходимо поддерживать и их недостатки.) Затем при помощи библиотечной процедуры sys_sendsig отправляется уведомление ядру. В завершение сбрасывается бит-индикатор задержанного сигнала и вызывается функция unpause, которая прерывает любой системный вызов, остановивший процесс. В следующий раз, когда процесс получит управление, начнет работу обработчик.

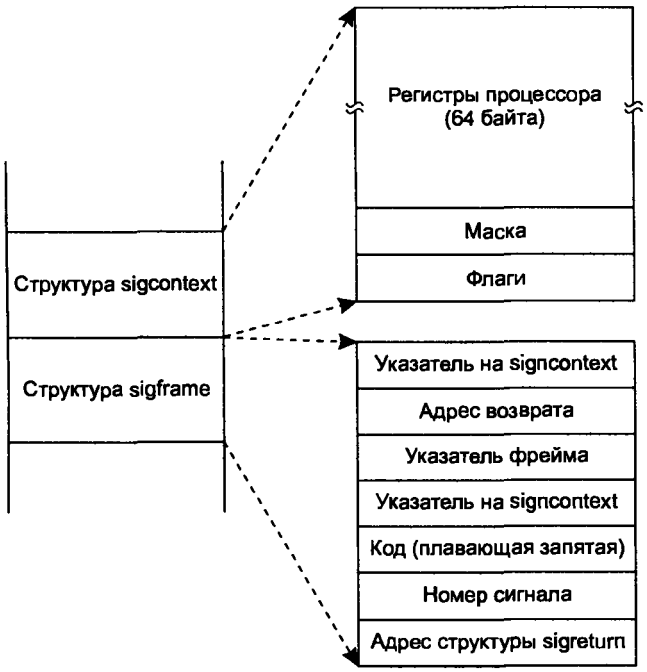


Рис. 4.38. Чтобы подготовиться к запуску обработчика, в стек помещаются структуры sigcontext и sigframe. Регистры процессора копируются из кадра стека на момент переключения контекстов

Теперь давайте взглянем на код завершения процесса, под меткой doterminate. Оператор goto и метка — самый простой способ обработать возможный отказ при вызове adjust. Итак, сюда передается управление, если сигнал по тем или другим причинам не может или не должен быть обработан. В таком случае вызо-

вом mm_exit процесс всегда завершается (как если бы он хотел этого сам), а иногда, в зависимости от сигнала, может выводиться дамп.

В функции check_sig менеджер памяти проверяет, может ли быть послан сигнал. Так, вызов

```
kill (0,sig);
```

означает, что указанный сигнал должен быть отправлен всем процессам в текущей группе (то есть всем процессам, запущенным с того же терминала). Сигналы, исходящие от ядра, и сигнал REBOOT также могут затрагивать несколько процессов. По этой причине check_sig в цикле перебирает все процессы из таблицы процессов, выбирая из них потенциальных получателей. В цикле содержится много тестов, и только если все они пройдены, при помощи sig_proc отправляется сигнал.

Еще одна функция, которая несколько раз вызывается в недавно рассмотренном нами коде, называется sig_pending. Она перебирает все биты в битовой карте mp_sigpending процесса, сверяясь с битовыми картами do_sigmask, do_sigretrun и do_sigsuspend, и смотрит, не была ли снята блокировка с одного из задержанных сигналов. Обнаружив первый такой сигнал, она отправляет его. Так как все обработчики событий со временем вызывают do_sigretrun, в результате доставляются все разблокированные задержанные сигналы.

Процедура unpause необходима для отправки сигналов процессам, приостановленным на одном из вызовов pause, wait, read, write или sigsuspend. Выяснить, обусловлена ли остановка вызовами pause, wait или sigsuspend, можно, сверившись с таблицей процессов менеджера памяти. Если та не дает ответа, запрос передается файловой системе, которая при помощи собственной функции do_unpause проверяет, был ли процесс приостановлен на read или write. В любом случае, результат одинаков: незаконченный вызов завершается с ошибкой, а процесс вновь запускается и может обработать сигнал.

Последняя процедура в этом файле называется dump_core, она записывает на диск дампы памяти. Дамп состоит из заголовка, содержащего информацию о размере занимаемых процессом сегментов, всей информации состояния процесса (это копия записи процесса из таблицы процессов ядра) и образов всех сегментов. Отладчик может прочитать эту информацию, чтобы помочь программисту определить, что пошло не так в работе программы. Код записи в файл прямолинеен. Здесь снова возникает потенциальная проблема, упомянутая в предыдущем разделе, но теперь в несколько иной форме. Чтобы гарантировать, что записываемый в дамп сегмент стека содержит самую последнюю информацию, вызывается adjust. Из-за проверки границы безопасности этот вызов может «провалиться». Правда, дамп записывается в любом случае, независимо от успеха вызова, но информация о стеке может оказаться неправильной.

4.8.7. Прочие системные вызовы

Файл getset.c содержит только одну процедуру, do_getset, выполняющую оставшиеся семь системных вызовов. Эти вызовы перечислены в табл. 4.6. Все они на-

столько просты, что выделять для них отдельные процедуры не имеет смысла. Вызовы getuid и getgid возвращают как реальные, так и эффективные значения UID (идентификатор пользователя) и GID (идентификатор группы).

Установить GID или UID несколько сложнее, чем просто прочитать. Сначала нужно проверить, уполномочен ли процесс менять эти значения. Если проверка пройдена, об изменении GID или UID необходимо информировать файловую систему, так как эти значения влияют на защиту файлов. Вызов setsid создает новый сеанс, этого нельзя делать процессу, который уже является лидером группы. После проверки работу вызова завершает файловая система, делая процесс лидером группы без управляющего терминала.

Таблица 4.6. Системные вызовы, поддерживаемые mm/getset.c

Системный вызов	Описание
getuid	Возвращает реальный и эффективный идентификаторы пользователя
getgid	Возвращает реальный и эффективный идентификаторы группы
getpid	Возвращает идентификаторы процесса и его родителя
setuid	Устанавливает реальный и эффективный идентификаторы пользователя
setgid	Устанавливает реальный и эффективный идентификаторы группы
setsid	Создает новый сеанс и возвращает идентификатор процесса
getpgrp	Возвращает идентификатор группы процессов

Минимальную поддержку отладки посредством системного вызова ptrace обеспечивает код из файла trace.c. Системный вызов ptrace умеет выполнять одиннадцать различных команд, перечисленных в табл. 4.7. Четыре из них, enable, exit, resume и step, обрабатываются менеджером памяти. Здесь же выполняются запросы на начало и завершение трассировки. Все прочие запросы перенаправляются задаче системы, имеющей доступ к таблице процессов ядра. За это отвечает функция sys_trace. В конце файла trace.c есть две вспомогательные функции, нужные для трассировки. Это функция stop_proc, которая останавливает трассируемый процесс, получивший сигнал, и функция findproc, помогающая do_trace найти отлаживаемый процесс в таблице процессов.

4.8.8. Утилиты менеджера памяти

Оставшиеся файлы содержат вспомогательные процедуры и таблицы. Файл alloc.c помогает системе отслеживать, какие участки памяти заняты, а какие свободны. В нем определено четыре точки входа:

- alloc_mem — запрос блока памяти заданного размера;
- free_mem — возврат блока памяти;
- max_hole — вычисляет размер самого большого свободного блока («дыры»);
- mem_init — инициализирует список свободных блоков при запуске менеджера памяти.

Как уже было сказано ранее, функция alloc_mem просто ищет в списке первый блок подходящего размера. Если она обнаруживает достаточно большой блок, она отрезает от него нужную часть, а лишнее оставляет в списке. Если требуется весь блок, вызывается функция del_slot, которая полностью удаляет блок из списка.

Таблица 4.7. Команды отладки, поддерживаемые mm/trace.c

Команда	Описание
T_STOP	Останавливает процесс
T_OK	Включает трассировку процесса родителем
T_GETINS	Возвращает значение из пространства команд
T_GETDATA	Возвращает значение из области данных
T_GETUSER	Возвращает значение из таблицы процессов пользователя
T_SETINS	Записывает значение в пространство команд
T_SETDATA	Записывает значение в область данных
T_SETUSER	Записывает значение в таблицу процессов пользователя
T_RESUME	Возобновляет выполнение
T_EXIT	Выход
T_STEP	Устанавливает бит трассировки

Функция free_mem должна проверять, можно ли объединить возвращенный блок с соседними. Если это возможно, вызывается merge, которая объединяет указанные блоки и обновляет список.

Функция max_hole сканирует список свободных блоков и возвращает самое большое из найденных ею значений. Начальный список, охватывающий всю доступную память, строится функцией mem_init.

Процедуры из следующего файла, utility.c, используются в различных местах по всему коду менеджера памяти. Процедура allowed проверяет, предоставлен ли доступ к файлу. Эта функция необходима, например, do_exec, чтобы проверить, является ли файл исполняемым.

Процедура no_sys не должна вызываться никогда. Она здесь только на тот случай, если менеджер памяти получит системный вызов с недопустимым номером или вызов, не обрабатываемый им.

Когда менеджер памяти обнаруживает ошибку, после которой невозможно восстановление, он вызывает функцию panic. Она сообщает об ошибке задачи системе, которая «со скрежетом» останавливает «несущуюся в пропасть» MINIX. Просто так эта функция не вызывается — не делайте этого.

Последняя функция в utility.c, tell_fs, подготавливает сообщение и отправляет его файловой системе, когда последнюю необходимо информировать об обработанных менеджером памяти событиях.

Две подпрограммы из файла putk.c также являются утилитами, хотя и несколько иного типа, чем предыдущие. Время от времени в коде менеджера памяти встречаются вызовы printf, большей частью для отладки. Функция panic также вызывает printf. Ранее уже упоминалось, что в данном случае имя printf в действительности является макросом, разворачиваемым в printk, то есть эти вызовы не

используют стандартные процедуры ввода/вывода, работающие через файловую систему. Функции putk и printk напрямую общаются с задачей терминала, что запрещено обычным пользователям. Процедуры с точно такими же названиями мы видели в коде ядра.

Резюме

В этой главе были проанализированы как общие принципы управления памятью, так и их реализация в MINIX. Мы увидели, что в простейших системах вообще нет свопинга или страничной организации памяти. Программа, загруженная в память, остается там до своего завершения. Некоторые операционные системы позволяют находиться в памяти одновременно только одному процессу, в то время как другие поддерживают многозадачность.

Следующим шагом вперед является свопинг (то есть загрузка и выгрузка целых процессов). Когда используется подкачка такого вида, система может обрабатывать большее количество процессов, чем то, для которого достаточно пространства в памяти. Процессы, для которых в ней нет места, целиком выгружаются на диск. Свободные области в памяти и на диске можно отслеживать с помощью битового массива или списка свободных участков.

Современные компьютеры часто поддерживают некоторую форму виртуальной памяти. В простейшем виде адресное пространство каждого процесса делится на части постоянного размера, называемые страницами, которые могут размещаться в любом доступном страничном блоке в памяти. Существует множество алгоритмов замещения страниц, два наилучших — это алгоритмы «старения» и алгоритм «второй попытки». Для хорошей работы страничных систем недостаточно выбора алгоритма; кроме этого, необходимо обратить внимание на такие вопросы, как определение рабочего набора, стратегию предоставления памяти и размер страниц.

Сегментация помогает в управлении структурами данных, изменяющими свой размер во время выполнения, и упрощает процессы компоновки и совместного доступа. Она также облегчает предоставление различных видов защиты разным сегментам. Иногда сегментация и разбивка на страницы комбинируются, чтобы обеспечить двумерную виртуальную память. Системы MULTICS и Intel Pentium поддерживают сегментацию и страничную организацию памяти.

Управление памятью в MINIX реализовано просто. Процессу выделяется память, когда он делает системный вызов fork или exec. После того как участок памяти выделен, он никогда не меняет своих размеров, пока работает процесс. На машинах с Intel-процессорами MINIX реализует для процессов две модели памяти. У маленьких программ инструкции и данные могут размещаться в одном сегменте. В более сложных программах адресные пространства данных и кода могут быть выделены. Такие процессы могут разделять общий код, поэтому при вызове fork им выделяется память лишь под данные и стек. Такое не исключено и при вызове exec, если окажется, что в памяти уже находится процесс, использующий тот же код, что нужен новой программе.

По большей части работа менеджера памяти связана не с отслеживанием памяти, что делается при помощи списка свободных блоков и алгоритма выбора первого подходящего блока, а с обслуживанием системных вызовов, относящихся к управлению памятью. Некоторые из системных вызовов обеспечивают работу сигналов в стиле POSIX, а так как по умолчанию большинство таких сигналов завершают процесс, имеет смысл обрабатывать их в менеджере памяти, инициирующем завершение всех процессов. Отдельные системные вызовы напрямую к работе с памятью не относятся, но также обрабатываются менеджером памяти, в основном из-за того, что он меньше файловой системы и поместить их здесь удобнее.

Вопросы

1. Компьютерная система имеет достаточно места для того, чтобы содержать в оперативной памяти четыре программы. Эти программы простаивают в ожидании ввода/вывода половину времени. Какая часть времени работы центрального процессора пропадает?
2. Рассмотрим систему обычной подкачки, в памяти которой содержатся свободные участки следующих размеров и в следующем порядке: 10 Кбайт, 4 Кбайт, 20 Кбайт, 18 Кбайт, 7 Кбайт, 9 Кбайт, 12 Кбайт и 15 Кбайт. Какой из них будет выбран для успешного удовлетворения запроса сегмента размером

   1) 12 Кбайт,

   2) 10 Кбайт,

   3) 9 Кбайт

   по алгоритму «первый подходящий»? Ответьте на тот же самый вопрос для алгоритмов «самый подходящий», «самый неподходящий» и «следующий подходящий».
3. В чем разница между физическим адресом и виртуальным?
4. Опираясь на таблицу страниц на рис. 4.8, сосчитайте физический адрес, соответствующий каждому из следующих виртуальных адресов:

   1) 20,

   2) 4100,

   3) 8300.
5. Процессор Intel 8086 не поддерживает виртуальную память. Тем не менее некоторые компании ранее продавали системы, содержащие неизмененный процессор 8086 и выполняющие страничную подкачку. Предложите гипотезу того, как они это делали. Подсказка: подумайте о логическом расположении диспетчера памяти (MMU).
6. Считая, что команда выполняется за 1 мкс, а страничное прерывание требует дополнительно n мкс, напишите выражение для фактического време-

ни выполнения команды с учетом того, что прерывания происходят через каждые k инструкций.

7. Компьютер имеет 32-разрядное адресное пространство и страницы размером 8 Кбайт. Таблица страниц целиком поддерживается аппаратно, на запись в ней отводится одно 32-разрядное слово. При запуске процесса таблица страниц копируется из памяти в аппаратуру, одно слово требует 100 нс. Если каждый процесс работает в течение 100 мс (включая время загрузки таблицы страниц), какая доля времени процессора жертвуется на загрузку таблицы страниц?
8. Компьютер с 32-разрядным адресом использует двухуровневую таблицу страниц. Виртуальные адреса расщепляются на 9-разрядное поле верхнего уровня таблицы, 11-разрядное поле второго уровня таблицы страниц и смещение. Чему равен размер страниц и сколько их в адресном пространстве?
9. Ниже показан алгоритм фрагмента программы для компьютера с размером страницы 512 байт. Программа расположена по адресу 1020, указатель стека равен 8192 (стек увеличивается по направлению к нулю). Напишите последовательность страничных обращений, создаваемую этой программой. Каждая инструкция занимает 4 байта (1 слово), включая непосредственные константы. В последовательности обращений учитываются обращения как к инструкциям, так и к данным.

   Загрузить слово 6144 в регистр 0.
   Поместить содержимое регистра 0 в стек.
   Вызвать процедуру по адресу 5120, помещая в стек адрес возврата.
   Вычесть константу 16 из указателя стека.
   Сравнить полученный результат с константой 4.
   При равенстве перейти на адрес 5152.

10. Предположим, что 32-разрядный виртуальный адрес разбивается на четыре поля. Первые три используются для трехуровневой системы таблиц страниц. Четвертое поле — это смещение. Зависит ли количество страниц от размера всех четырех полей? Если нет, то какие из полей имеют значение, а какие нет?
11. Компьютер, процессы которого имеют 1024 страницы в своем адресном пространстве, хранит таблицы страниц в памяти. На чтение слова из таблицы страниц требуется 5 нс. Чтобы уменьшить затраты, в компьютере присутствует буфер быстрого преобразования адреса (TLB), содержащий 32 пары (виртуальная страница, физический страничный блок), который может выполнить поиск за 1 нс. При какой частоте обращений к памяти, успешно реализуемых в TLB, средние затраты будут ниже 2 нс?
12. Буфер быстрого преобразования адреса на машинах VAX не содержит бита R. Почему?
13. Машина поддерживает 48-разрядные виртуальные адреса и 32-разрядные физические адреса. Размер страницы равен 8 Кбайт. Сколько требуется записей в таблице страниц?

14. Компьютер имеет четыре страничных блока. Время загрузки, время последнего доступа и биты R и M для каждой страницы показаны ниже (время считается в тиках системных часов).

Страница	Загружена	Последнее обращение	R	M
0	126	280	1	0
1	230	265	0	01
2	140	270	0	0
3	110	285	1	1

1) Какую страницу выгрузит алгоритм NRU?
2) Какую страницу выгрузит алгоритм FIFO?
3) Какую страницу выгрузит алгоритм LRU?
4) Какую страницу выгрузит алгоритм «вторая попытка»?

15. Если используется алгоритм замещения страниц FIFO в системе с четырьмя страничными блоками и восемью страницами, сколько страничных прерываний произойдет для последовательности обращений 0172327103 при условии, что четыре страничных блока изначально пусты? Теперь решите эту задачу для алгоритма LRU.

16. У маленького компьютера четыре страничных блока. Во время первого тика часов биты R равны 0111 (у страницы 0 бит R равен 0, у остальных — 1). Во время последующих тиков часов биты R принимают значения 1011, 1010, 1101, 0010, 1010, 1100 и 0001. Считая, что используется алгоритм старения с 8-разрядным счетчиком, напишите четыре значения, которые примет счетчик после последнего тика.

17. Сколько времени займет загрузка с диска программы размером 64 Кбайт, если его среднее время поиска равно 10 мс, время оборота — 10мс, каждая дорожка содержит 32 Кбайт:

1) для размера страницы 2 Кбайт?
2) для размера страницы 4 Кбайт?

Страницы раскиданы по диску случайно, и количество цилиндров так велико, что можно игнорировать вариант, при котором две страницы оказываются на одном и том же цилиндре.

18. Одна из первых машин с системой разделения времени PDP-1 имела память 4 К 18-разрядных слов. В каждый конкретный момент времени она содержала в памяти один процесс. Когда планировщик решал запустить другой процесс, процесс в памяти записывался на страничный барабан с 4 К 18-разрядных слов по окружности барабана. Барабан мог начать запись (или чтение) с любого слова, а не только со слова 0. Как вы полагаете, почему была выбрана эта конструкция?

19. Компьютер обеспечивает каждый процесс 65 536 байтами адресного пространства, разделенного на страницы по 4096 байт. Некая программа имеет размер текста 32 768 байт, размер данных 16 386 байт и размер стека 15 870 байт. Поместится ли эта программа в адресном пространстве? А если бы размер страницы был 512 байт, она поместилась бы? Помните, что страница не может вмещать части двух разных сегментов.
20. Было замечено, что количество инструкций, выполненных между страничными прерываниями, прямо пропорционально количеству страничных блоков, предоставленных программе. Если доступная память увеличивается вдвое, то средний интервал между страничными прерываниями также увеличивается вдвое. Предположим, что нормальная инструкция занимает 1 мкс, но если происходит страничное прерывание, она выполняется за 2001 мкс (то есть 2 мс идут на обработку прерывания). Если программа требует для работы 60 с и в процессе она вызывает 15 000 страничных прерываний, сколько времени она работала бы в условиях удвоенного количества доступной исходной памяти?
21. Разработчики из компании «Экономные операционные системы» размышляют о способе уменьшения количества резервного пространства для хранения данных, необходимого в их операционной системе. Ведущий специалист предложил вообще не беспокоиться о сохранении текста программы в области подкачки, а просто загружать его страницами прямиком из двоичного файла всякий раз, когда он требуется. При каком условии, если оно существует, эта идея работает для текста программы? А при каком условии, опять же, если оно существует, она работает для данных?
22. Объясните разницу между внутренней и внешней фрагментацией. Какая из них происходит в страничных системах? А какая имеет место в системах с чистой сегментацией?
23. Когда поддерживаются и сегментация, и страничная организация памяти, как в системе MULTICS, сначала должен быть найден дескриптор сегмента, затем идентификатор страницы. Может ли при таком двухуровневом поиске работать также буфер быстрого преобразования адреса (TLB)?
24. Почему при принятой в MINIX системе управления памятью необходимы программы типа chmem?
25. Измените код MINIX так, чтобы зомби уничтожались сразу же после появления, не дожидаясь, пока об этом позаботится родитель.
26. В текущей версии MINIX при выполнении вызова exec менеджер памяти ищет свободный блок памяти достаточного размера для загрузки образа. Если такой блок не обнаружен, происходит отказ от выполнения. Более эффективный алгоритм должен проверять, будет ли в памяти достаточно места уже после того, как текущая память, занимаемая процессом, освобождена. Реализуйте такой алгоритм.
27. Выполняя системный вызов exec, менеджер памяти использует трюк, при помощи которого файловая система считывает весь сегмент сразу. Приду-

майте, как применить этот же подход для вывода дампов памяти, и реализуйте его.

28. Измените код MINIX, добавив поддержку свопинга.
29. В разделе 4.8.4 было указано, что системный вызов **exec** работает не оптимально, выполняя поиск свободного блока памяти до того, как высвобождена текущая память процесса. Улучшите алгоритм.
30. В разделе 4.8.4 было указано, что память для кода и данных лучше выделять раздельно. Реализуйте эту идею.
31. Измените код **adjust** так, чтобы избежать ситуаций, когда получивший сигнал процесс карается завершением зря, из-за слишком строгой проверки доступного стека.

Глава 5
Файловые системы

Всем компьютерным приложениям нужно хранить и получать информацию. В собственном адресном пространстве работающий процесс может хранить ограниченное количество данных, и емкость такого хранилища ограничена размером виртуального адресного пространства. Для некоторых приложений этого размера вполне достаточно, но для других, например систем резервирования авиабилетов, систем банковского или корпоративного учета, одного только виртуального адресного пространства будет недостаточно.

Кроме того, после завершения работы процесса информация, хранящаяся в его адресном пространстве, теряется. Для большинства приложений (например, баз данных) эта информация должна храниться неделями, месяцами или даже вечно. Исчезновение данных после завершения работы процесса для таких программ неприемлемо. Информация должна продолжать существовать даже при аварийном завершении процесса в случае сбоя компьютера.

Третья проблема состоит в том, что часто возникает необходимость нескольким процессам одновременно получить доступ к одним и тем же данным (или части данных). Если интерактивный телефонный справочник будет храниться в адресном пространстве одного процесса, то доступ к нему будет только у этого процесса. Для решения этой проблемы необходимо отделить информацию от процесса.

Таким образом, к устройствам долговременного хранения информации предъявляются три следующих важных требования:

- ✦ устройства должны позволять хранить очень большие объемы данных;
- ✦ информация должна оставаться в сохранности после прекращения работы процесса, использующего ее;
- ✦ несколько процессов должны иметь возможность получения параллельного доступа к информации.

Обычное решение всех этих проблем состоит в размещении информации на дисках и других внешних хранителях в модулях, называемых *файлами*. Процессы могут по мере надобности читать их и создавать новые файлы. Информация, хранящаяся в файлах, должна обладать *устойчивостью* (в данном контексте иногда применяется термин *персистентность*), то есть на нее не должно оказывать влияния создание или прекращение работы какого-либо процесса. Файл должен исчезать только тогда, когда его владелец дает команду удаления файла.

Файлами управляет операционная система. Их структура, именование, использование, защита, реализация и доступ к ним являются важными элементами устройства операционной системы. Часть операционной системы, работающая с файлами, называется *файловой системой*. Ей и посвящена данная глава.

С точки зрения пользователя, наиболее важным аспектом файловой системы является ее представление со стороны, то есть как выглядят именование и защита файлов, операции с файлами и т. д. Такие детали внутреннего устройства, как использование списков или битовых карт для слежения за свободными и занятыми блоками диска, или число физических секторов в логическом блоке, представляют для пользователя меньший интерес, хотя и крайне важны для разработчиков файловой системы. По этой причине мы разбили главу на несколько разделов. Первые два раздела посвящены пользовательскому интерфейсу файлов и каталогов. В следующих разделах мы рассмотрим способы реализации файловой системы. И в завершение будут приведены несколько примеров существующих файловых систем.

5.1. Файлы

В следующих нескольких разделах мы рассмотрим файлы с точки зрения пользователя, то есть обсудим их использование и свойства.

5.1.1. Именование файлов

Файлы относятся к абстрактному механизму. Они предоставляют способ сохранять информацию на диске и считывать ее снова позднее. При этом от пользователя должны скрываться такие подробности, как способ и место хранения информации, а также детали работы дисков.

Вероятно, наиболее важной характеристикой любого абстрактного механизма является то, как именуются управляемые объекты, поэтому мы начнем изучение файловой системы с именования файлов. При создании файла процесс дает файлу имя. Когда процесс завершает работу, файл продолжает свое существование, и по его имени к нему могут получить доступ другие процессы.

Правила именования файлов варьируются от системы к системе, но все современные операционные системы поддерживают использование в качестве имен файлов 8-символьных текстовых строк. То есть andrea, bruce и cathy являются допустимыми именами файлов. Часто в именах файлов также разрешается употребление цифр и специальных символов, поэтому допустимы и такие имена, как 2, urgent! и Fig.2-14. Многие файловые системы поддерживают имена файлов длиной до 255 символов.

В некоторых файловых системах, например UNIX, различаются прописные и строчные буквы, тогда как в других, таких как MS-DOS, нет. Таким образом, имена maria, Maria и MARIA будут означать в системе UNIX три различных файла, и в то же время в MS-DOS все три имени будут соответствовать одному файлу.

Во многих операционных системах имя файла может состоять из двух частей, разделенных точкой, например prog.c. Часть имени файла после точки называется *расширением имени файла* и обычно характеризует его тип. Так, в MS-DOS имя файла может содержать от 1 до 8 символов плюс расширение от 0 до 3 символов. В UNIX размер расширения файла зависит от пользователя. Кроме того,

внешне имя файла может включать несколько расширений, например prog.c.Z, где обозначением .Z обычно отмечают, что файл (prog.c) был сжат с помощью алгоритма Лемпеля–Зива. Некоторые часто встречающиеся расширения имен файлов и их смысловая нагрузка представлены в табл. 5.1.

Таблица 5.1. Некоторые типичные расширения имен файлов

Расширение	Значение
file.bak	Резервная копия файла
file.c	Исходный текст программы на C
file.gif	Изображение в формате GIF
file.hlp	Файл справки
file.html	Документ в формате HTML (веб-страница)
file.jpg	Статическое изображение стандарта JPEG
file.mp3	Музыка в формате MPEG-1, уровень 3
file.mpg	Фильм в формате MPEG
file.o	Объектный файл (еще не скомпонованный выходной файл компилятора)
file.pdf	Документ формата PDF (программы Adobe Acrobat)
file.ps	Документ формата PostScript
file.tex	Входной файл для программы форматирования TEX
file.txt	Текстовый файл общего назначения
file.zip	Архив, сжатый с помощью алгоритма Лемпеля–Зива

В отдельных системах (например, в UNIX) расширения являются просто соглашениями, и операционная система не принуждает пользователя строго этих соглашений придерживаться. Файл file.txt может быть текстовым файлом, но это скорее напоминание пользователю, а не руководство к действию для операционной системы. С другой стороны, не исключено, что компилятор языка C откажется компилировать файлы с расширениями, отличными от .c.

Подобные соглашения особенно полезны, когда одна и та же программа должна управлять различными типами файлов. Например, интегрированному компилятору языка C может быть предоставлен список файлов, которые он должен преобразовать в объектный код и затем скомпоновать, причем некоторые из файлов могут содержать программы на языке C, тогда как другие могут быть ассемблерными файлами. В этом случае именно по расширениям компилятор отличает одни файлы от других.

5.1.2. Структура файла

Файлы могут быть структурированы несколькими различными способами. Три типа структур показаны на рис. 5.1. Файл на рис. 5.1, *а* представляет собой неструктурированную последовательность байтов. В данном случае операционная система не интересуется содержимым файла. Все, что она видит, — это байты.

Значения этим байтам присваиваются программами уровня пользователя. Такой подход характерен как для систем UNIX, так и в Windows.

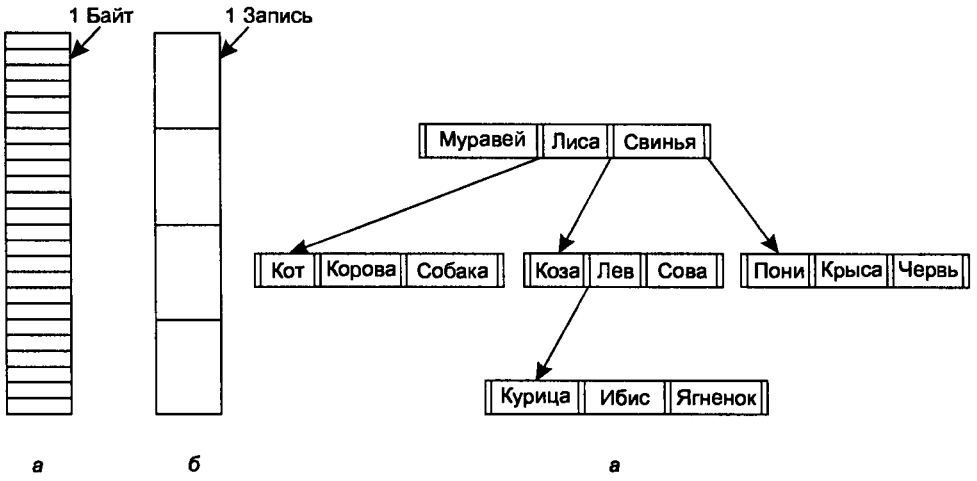


Рис. 5.1. Три типа файлов: *а* — последовательность байтов; *б* — последовательность записей; *а* — дерево

Рассмотрение операционной системой файлов как просто последовательностей байтов обеспечивает максимальную гибкость. Программы пользователя могут помещать в файлы все что угодно и именовать их любым удобным для них способом. Операционная система не вмешивается в этот процесс, что особенно ценно для пользователей, собирающихся сделать что-либо необычное.

Первый шаг по направлению к структуризации показан на рис. 5.1, *б*. В данной модели файл представляет собой последовательность записей фиксированной длины, каждая со своей внутренней структурой. Для файлов, состоящих из записей, важным является то, что операция чтения возвращает одну запись, а операция записи обновляет или дополняет одну запись. Несколько десятилетий назад, когда вовсю применялись перфокарты, состоящие из 80 колонок отверстий, многие операционные системы (на мэйнфреймах) оперировали файлами, состоящими из 80-символьных записей — образов перфокарт. Этими операционными системами поддерживались также файлы, составленные из 132-символьных записей, предназначенные для линейных принтеров (которые в те дни печатали по 132 символа в строке). В результате программы читали из входных файлов 80-символьные блоки и тут же расширяли их до 132-символьных блоков. Ни одна современная универсальная система не работает подобным образом.

Третий вариант файловой структуры представлен на рис. 5.1, *в*. При такой организации файл представляет собой дерево записей, не обязательно одной и той же длины. Каждая запись содержит поле *ключа* в фиксированной позиции. Дерево упорядочено по ключевому полю, с целью быстрого поиска по заданному ключу.

Основной файловой операцией здесь является не получение следующей записи, хотя это также возможно, а получение записи с указанным значением ключа. Для файла, показанного на рис. 5.1, *в*, можно, например, запросить у системы запись с ключом «пони», не беспокоясь о точном положении записи в файле (вольера в зоопарке). При добавлении новой записи операционная система, а не пользователь должна решать, куда ее поместить. Такой тип файлов принципиально отличается от неструктурированных потоков байтов, применяемых в UNIX и Windows. Как бы то ни было, они широко применяются на больших мэйнфреймах, еще используемых для коммерческой обработки данных.

5.1.3. Типы файлов

Многие операционные системы поддерживают различные типы файлов. Например, в системах UNIX и Windows проводится различие между обычными файлами и каталогами. В системе UNIX также различаются символьные и блочные специальные файлы. К *обычным* (regular) файлам относятся все файлы, содержащие информацию пользователя. Все файлы на рис. 5.1 являются просто файлами. *Каталоги* — это системные файлы, обеспечивающие структуризацию файловой системы. Мы рассмотрим их подробнее ниже. *Символьные специальные файлы* имеют отношение к вводу/выводу и используются для моделирования последовательных устройств ввода/вывода, таких как терминалы, принтеры и сети. *Блочные специальные файлы* находят применение в моделировании дисков. В данной главе мы в первую очередь будем рассматривать стандартные файлы.

Эти файлы в основном являются либо последовательностями ASCII-строк, либо двоичными наборами байтов. В некоторых системах каждая ASCII-строка завершается символом возврата каретки. В других (например, UNIX) используется символ перевода строки. Есть системы (например, MS-DOS), где используются оба символа. Строки не обязаны иметь одну и ту же длину.

Главным преимуществом ASCII-файлов является то, что они могут отображаться на экране и выводиться на печать «как есть», без какого-либо преобразования, и могут редактироваться любым текстовым редактором. Более того, если большое количество программ использует ASCII-файлы для входа и выхода, оказывается несложным соединить вход одной программы с выходом другой, как это делается в конвейерах оболочки. (Обмен данными между процессами не становится проще, но интерпретация информации облегчается, если ее представление стандартизировано.)

Если файлы не являются ASCII-файлами, они причисляются к двоичным. При выводе их на принтер получается невразумительный набор символов, напоминающий свалку мусора. Но в действительности у них есть некая внутренняя структура, известная использующей их программе.

Например, на рис. 5.2, *а* показан простой исполняемый двоичный файл из состава одной из версий системы UNIX. Хотя технически файл представляет собой всего лишь последовательность байтов, операционная система станет исполнять его только в том случае, если этот файл имеет соответствующий формат. Файл состоит из пяти разделов: заголовка, текста, данных, данных переадреса-

ции и таблицы символов. Заголовок начинается с так называемого *«магического» числа*, идентифицирующего файл как исполняемый (чтобы предотвратить случайный «запуск» файла другого формата). Следом за сигнатурой в заголовке располагаются поля с размерами различных частей файла, адресом точки входа файла и некоторые флаговые биты. За заголовком следуют текст программы и данные. Они загружаются в оперативную память и настраиваются на работу по адресу загрузки при помощи данных переадресации. Таблица символов используется для отладки.

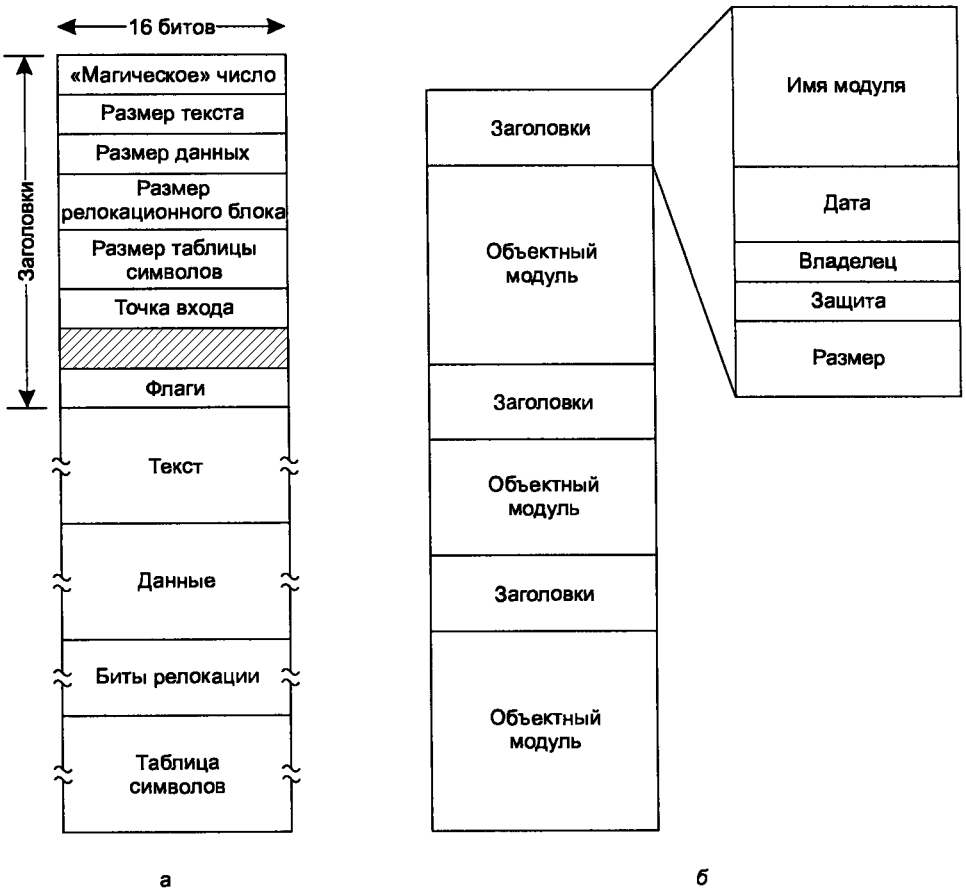


Рис. 5.2. *а* — исполняемый файл; *б* — архив

Второй пример двоичного файла представляет собой файл архива, также из системы UNIX. Он состоит из набора библиотечных процедур (модулей), откомпилированных, но не скомпонованных. Каждой процедуре предшествует заголовок, содержащий ее имя, дату создания, имя владельца, код защиты и размер. Как и в случае исполняемого файла, заголовки модулей хранят большое количе-

ство двоичных чисел. Если в таком виде подать их на принтер, получится полная тарабарщина.

Все операционные системы должны распознавать по крайней мере один тип файлов — свои собственные исполняемые файлы, но некоторые операционные системы различают и другие типы файлов. Старая TOPS-20 (для компьютера DECsystem 20) даже анализировала время создания каждого предоставляемого ей на исполнение файла. Затем она находила исходный файл и проверяла, не был ли он изменен, после того как был создан исполняемый файл. Если оказывалось, что исполняемый файл уже устарел, операционная система автоматически перекомпилировала исходный файл. В переводе на язык понятий UNIX — программа make была встроена в оболочку. Расширения имен файлов были обязательными, чтобы операционная система могла определить, какая двоичная программа от какого исходного файла произошла.

Однако такая жесткая привязка типов файлов к содержимому может оказаться неудобной для пользователя, пытающегося сделать что-нибудь не предусмотренное проектировщиками операционной системы. Представьте, например, систему, в которой файлы программного вывода автоматически получают расширение .dat (файлы данных). Пусть пользователь написал программу, форматирующую исходные тексты программ на C. Программа читает файл с расширением .c, обрабатывает его и затем сохраняет результат в файле со стандартным расширением .dat. Если пользователь затем попытается предложить этот файл C-компилятору, операционная система не позволит его скомпилировать, так как у файла для данного действия неверное расширение. Попытка скопировать file.dat в file.c также будет отвергнута операционной системой.

Хотя такая «дружественность» по отношению к пользователю («защита от дурака») может быть полезна для новичков, она ставит опытных пользователей в безвыходное положение, заставляя их тратить массу усилий на попытки перехитрить операционную систему.

5.1.4. Доступ к файлам

В старых операционных системах предоставлялся только один тип доступа к файлам — *последовательный*. В этих системах процесс мог читать байты или записи файла только по порядку от начала к концу. Такой доступ к файлам исходит из времен, когда дисков еще не было и компьютеры оснащались магнитофонами. Поэтому даже в дисковых операционных системах при последовательном доступе к файлу имитировалось его чтение или запись на ленточный накопитель с возможностью перемотки назад.

С появлением дисков стало возможным читать байты или записи файла в произвольном порядке или получать доступ к записям по ключу. Файлы, байты которых могут быть прочитаны в произвольном порядке, называются *файлами произвольного доступа*. Такие файлы используются многими приложениями.

Файлы произвольного доступа очень важны для ряда приложений, например для баз данных. Если клиент звонит в авиакомпанию с целью зарезервировать место на конкретный рейс, программа резервирования авиабилетов должна

иметь возможность получить доступ к нужной записи, не читая все тысячи предшествующих записей с информацией о других рейсах.

Место начала чтения указывается двумя способами. В первом случае каждая операция read неявно устанавливает позицию в файле. Во втором варианте используется специальная операция seek, устанавливающая новую текущую позицию. После выполнения операции seek файл можно читать последовательно с текущей позиции.

В некоторых старых операционных системах, работающих на мэйнфреймах, способ доступа к файлу (последовательный или произвольный) указывался в момент создания файла. Это позволяло операционной системе применять различные методы для хранения файлов разных классов. В современных операционных системах такого различия не проводится. Все файлы автоматически являются файлами произвольного доступа.

5.1.5. Атрибуты файла

У каждого файла есть имя и данные. Помимо этого, все операционные системы связывают с каждым файлом также и другую информацию, например дату и время создания файла, а также его размер. Мы будем называть эти дополнительные сведения *атрибутами файла*. Список атрибутов значительно варьируется от системы к системе. В табл. 5.2 показаны некоторые возможные атрибуты, однако существуют также и другие. На практике ни в одной операционной системе не используются сразу все приведенные в таблице атрибуты файлов, но каждый из них встречается в той или иной системе.

Первые четыре атрибута относятся к защите файла и содержат информацию о том, кто вправе получить доступ к файлу, а кто нет. Возможны различные схемы реализации защиты файла, несколько из них мы рассмотрим ниже. В некоторых системах пользователь должен для получения доступа к файлу указать пароль. В этом случае пароль должен входить в атрибуты файла.

Флаги представляют собой биты или короткие поля, управляющие некоторыми специфическими свойствами. Например, скрытые файлы не появляются в перечне файлов при распечатке каталога. Флаг архивации представляет собой бит, следящий за тем, была ли создана для файла резервная копия. Этот флаг очищается программой архивирования и устанавливается операционной системой при изменении файла. Таким образом, программа архивирования может определить, какие файлы следует архивировать. Флаг «временный» позволяет автоматически удалять помеченный им файл по окончании работы создавшего файл процесса.

Атрибуты длины записи, позиции ключа и длины ключа присутствуют только у тех файлов, записи которых могут искаться по ключу.

Различные атрибуты, хранящие значения времени, позволяют следить за тем, когда файл был создан, в последний раз изменен и когда к нему в последний раз предоставлялся доступ. Эти сведения можно использовать в различных целях. Например, если исходный файл программы был модифицирован после создания соответствующего ему объектного файла, данный исходный файл должен быть перекомпилирован.

Таблица 5.2. Некоторые возможные атрибуты файлов

Атрибут	Значение
Защита	Кто и каким образом может получить доступ к файлу
Пароль	Пароль для получения доступа к файлу
Создатель	Идентификатор пользователя, создавшего файл
Владелец	Текущий владелец
Флаг «только чтение»	0 — для чтения/записи; 1 — только для чтения
Флаг «скрытый»	0 — нормальный; 1 — не показывать в перечне файлов каталога
Флаг «системный»	0 — нормальный; 1 — системный
Флаг «архивный»	0 — файл помещен в резервное хранилище; 1 — требуется резервное копирование
Флаг ASCII/двоичный	0 — ASCII; 1 — двоичный
Флаг произвольного доступа	0 — только последовательный доступ; 1 — произвольный доступ
Флаг «временный»	0 — нормальный; 1 — для удаления файла по окончании работы процесса
Флаги блокировки	0 — неблокированный; отличный от нуля для блокированного
Длина записи	Количество байтов в записи
Позиция ключа	Смещение до ключа в записи
Длина ключа	Количество байтов в поле ключа
Время создания	Дата и время создания файла
Время последнего доступа	Дата и время последнего доступа к файлу
Время последнего изменения	Дата и время последнего изменения файла
Текущий размер	Количество байтов в файле
Максимальный размер	Количество байтов, до которого можно увеличивать размер файла

Текущий размер файла указывает количество байтов в файле в настоящий момент. В некоторых старых операционных системах мэйнфреймов при создании файла требовалось указать также максимальную длину файла, что позволяло операционной системе зарезервировать достаточно места для последующего увеличения файла. Современные операционные системы, работающие на персональных компьютерах, умеют обходиться без подобного резервирования.

5.1.6. Операции с файлами

Файлы позволяют сохранять информацию и получать ее позднее. В различных операционных системах имеются различные наборы файловых операций. Ниже перечислены наиболее часто встречаемые системные вызовы, имеющие отношение к файлам.

- Create (создание). Файл создается без данных. Этот системный вызов объявляет о появлении нового файла и позволяет установить некоторые его атрибуты.
- Delete (удаление). Когда файл уже более не нужен, его удаляют, чтобы освободить пространство на диске. Этот системный вызов присутствует в каждой операционной системе.
- Open (открытие). Прежде чем использовать файл, процесс должен его открыть. Системный вызов open позволяет системе прочитать в оперативную память атрибуты файла и список дисковых адресов для быстрого доступа к содержимому файла при последующих вызовах.
- Close (закрытие). Когда все операции с файлом закончены, атрибуты и дисковые адреса становятся не нужны, поэтому файл следует закрыть, чтобы освободить пространство во внутренней таблице системы. Многие операционные системы позволяют одновременно открыть лишь ограниченное количество файлов. Запись на диск производится поблочно, а закрытие файла вызывает запись последнего блока файла, даже если этот блок еще не заполнен до конца.
- Read (чтение). Чтение данных из файла. Обычно байты поступают с текущей позиции в файле. Считывающий процесс должен указать количество требуемых данных и предоставить для них буфер.
- Write (запись). Запись данных в файл, также в текущую позицию в файле. Если текущая позиция находится в конце файла, размер файла автоматически увеличивается. В противном случае запись производится поверх существующих данных, которые теряются навсегда.
- Append (добавление). Этот системный вызов представляет собой усеченную форму вызова write. Он может только добавлять данные к концу файла. Данный вызов может отсутствовать в операционных системах с минимальным набором системных вызовов.
- Seek (позиционирование). Для файлов произвольного доступа требуется способ указать, где располагаются данные в файле. Эта операция устанавливает указатель текущей позиции на определенное место файла. Последующие данные будут считаны из этой позиции и записаны в нее.
- Get attributes (получение атрибутов). Процессам часто требуются атрибуты интересующих их файлов. Например, для сборки программ, состоящих из большого числа отдельных исходных модулей, в системе UNIX часто используется программа make. Эта программа исследует время изменения всех исходных и объектных файлов, благодаря чему система обходится обработкой минимального их количества. Для выполнения своей работы программе требуется знать атрибуты файлов.
- Set attributes (установка атрибутов). Некоторые атрибуты файла могут устанавливаться пользователем после создания файла. Этот системный вызов предоставляет такую возможность. Например, для файла может быть установлен код защиты доступа и большинство других флагов.

- Rename (переименование). Позволяет изменить имя файла. Присутствие вызова в операционной системе не является необходимым, так как обычно файл можно скопировать с новым именем, а старый экземпляр удалить.

5.2. Каталоги

В файловых системах файлы обычно организуются в *каталоги*, которые, в свою очередь, в большинстве операционных систем также являются файлами. В данном разделе мы рассмотрим каталоги, их организацию, свойства и действия, которые могут быть выполнены над ними.

5.2.1. Иерархические системы каталогов

Обычно каталог содержит некоторое число записей, по одной записи на файл. Один из вариантов показан на рис. 5.3, *а*, где каждая запись каталога включает в себя имя файла, его атрибуты и адрес данных файла на диске. Другой вариант представлен на рис. 5.3, *б*. Здесь запись каталога хранит имя файла и указатель на структуру данных с атрибутами и дисковыми адресами. Широко применяются оба этих варианта.

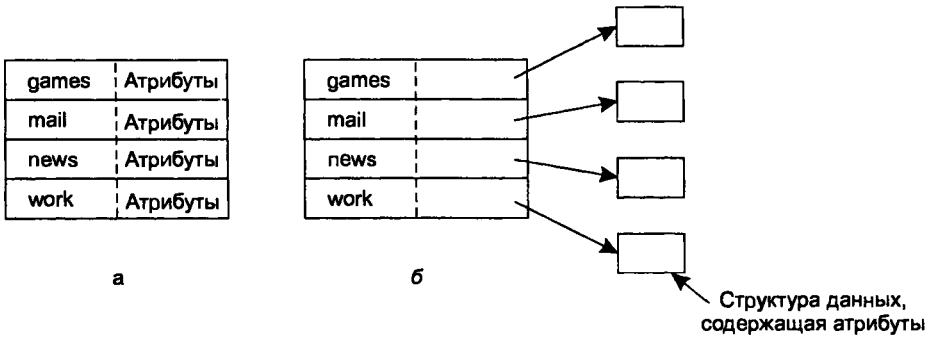


Рис. 5.3. *а* — атрибуты хранятся в каталоге; *б* — атрибуты хранятся отдельно

Когда открывается файл, операционная система ищет его запись в каталоге. Затем она извлекает и загружает в память атрибуты и дисковые адреса, либо из самой записи, либо из структуры, на которую запись ссылается. При всех последующих обращениях к файлу используется информация из памяти.

Количество каталогов меняется от системы к системе. В простейшем варианте имеется один каталог, в котором хранятся все файлы всех пользователей, согласно рис. 5.4, *а*. Если пользователей много и они для файлов выберут одинаковые имена, конфликты и путаница быстро приведут систему в непригодное для работы состояние. Подобная схема применялась в микрокомпьютерных системах, но сейчас ее редко можно увидеть.

Развитием этой идеи является двухуровневая иерархия, когда каждому пользователю выделяется отдельный каталог (рис. 5.4, *б*). Благодаря двухуровневой иерархии исчезают конфликты имен файлов между различными пользователями, но ее недостаточно для пользователей с большим числом файлов. Обычно пользователям бывает необходимо логически группировать свои файлы. Например, у профессора может быть набор файлов, образующих книгу, которую он пишет для одного курса, другое множество файлов, содержащее программы студентов для иного курса. Третий набор файлов может содержать исходные тексты разрабатываемого профессором нового компилятора, четвертая группа файлов — предложения различных грантов, а также электронная почта, расписание встреч, статьи, игры и т. д. Требуется некий гибкий способ, позволяющий объединять эти файлы в группы.

Следовательно, нужна некая общая иерархия (то есть дерево каталогов). При таком подходе каждый пользователь может сам создать себе столько каталогов, сколько ему нужно, группируя свои файлы естественным образом. Этот подход проиллюстрирован на рис. 5.4, *в*. Здесь каталоги A, B и C, вложенные в корневой каталог, принадлежат различным пользователям, два из которых создали подкаталоги для проектов, над которыми они работают.

Возможность создавать произвольное количество подкаталогов является мощным структурирующим инструментом, позволяющим пользователям организовать свою работу. По этой причине почти все современные файловые системы реализуются подобным образом.

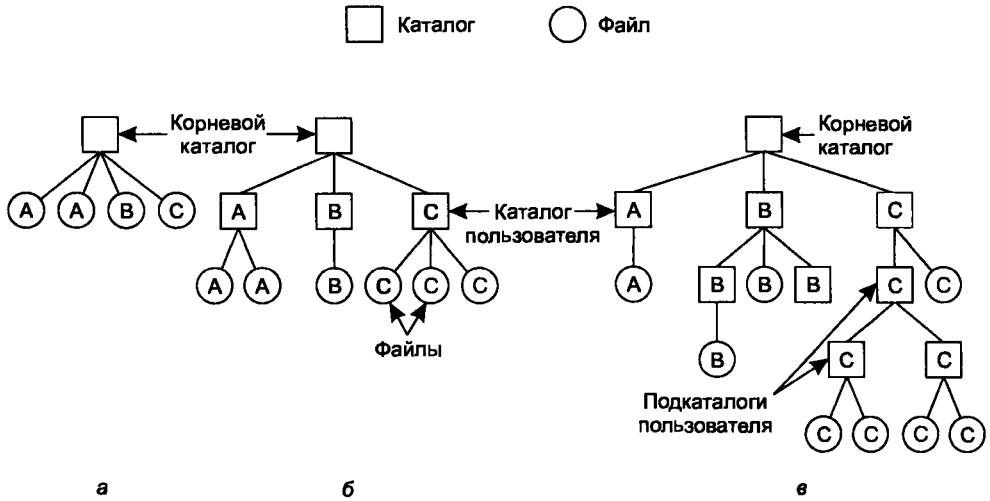


Рис. 5.4. Три варианта устройства файловой системы: *а* — один каталог на всех пользователей; *б* — отдельный каталог для каждого пользователя; *в* — дерево каталогов для каждого из пользователей. Буквами обозначены владельцы каталога или файла

5.2.2. Пути

При организации файловой системы в виде дерева каталогов требуется некоторый способ указания файла. Для этого обычно используется два различных метода. В первом случае обращение к файлу выполняется по *абсолютному имени пути*, составленному из имен всех каталогов от корневого до того, в котором содержится файл, и имени самого файла. Например, путь /usr/ast/mailbox означает, что корневой каталог содержит подкаталог usr, в который, в свою очередь, вложен подкаталог ast, где находится файл mailbox. Абсолютные имена путей всегда начинаются от корневого каталога и являются уникальными. В системе UNIX компоненты пути разделяются косой чертой /. В Windows в качестве разделителя принята обратная косая черта \. В системе MULTICS использовался символ >. Таким образом, одно и то же имя пути в этих трех операционных системах будет выглядеть следующим образом:

```
Windows\usr\ast\mailbox
UNIX/usr/ast/mailbox
MULTICS>usr>ast>mailbox
```

Если первой буквой имени пути был разделитель, это означало, независимо от используемого в качестве разделителя символа, что путь абсолютный.

Применяется и *относительное имя пути*. Оно используется вместе с концепцией *рабочего каталога* (также называемого *текущим каталогом*). Пользователь может назначить один из каталогов текущим рабочим каталогом. В этом случае все имена путей без начального символа разделителя считаются относительными и отсчитываются относительно текущего каталога. Например, если текущим каталогом является /usr/ast, тогда к файлу с абсолютным путем /usr/ast/mailbox можно обратиться просто как к mailbox. Другими словами, команда UNIX

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

и команда

```
cp mailbox mailbox.bak
```

выполнят одно и то же действие, если рабочим каталогом является /usr/ast. Относительная форма часто оказывается более удобной, но она подразумевает то же самое, что и абсолютная.

Некоторым программам бывает нужно получить доступ к файлам независимо от того, какой каталог является в данный момент текущим. В этом случае они всегда должны указывать абсолютные имена. Например, программе проверки правописания может понадобиться для выполнения работы прочитать файл /usr/lib/dictionary. В этом случае она должна использовать полное, абсолютное имя файла, так как она не знает, каким будет рабочий каталог при ее вызове. Абсолютное имя файла будет работать всегда, независимо от того, какой каталог является текущим в данный момент.

Если программе проверки правописания понадобится большое количество файлов из каталога /usr/lib, она может, обратившись к операционной системе, поменять рабочий каталог на /usr/lib, после чего указывать просто имя dictionary для первого аргумента системного вызова open. Явно указав свой рабочий каталог,

программа может использовать в дальнейшем относительные имена, поскольку точно знает, где она находится в дереве каталогов.

У каждого процесса есть свой рабочий каталог, поэтому, когда процесс меняет свой рабочий каталог и потом завершает работу, это не влияет на работу других процессов, и в файловой системе не остается никаких следов от подобных изменений. Таким образом, процесс может без опасений менять свой рабочий каталог, когда это ему удобно. С другой стороны, если *библиотечная процедура* поменяет свой рабочий каталог и не восстановит его при возврате управления, программа, вызвавшая ее, может оказаться не в состоянии продолжать свою работу, так как ее предположения о текущем каталоге окажутся неверными. По этой причине библиотечные процедуры редко меняют рабочие каталоги, а когда все-таки меняют, обязательно восстанавливают старое имя перед возвратом.

Большинство операционных систем, поддерживающих иерархические каталоги, имеют специальные элементы в каждом каталоге. Это «.» и «..», означающие текущий каталог и родительский каталог. Чтобы продемонстрировать, как это работает, обратимся к дереву каталогов системы UNIX, показанному на рис. 5.5.

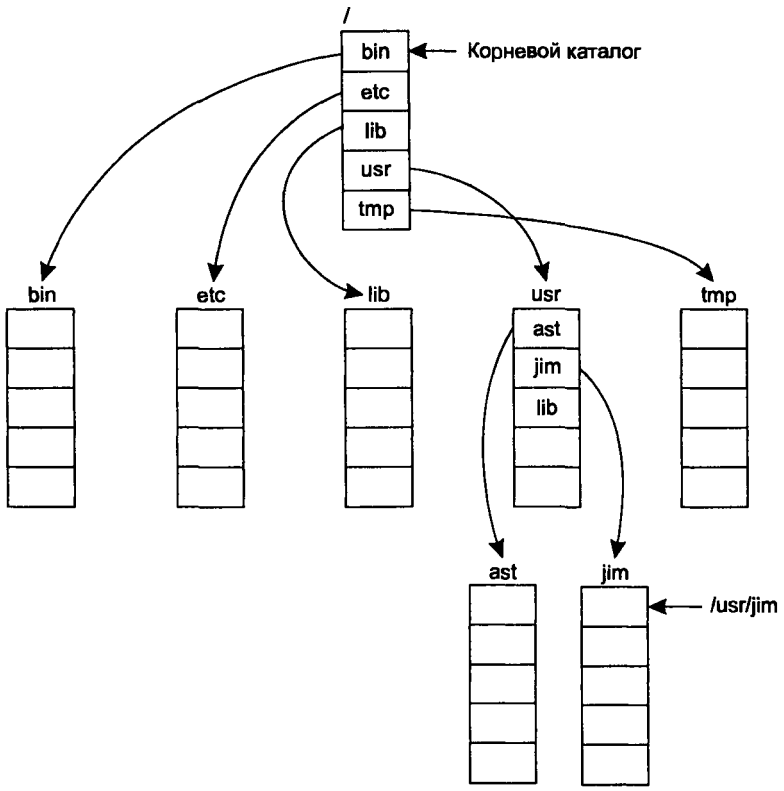


Рис. 5.5. Дерево каталогов UNIX

Для некоторого процесса каталог /usr/ast является рабочим. Чтобы переместиться вверх по дереву, он может использовать обозначение «..». Например, он может копировать файл /usr/lib/dictionary в свой собственный каталог при помощи команды

```
cp ../lib/dictionary
```

Две точки являются инструкцией системе подняться вверх (в каталог usr). После этого нужно открыть каталог lib и найти в нем файл dictionary.

5.2.3. Операции с каталогами

Системные вызовы, управляющие каталогами, значительно менее схожи в различных системах, чем системные вызовы для работы с файлами. Чтобы дать представление о том, что они собой представляют и как работают, приведем следующий пример (взятый из UNIX).

- Create. Создать каталог. Только что созданный каталог пуст и не содержит других элементов, кроме «.» и «..», автоматически помещаемых в каталог операционной системой.
- Delete. Удалить каталог. Может быть удален только пустой каталог. Элементы «.» и «..» файлами не являются и удалены быть не могут.
- Opendir. Открыть каталог. После этой операции каталог может быть прочитан. Например, для распечатки всех файлов каталога программа, создающая список, открывает каталог, чтобы прочитать имена всех содержащихся в нем файлов. Прежде чем каталог может быть прочитан, его следует открыть, подобно открытию и чтению файла.
- Closedir. Закрыть каталог. Когда каталог прочитан, его следует закрыть, чтобы освободить место во внутренней таблице системы.
- Readdir. Прочитать следующий элемент открытого каталога. В прежние времена было возможно читать каталоги с помощью обычного системного вызова read, но такой подход был небезопасен, так как требовал от программиста умения работать с внутренней структурой каталогов. Поэтому был создан отдельный системный вызов readdir, всегда возвращающий одну запись каталога стандартного формата независимо от используемой структуры каталогов.
- Rename. Переименование каталога. Во многих отношениях каталоги аналогичны файлам и могут переименовываться так же, как и файлы.
- Link. Связывание представляет собой технику, позволяющую файлу появляться сразу в нескольких каталогах. Этот системный вызов принимает в качестве входных параметров имя файла и имя пути и создает связь между ними. Таким образом, один и тот же файл может появляться сразу в нескольких каталогах. Подобная связь, увеличивающая на единицу счетчик *i*-узла файла (для учета количества каталогов со ссылками на этот файл), иногда называется *жесткой связью*.

- Unlink. Удаление ссылки на файл из каталога. Если файл присутствует только в одном каталоге, данный системный вызов удалит его из файловой системы. Если существует несколько ссылок на этот файл, будет удалена только указанная ссылка, а остальные останутся. Этот системный вызов применяется для удаления файла в операционной системе UNIX.

В приведенном выше списке перечислены наиболее важные системные вызовы, но существует также множество других, например, для управления защитой информации.

5.3. Реализация файловой системы

Теперь пора перейти от рассмотрения файловой системы с точки зрения пользователя к рассмотрению с точки зрения разработчика системы. Пользователей интересует то, как называются файлы, какие операции допустимы над ними, как выглядит дерево каталогов и тому подобные интерфейсные вопросы. Проектировщики файловых систем интересуются тем, как хранятся файлы и каталоги, как осуществляется управление дисковым пространством и как добиться надежной и эффективной работы файловой системы. В следующих разделах мы познакомимся с этим взглядом на файловую систему.

5.3.1. Реализация файлов

Вероятно, наиболее важным моментом в реализации хранения файлов является учет соответствия блоков диска файлам. Для определения того, какой блок какому файлу принадлежит, в различных операционных системах применяются различные методы. Некоторые из них будут рассмотрены в данном разделе.

Неразрывные файлы

Простейшей схемой выделения файлам определенных блоков на диске является система, в которой файлы представляют собой наборы последовательных соседних блоков диска. Тогда на диске с блоками по 1 Кбайт файл размером в 50 Кбайт будет занимать 50 последовательных блоков. При 2-килобайтовых блоках такой файл займет 25 смежных блоков.

У монолитных файлов есть два существенных преимущества. Во-первых, такую модель легко реализовать, так как системе, чтобы определить, какие блоки принадлежат тому или иному файлу, нужно следить всего лишь за двумя числами: номером первого блока файла и числом блоков в файле. Зная первый блок файла, любой другой его блок легко получить при помощи простой операции сложения.

Во-вторых, при работе с неразрывными файлами производительность просто превосходна, так как весь файл может быть прочитан с диска за одну операцию. Требуется только одна операция позиционирования (для первого блока). После этого более не нужно искать цилиндры и тратить время на ожидание поворота диска, поэтому данные могут считываться с максимальной скоростью, на какую

способен диск. Таким образом, непрерывные файлы легко реализуются и обладают высокой производительностью.

К сожалению, у такой методики есть и два существенных недостатка. Прежде всего, она непригодна, если максимальный объем файла заранее неизвестен. Не имея этой информации, файловая система не знает, сколько места резервировать. Правда, в системах, где файлы должны записываться за один раз, это может стать преимуществом.

Вторым недостатком является фрагментация дисков, к которой приводит подобная методика выделения. Впустую расходуется свободное пространство, которое можно было бы потратить с пользой. Уплотнение файлов с целью высвобождения места на диске обычно приводит к чрезмерно большим затратам, хотя, возможно, в ночное время, когда система все равно бездействует, это и не столь важно.

Списки

Второй метод размещения файлов состоит в представлении каждого файла в виде однонаправленного списка блоков диска, как показано на рис. 5.6. Первое слово каждого блока является указателем на следующий блок. В остальной части блока хранятся данные.

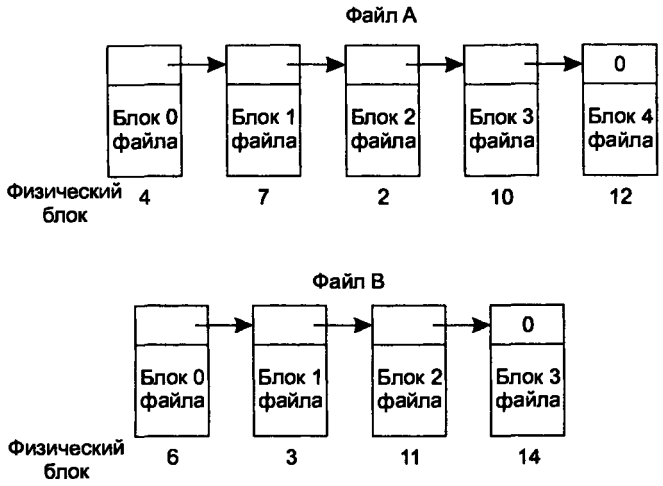


Рис. 5.6. Размещение файла в виде связного списка блоков диска

В отличие от систем с непрерывными файлами, такой метод позволяет использовать каждый блок диска. Нет потерь дискового пространства на фрагментацию (кроме потерь в последних блоках файла). Кроме того, в каталоге нужно хранить только адрес первого блока файла. Всю остальную информацию можно найти там.

С другой стороны, хотя последовательный доступ к такому файлу несложен, произвольный доступ будет довольно медленным. Чтобы получить доступ к бло-

ку n, операционная система должна сначала прочитать первые n – 1 блоков по очереди. Очевидно, такая схема оказывается очень медленной.

Список с индексацией

Оба недостатка предыдущей схемы организации файлов в виде списков могут быть устранены, если указатели на следующие блоки хранить не прямо в блоках, а в отдельной таблице, загружаемой в память. На рис. 5.7 показан внешний вид такой таблицы для файлов с рис. 5.6. На обоих рисунках показаны два файла. Файл A занимает блоки диска 4, 7, 2, 10 и 12, а файл B — блоки 6, 3, 11 и 14. С помощью таблицы мы можем начать с блока 4 и следовать по цепочке до конца файла. Те же действия применимы для второго файла, если начать с блока 6. Обе цепочки завершаются специальным маркером (например, 1), не являющимся допустимым номером блока. Такая таблица, загружаемая в оперативную память, называется *FAT* (File Allocation Table — таблица размещения файлов).

Рис. 5.7. Таблица размещения файлов

При такой организации все блоки доступны для данных. Кроме того, значительно упрощается произвольный доступ. Хотя для обращения к какому-либо блоку файла все равно понадобится проследовать по цепочке по всем ссылкам вплоть до требуемого блока, в данном случае вся цепочка ссылок уже хранится в памяти и ее прохождение не требует дополнительных дисковых операций. Как и в предыдущем случае, в каталоге достаточно хранить один целый индекс (номер начального блока файла) для обеспечения доступа ко всем частям файла.

Основной недостаток этого метода состоит в том, что вся таблица должна постоянно находиться в памяти. Для 500-мегабайтового диска с блоками размером 1 Кбайт потребовалась бы таблица из 500 000 записей, по одной для каждого из 500 000 блоков диска. Каждая запись должна состоять как минимум из трех байтов. Для ускорения поиска размер записей должен быть увеличен до 4 байт. Таким образом, резидентная таблица будет занимать 1,5 или 2 Мбайт оперативной памяти. Таблица, конечно, может быть размещена в виртуальной памяти, но и в этом случае ее размер оказывается чрезмерно большим, к тому же постоянная выгрузка таблицы на диск и загрузка ее с диска существенно снизят производительность файловых операций.

I-узлы

Последний метод отслеживания принадлежности блоков диска файлам заключается в связывании с каждым файлом структуры данных, называемой *i-узлом* (index node — индекс-узел), содержащей атрибуты файла и адреса блоков файла. Простой пример *i*-узла показан на рис. 5.8. При наличии *i*-узла можно найти все блоки файла. Большое преимущество такой схемы перед хранящейся в памяти таблицей из списков состоит в том, что каждый конкретный *i*-узел должен находиться в памяти только тогда, когда соответствующий ему файл открыт. Если каждый *i*-узел занимает n байтов, а одновременно открыть можно k файлов, для массива *i*-узлов потребуется в памяти всего kn байтов.

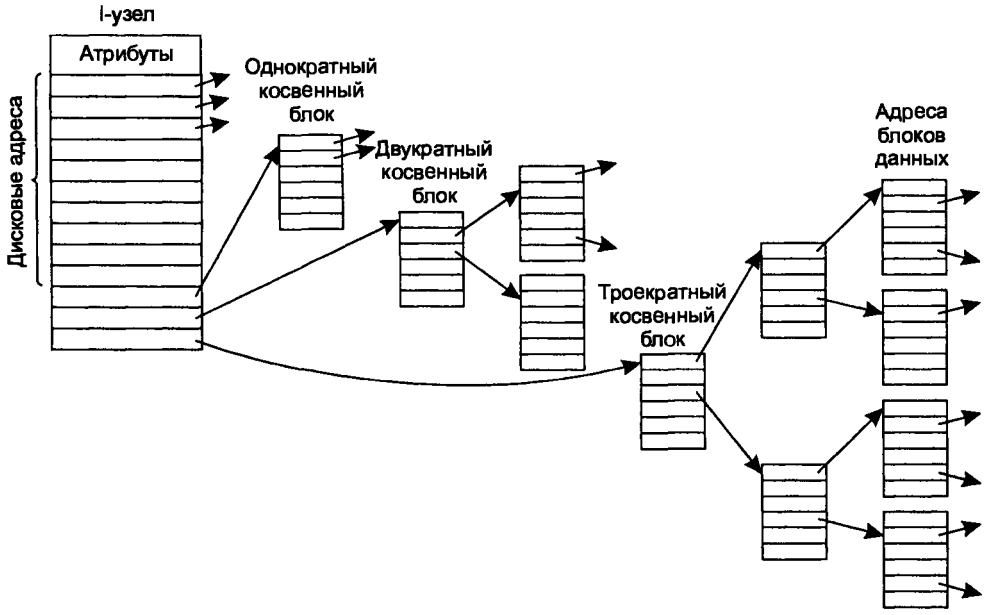


Рис. 5.8. Пример i-узла

Обычно эта величина значительно меньше, чем размер FAT, описанной в предыдущем разделе. Это легко объясняется. Размер таблицы, хранящей список всех блоков диска, пропорционален емкости самого диска. Для диска из n блоков потребуется n записей в таблице. Таким образом, размер таблицы линейно растет с ростом размера диска. Для схемы i-узлов, напротив, требуется массив в памяти с длиной, пропорциональной максимальному количеству файлов, которые можно открыть одновременно. При этом не важно, будет ли размер диска 1 Гбайт, 10 Гбайт или 100 Гбайт.

С такой схемой связана проблема, суть которой в том, что при выделении каждому файлу фиксированного количества дисковых адресов этого количества может не хватить. Одно из решений заключается в резервировании последнего дискового адреса не для блока данных, а для следующего адресного блока. Более того, можно создавать целые цепочки и даже деревья адресных блоков. Мы снова вернемся к теме i-узлов, когда приступим к изучению системы UNIX позднее.

5.3.2. Реализация каталогов

Прежде чем прочитать файл, его следует открыть. При открытии файла операционная система оперирует указанным пользователем именем пути, чтобы найти запись в каталоге. Запись в каталоге содержит информацию, необходимую для нахождения блоков диска. В зависимости от системы это может быть дисковый адрес всего файла (для монолитных файлов), номер первого блока файла (обе схемы списков) или номер i-узла. Во всех случаях основная функция каталоговой системы состоит в преобразовании ASCII-имени в информацию, необходимую для поиска данных.

С этой проблемой тесно связан вопрос хранения атрибутов файла. Каждая файловая система поддерживает различные атрибуты файла, такие как дату создания файла, имя владельца и т. д., и всю эту информацию нужно где-то хранить. Один из очевидных вариантов — поместить эти сведения непосредственно в запись каталога. Системы с i-узлами могут хранить атрибуты в i-узлах, а не в записях каталога. Как мы увидим позднее, у этого метода есть определенные преимущества по сравнению с помещением атрибутов прямо в записи каталогов.

Каталоги в CP/M

Мы начнем изучение каталогов с самого простого примера — файловой системы CP/M, проиллюстрированной на рис. 5.9. В этой системе только один каталог, поэтому файловой системе, чтобы найти файл, нужно просмотреть только этот единственный каталог. Обнаружив запись, файловая система получает информацию и о номерах блоков на диске, а также атрибуты файла, поскольку все это хранится в каталоге. Если файлу требуется больше дисковых блоков, чем помещается в одной записи, для него выделяются дополнительные.

Рассмотрим назначение полей на рис. 5.9. Поле Код пользователя помогает определить, какому пользователю принадлежит файл. При поиске файла проверяются только те записи, которые принадлежат текущему пользователю. Следующие два поля задают имя/расширение файла. Поле Экстент необходимо потому,

что файлы размером больше 16 блоков занимают несколько записей в каталоге. Это поле определяет, какая запись первая, какая вторая и т. д. Поле Число блоков сообщает, сколько блоков из 16 потенциально доступных занято. Последние 16 записей содержат сами номера блоков. Последний блок в файле может и не быть полным, и система не может точно определить размер файла (то есть она хранит размер в блоках, а не в байтах).

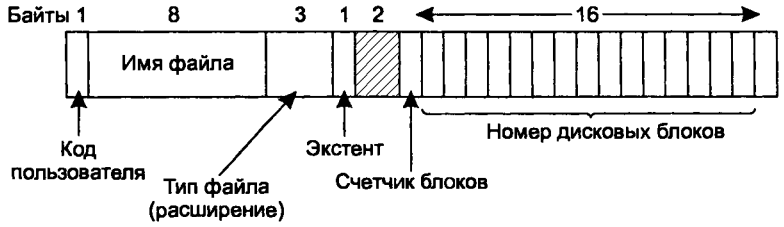


Рис. 5.9. Запись файла в каталоге содержит все номера его блоков

Каталоги в MS-DOS

Теперь давайте рассмотрим системы с иерархической структурой каталогов. На рис. 5.10 показана структура записи каталога в MS-DOS. Запись имеет размер 32 байта и содержит имя файла, его атрибуты и номер его первого блока. Этот номер используется как индекс в таблице на рис. 5.7. Остальные блоки можно найти, следуя по цепочке.

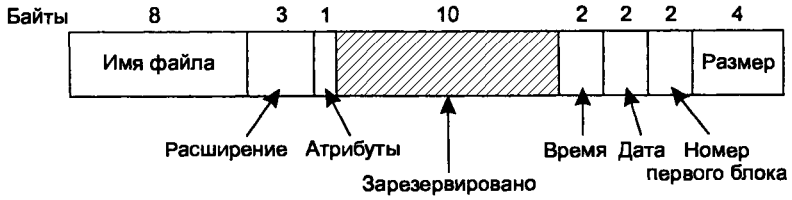


Рис. 5.10. Структура каталога в MS-DOS

В MS-DOS каталоги могут содержать в себе другие каталоги, образуя иерархическую структуру. В этой системе принято, чтобы каждая программа создавала свою ветку в корневом каталоге диска, в результате различные приложения не конфликтуют.

Каталоги в UNIX

В UNIX применяется исключительно простая структура каталогов, показанная на рис. 5.11. Здесь каждая запись состоит из имени файла и номера *i*-узла. Вся остальная информация, о размере файла, его типе, владельцах, времени изменения и занимаемых им дисковых блоках, хранится в *i*-узле. В некоторых из UNIX-систем применяется другая схема, но, в любом случае, элемент каталога состоит исключительно из ASCII-строки и номера *i*-узла.

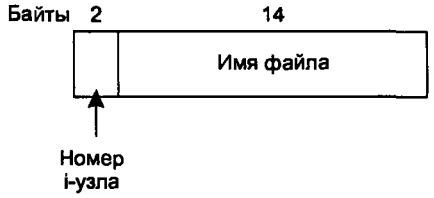


Рис. 5.11. Запись каталога в UNIX

Когда открывается файл, файловая система должна найти на диске указанное ей имя файла. Рассмотрим, как будет происходить поиск файла /usr/ast/mbox. В качестве примера мы взяли UNIX, но все сказанное относится и к другим иерархическим системам.

Сначала файловая система обнаруживает корневой каталог. В UNIX его *i*-узел расположен в фиксированном месте на диске. Затем она выделяет первый компонент пути, usr, и ищет в корневом каталоге номер *i*-узла для файла /usr. Обнаружить *i*-узел по его номеру несложно, так как расположение узлов фиксировано. Далее система продолжает поиск с этого *i*-узла и находит следующий компонент, ast. Обнаружив его, система получает номер *i*-узла для каталога /usr/ast. Наконец, в этом каталоге ищется сам файл mbox. Затем *i*-узел файла считывается в память и остается там до тех пор, пока файл не будет закрыт. Процесс поиска файла проиллюстрирован на рис. 5.12.

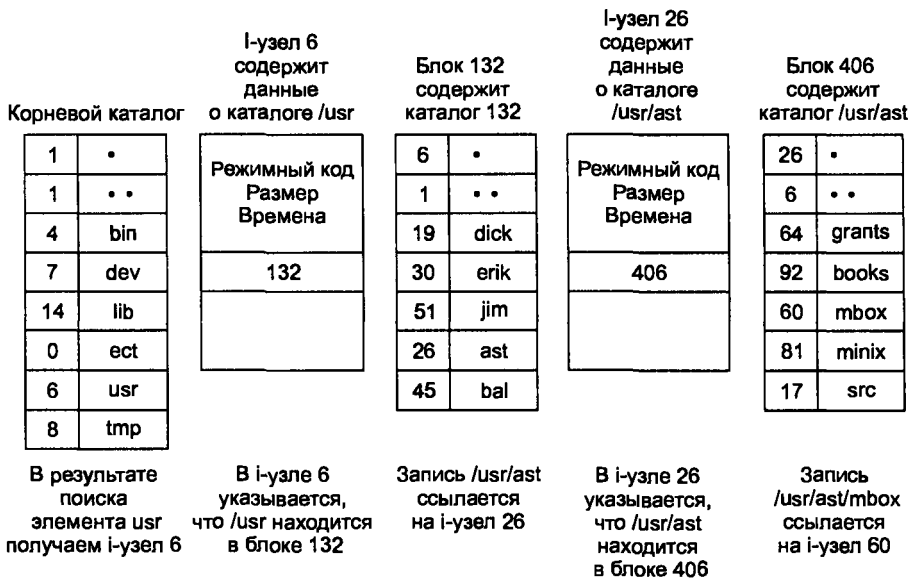


Рис. 5.12. Шаги поиска файла /usr/ast/mbox

Относительные пути обрабатываются точно так же, как и абсолютные, за исключением того, что поиск начинается не с корневого каталога, а с текущего.

В каждом каталоге есть записи с именами «.» и «..», которые добавляются при создании каталога. Записи «.» соответствует *i*-узел самого каталога, а «..» ссылается на *i*-узел каталога верхнего уровня. Таким образом, если дано имя файла ../dick/prog.c, система найдет «..» в текущем каталоге, получит *i*-узел родительского и будет искать в нем dick. Для обработки таких имен не применяется никаких специальных алгоритмов. С точки зрения системы каталогов, это обычные ASCII-строки, ничем не отличающиеся от прочих имен.

5.3.3. Организация дискового пространства

Обычно файлы хранятся на диске, поэтому организация дискового пространства является основной заботой разработчиков файловой системы. Для хранения файла из *n* байтов возможны две стратегии: выделение на диске *n* последовательных байтов или разбиение файла на несколько непрерывных блоков. Та же дилемма присутствует в системах управления памяти, где имеется выбор между чистой сегментацией и страничной организацией памяти.

Как уже было показано, при хранении файла в виде непрерывной последовательности байтов возникает проблема, связанная с увеличением его размеров. Единственный способ увеличить цельный файл состоит в перемещении его на новое место на диске. Проблема существенна и для управления сегментами памяти, с той разницей, что перемещение сегмента в памяти является более быстрой операцией по сравнению с перемещением файла на диске. По этой причине почти все файловые системы хранят файлы в виде блоков фиксированного размера, но не обязательно последовательных.

Размер блока

Как только принято решение хранить файлы блоками фиксированного размера, возникает вопрос о размере последних. Учитывая организацию дисков, очевидными кандидатами на роль блоков являются сектор, дорожка и цилиндр диска (минусом такого выбора является зависимость этих параметров от устройств). В системе управления страницами памяти размер страницы также входит в число основных претендентов.

Если выбрать большую единицу хранения, такую как цилиндр, это будет означать, что любой файл, даже состоящий из одного байта, займет как минимум целый цилиндр. Опыт показал, что средний размер файла в системе UNIX около 1 Кбайт, поэтому при выделении каждому файлу 32-килобайтового блока будет расходоваться понапрасну 31/32 или 97 % общего дискового пространства [90].

С другой стороны, при использовании маленьких единиц хранения каждый файл будет состоять из большого числа блоков. Для чтения каждого блока файла обычно требуется операция поиска нужного цилиндра и ожидание поворота диска, поэтому чтение файла, состоящего из большого числа блоков, окажется медленным.

Например, представьте себе диск с 131 072 байтами (128 Кбайт) на дорожку, периодом вращения 8,33 мс и средним временем поиска 10 мс. При этом время,

требующееся для чтения блока из k байтов, будет равно сумме времени поиска, ожидании поворота и времени переноса данных:

$$10 + 4{,}165 + (k/131\,072) \times 8{,}33.$$

Жирная кривая на рис. 5.13 показывает зависимость скорости передачи данных от размера блока. Если мы предположим, что файлы в основном имеют размер 1 Кбайт (медианный размер), то прерывистая линия на рис. 5.13 отразит эффективность задействования дискового пространства. Плохие новости в том, что эффективное использование места на диске (блок меньше 2 Кбайт) сопряжено с низкой пропускной способностью, и наоборот.

Поэтому обычно выбирается компромиссное решение, и размер блока делается равным 512 байт, 1 Кбайт или 2 Кбайт. Если сектор диска имеет размер 512 байт, а блока — 1 Кбайт, то файловая система всегда будет считывать или записывать два последовательных сектора, рассматривая их как единый, неделимый элемент. Какое бы решение ни было принято, его необходимо периодически пересматривать, так как по мере развития технологий пользователи, получив больше ресурсов, начинают требовать еще больше. Один из системных администраторов сообщил нам, что средний размер файлов в университетской системе медленно возрастал год за годом и к 1997 году достиг 12 Кбайт для студентов и 15 Кбайт для факультета.

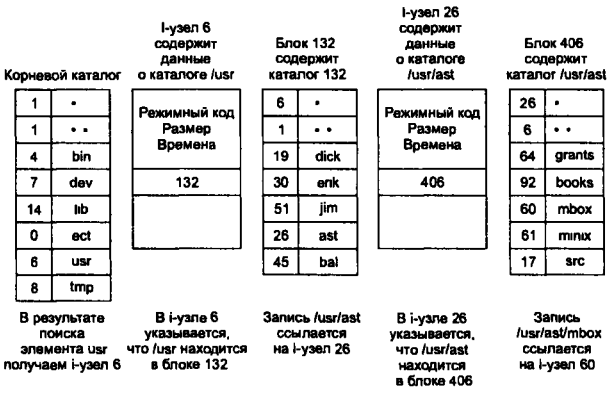


Рис. 5.13. Зависимость скорости чтения/записи данных диска (жирная линия, левая шкала) и эффективности использования дискового пространства (штриховая линия, правая шкала) от размера блоков. Все файлы по 2 Кбайт

Учет свободных блоков

После того как мы выбрали размер блоков, следует определиться, как учитывать свободные и занятые блоки. Широкое распространение получили два метода, представленные на рис. 5.14. Первый метод — не что иное, как использование списка блоков диска. При этом в каждом блоке списка содержится столько номеров свободных блоков, сколько может поместиться в один блок. При размере блока, равном 1 Кбайт, и 32-разрядных номерах блоков каждый блок списка свободных блоков может содержать номера 255 свободных блоков. (Одно 32-разрядное слово

нужно для указателя на следующий блок списка.) Для 16-гигабайтового диска потребуется список свободных блоков, состоящий из 16 794 блоков, чтобы охватить все 2^{24} дисковых блока. Часто для списка резервируется нужное число блоков в начале диска.

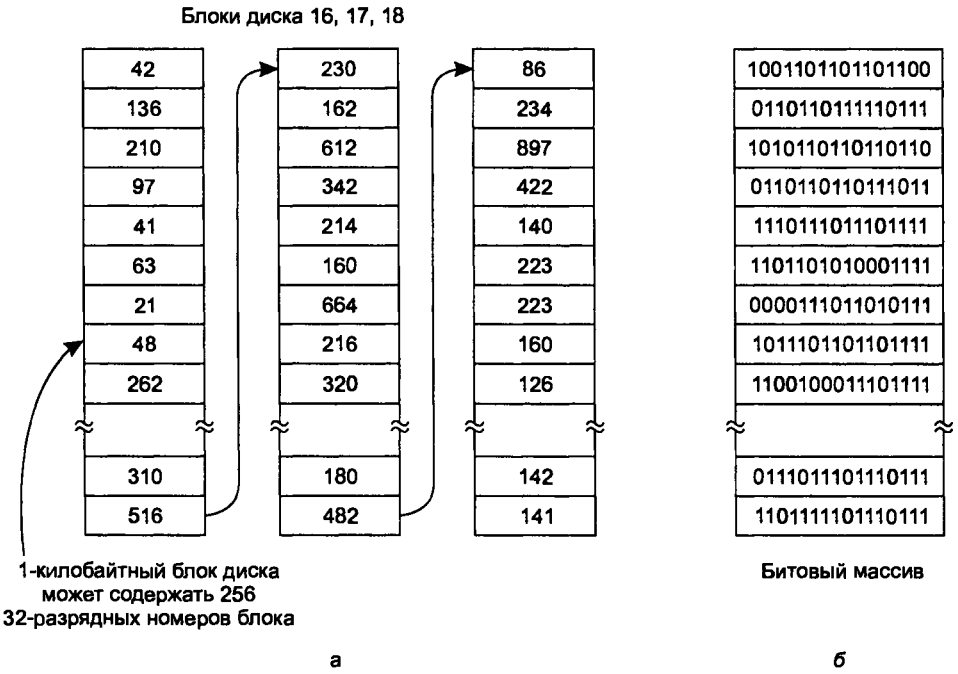


Рис. 5.14. *а* — хранение информации о свободных блоках в виде списка; *б* — битовый массив

Другой метод учета свободного дискового пространства заключается в хранении этой информации в виде битового массива (битовой карты). Здесь на каждый блок приходится всего по одному биту вместо 32. Свободные блоки обозначаются в массиве единицами, а занятые — нулями (или наоборот). 16-гигабайтовый диск состоит из 2^{24} килобайтовых блоков, таким образом, для него требуется массив размером 2^{24} бит, то есть 2048 блоков, что меньше, чем потребовалось бы для организации списка. Схема на основе списка более эффективна только тогда, когда диск практически полон.

Если памяти компьютера достаточно для хранения битовой карты, этот метод в целом предпочтителен. Когда же для учета свободных блоков на диске можно выделить только один блок памяти и практически все блоки заняты, списки становятся более эффективными. Если хранить в основной памяти только один блок битовой карты, может оказаться, что в ней не получится найти свободных блоков и придется считывать еще одну часть битовой карты. Когда в память загружается новый блок списка, перед переходом к следующему блоку списка можно будет выделить 255 блоков данных.

5.3.4. Надежность файловой системы

Разрушение файловой системы часто оказывается большим бедствием, чем поломка компьютера. Если компьютер приходит в негодность вследствие пожара, удара молнии или того, что пользователь прольет чашку кофе на клавиатуру, это неприятно, ремонт потребует денег, но обычно не причинит много хлопот. Недорогие персональные компьютеры можно заменить в течение часа, обратившись к соответствующему коммерсанту. (Исключение составляют университеты, где для приобретения персональных компьютеров требуется согласование этого вопроса в трех инстанциях, получение пяти подписей и 90 дней ожидания.)

В случае же потери файловой системы по вине аппаратуры или программного обеспечения или крыс, посчитавших, что одного отверстия на гибком диске недостаточно, восстановление всей информации будет трудным, требующим много времени, а часто и невозможным делом. Для пользователей, чьи программы, документы, файлы клиентов, счета, базы данных, маркетинговые планы или другие данные утеряны навсегда, последствия могут оказаться катастрофическими. Хотя операционная система не в силах защитить от физического уничтожения оборудование или носитель, она в состоянии помочь сберечь информацию. В данном разделе мы рассмотрим некоторые вопросы, касающиеся защиты файловой системы от уничтожения.

Как уже указывалось в главе 3, на дисках могут быть дефекты. Когда гибкие диски покидают фабрику, их качество, как правило, превосходно, но со временем на них могут появиться дефектные блоки. У жестких дисков такие блоки часто бывают врожденными, поскольку производство жестких дисков абсолютно без дефектов чересчур дорого. Фактически, старые жесткие диски обычно поставлялись со списком поврежденных блоков, обнаруженных при помощи тестов производителя. На таких дисках под список поврежденных блоков резервировался специальный сектор. Контроллер при первой инициализации считывал список поврежденных блоков и подменял их запасными блоками (или дорожками). После этого все запросы к поврежденным блокам перенаправлялись на запасные. По мере того как обнаруживались новые ошибки, список обновлялся в ходе низкоуровневого форматирования.

Технологии уверенно улучшаются, поэтому сейчас поврежденные блоки не так распространены, как раньше. Тем не менее они все еще встречаются. В главе 3 говорилось, что контроллер современного жесткого диска очень сложен. На таких дисках на каждой дорожке есть по крайней мере один дополнительный сектор, то есть как минимум одно повреждение можно пропустить, оставив зазор между двумя последовательными секторами. Кроме того, в каждом цилиндре есть несколько запасных секторов, которые контроллер может задействовать, если обнаружит, что для записи или чтения какого-либо сектора требуется слишком большое число попыток. Таким образом, пользователь обычно ничего не подозревает о поврежденных блоках и их учете. Однако, если происходит отказ современного IDE- или SCSI-диска, это обычно серьезная проблема, означающая, что у диска кончились запасные секторы. SCSI-диски, заменяя сбойный блок запасным, сообщают о «восстановлении после сбоя» (recovered error). Заметив

это, драйвер может напечатать на экране сообщение. Если такие сообщения появляются на экране часто, пришло время менять диски.

Существует простое программное решение проблемы сбойных блоков, применимое для старых дисков. Это решение сводится к тому, что ОС аккуратно составляет файл, содержащий в себе все такие блоки. Благодаря такому подходу блоки исключаются из списка свободных и никогда не будут задействованы для хранения данных. Если не делать к данному файлу обращений, никаких проблем (за исключением особых случаев) не возникнет. Соответственно, во время резервного копирования нужно избегать чтения этого файла.

Резервные копии

Большинство пользователей считает создание резервных копий файлов просто потерей времени. Однако когда в один прекрасный день их диск внезапно приказывает долго жить, они диаметрально меняют свои привычки. Компании, напротив, обычно хорошо осознают ценность своей информации и выполняют резервное копирование один раз в сутки, чаще всего на магнитную ленту. Современные магнитные ленты вмещают десятки и иногда даже сотни гигабайтов при цене в несколько центов за гигабайт. Тем не менее создание резервных копий является далеко не столь тривиальным делом, как это может показаться, поэтому мы ниже рассмотрим некоторые аспекты данной темы.

Создать резервную копию файловой системы на дискете можно, просто скопировав все содержимое дискеты на другую. Файлы на винчестере можно архивировать, скопировав их на магнитную ленту. Современные технологии предлагают магнитные ленты весьма большого объема.

Но для больших жестких дисков копирование всего содержимого на ленту неудобно, к тому же, это занимает много времени. Одной из легко реализуемых методик является применение двух дисков вместо одного, с тем недостатком, что свободное место используется менее эффективно. Пространство на каждом диске разбивается на две части: данные и резервная копия. Каждую ночь данные с диска 0 копируются в резервную область на диске 1, и наоборот, как показано на рис. 5.15. Таким образом, если один из дисков выйдет из строя, информация потеряна не будет.

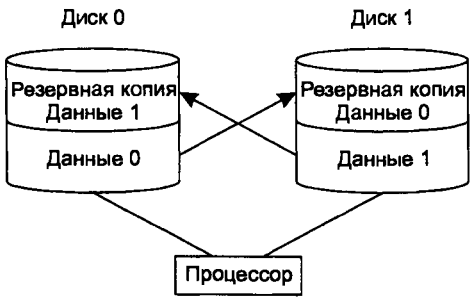


Рис. 5.15. Резервное копирование одного диска на другой приводит к двойному расходу свободного пространства

На практике применяется идея *инкрементных резервных копий*, или, как их еще называют, инкрементных дампов, являющаяся альтернативой полному копированию всей файловой системы. Простейшая форма инкрементного (наращиваемого) архивирования состоит в том, что полная резервная копия создается, скажем, раз в неделю или раз в месяц, а ежедневно сохраняются только те файлы, которые изменились с момента последней полной архивации. Еще лучше архивировать только те файлы, которые изменились со времени последней архивации.

Для реализации такого подхода необходимо хранить список времени последнего копировании для каждого из файлов. Если файл оказался изменен относительно своего дубликата, он снова копируется, и обновляется время копирования в таблице. Если цикл копирования равен одному месяцу, потребуется 31 лента на каждый день, плюс столько лент, сколько нужно для полного ежемесячного дампа. Применяются и другие, более сложные схемы с меньшим числом лент.

Кроме того, применяются автоматические методы, основанные на использовании нескольких дисков. Данные записываются на оба диска, а чтение происходит с одного, при этом запись на второй диск немного задерживается и может выполняться, когда система бездействует. Если один из дисков выйдет из строя, его можно заменить на новый и восстановить информацию.

Целостность файловой системы

Еще одним аспектом проблемы надежности является непротиворечивость файловой системы. Файловые системы обычно читают блоки данных, модифицируют их и записывают их обратно. Если в системе произойдет сбой прежде, чем все модифицированные блоки будут записаны на диск, файловая система может оказаться в противоречивом состоянии. Эта проблема становится особенно важной в случае, если одним из модифицированных и не сохраненных блоков оказывается блок *i*-узла, каталога или списка свободных блоков.

Для обеспечения целостности файловой системы большинство компьютеров оснащаются специальной обслуживающей программой, проверяющей состояние файловой системы. Ниже будет описано, как работает подобная проверочная утилита в UNIX и MINIX. В других системах есть похожие программы. Все они проверяют различные файловые системы (дисковые разделы) независимо друг от друга.

Обычно контролируется непротиворечивость двух типов: блоков и файлов. При проверке целостности блоков программа создает две таблицы, каждая из которых содержит счетчик для каждого блока, изначально установленный в 0. Счетчики в первой таблице фиксируют, в каком количестве каждый блок присутствует в файле. Счетчики во второй таблице содержат значения, сколько раз каждый блок учитывается в списке свободных блоков (или в битовом массиве).

Затем программа считывает все *i*-узлы. Начиная с *i*-узла, можно построить список всех номеров блоков, занятых соответствующим файлом. При считывании каждого номера блока счетчик этого блока увеличивается на единицу. Затем программа анализирует список или битовый массив свободных блоков, чтобы

обнаружить все неиспользуемые блоки. Каждый раз, встречая номер блока в списке свободных блоков, программа инкрементирует соответствующий счетчик во второй таблице.

Если файловая система непротиворечива, каждый блок будет встречаться только один раз, либо в первой, либо во второй таблице, как показано на рис. 5.16, *а*. Однако в результате сбоя эти таблицы могут принять вид, соответствующий рис. 5.16, *б*. В этом случае блок два отсутствует в каждой таблице. О таком блоке программа сообщит как о *недостающем*. Хотя пропавшие блоки не причиняют вреда, они занимают место на диске, снижая его емкость. Учесть же пропавшие блоки очень просто: программа проверки файловой системы просто добавляет их к списку свободных.

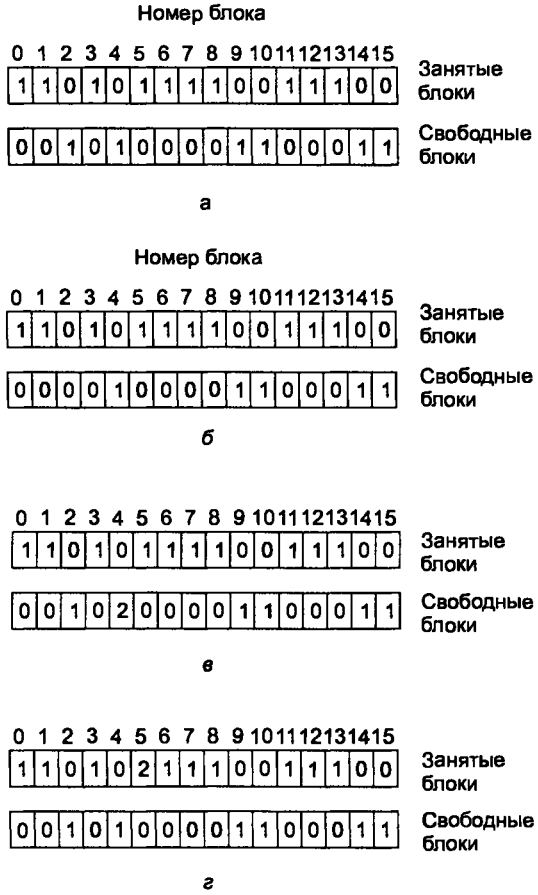


Рис. 5.16. Состояния файловой системы: *а* — непротиворечивое; *б* — пропавший блок; *в* — дубликат блока в списке свободных блоков; *г* — дубликат блока данных

Другая возможная ситуация показана на рис. 5.16, *в*. Здесь мы видим блок номер 4, дважды появляющийся в списке свободных. (Дубликаты свободных блоков могут появиться лишь тогда, когда в файловой системе действительно применены списки свободных блоков; в случае битового массива это невозможно.) Решение этой проблемы также несложно: построить список свободных блоков заново.

Гораздо хуже вариант, где один и тот же блок окажется сразу в двух файлах, как показано на рис. 5.16, *г* в случае с блоком 5. При удалении любого из этих файлов блок 5 будет помещен в список свободных блоков, что приведет к ситуации, в которой один и тот же блок одновременно является и свободным и занятым. Если удалить оба файла, этот блок будет помещен в список свободных блоков дважды.

В такой ситуации программа проверки файловой системы должна взять свободный блок, скопировать в него содержимое блока 5 и вставить эту копию в один из файлов. Таким образом, содержимое файлов останется неизменным (хотя почти наверняка один из файлов уже испорчен), но, по крайней мере, структура файловой системы после этой операции становится непротиворечивой. Программа также должна выдать сообщение об ошибке, чтобы пользователь мог изучить повреждение.

Помимо проверки правильности принадлежности блоков, программа проверки также анализирует каталоговую структуру. Для этого также используется таблица счетчиков, но уже не для блоков, а для файлов. Проверка начинается с корневого каталога с рекурсивным заходом в каждый каталог. Для каждого файла в каждом каталоге программа увеличивает на единицу счетчик использования файла. Благодаря жестким связям файл может присутствовать сразу в нескольких каталогах. Символьные связи не учитываются и не оказывают влияния на счетчик.

Когда сканирование дерева каталогов завершено, программа получает список, индексированный по номерам *i*-узлов, сообщающий, в скольких каталогах присутствует каждый файл. Затем программа сравнивает полученные числа со счетчиками связей, хранящимися в самих *i*-узлах. Эти счетчики содержат единицу при создании файла и инкрементируются всякий раз, когда создается связь (жесткая) с данным файлом. В непротиворечивой файловой системе оба счетчика должны совпадать. Однако возможны два типа ошибок: значение счетчика связи в *i*-узле может оказаться слишком велико или слишком мало.

Если счетчик связи больше, чем количество записей в каталоге, тогда даже при удалении всех файлов из каталогов счетчик все равно не уменьшится до нуля, и *i*-узел не будет удален. Эта ошибка не серьезная, но она приводит к расходованию дискового пространства файлом, находящимся вне всех каталогов. Чтобы исправить ее, следует установить значение счетчика равным числу существующих записей каталога.

Вторая ошибка таит в себе катастрофические последствия. Если у файла есть две каталоговые записи, связанные с ним, но в *i*-узле утверждается, что описатель у файла только один, тогда при удалении описателя этого файла в любом

каталоге счетчик *i*-узла уменьшится до нуля. При этом файловая система освободит все блоки, занимаемые файлом, в том числе и блок, в котором помещается сам *i*-узел. Таким образом, в одном из каталогов сохранится дескриптор файла, указывающий на неиспользуемый *i*-узел, чьи блоки могут быть вскоре выделены другим файлам. Решение здесь также заключается в присваивании значения счетчика *i*-узла фактическому числу дескрипторов файла.

Часто эти две операции, проверки блоков и проверки каталогов, для увеличения эффективности объединяют в один проход. Возможно также проведение и других проверок. Например, формат каталогов должен соответствовать определенным требованиям относительно *i*-узлов и ASCII-имен. Если *i*-узел оказывается больше числа *i*-узлов на диске, это означает, что каталог поврежден.

Более того, у каждого *i*-узла могут оказаться значения режима доступа, являющиеся допустимыми, но странными, как, например, 0007. Такое значение совсем отказывает в доступе владельцу и его группе, но зато разрешает всем посторонним читать, писать и исполнять файл. Программа должна хотя бы сообщать обо всех файлах, предоставляющих сторонним пользователям больше прав, чем владельцам. Каталоги, содержащие, скажем, более 1000 описателей файлов, также подозрительны. Расположенные в каталогах пользователей файлы, владельцем которых является суперпользователь и у которых установлен бит SETUID, представляют собой потенциальную проблему безопасности, так как такие файлы при запуске любым пользователем приобретают полномочия суперпользователя. Список технически возможных, но необычных ситуаций, о которых программа должна информировать, можно продолжать довольно долго.

До сих пор мы обсуждали проблему защиты пользователя от сбоев. Некоторые файловые системы также пытаются защитить пользователя от самого себя. Если пользователь собирается ввести команду

```
rm *.o
```

чтобы удалить все файлы с расширением .o (созданные компилятором объектные файлы), но случайно вместо этого набьет строку

```
rm * .o
```

(обратите внимание на пробел после звездочки), программа rm удалит все файлы в текущем каталоге, после чего сообщит, что не может найти файл .o. В системе MS-DOS и некоторых других системах при удалении файла устанавливается всего лишь один бит в каталоге или *i*-узле, отмечая, что файл удален. Блоки диска не возвращаются[1] в список свободных блоков до тех пор, пока они не понадобятся. Таким образом, если пользователь быстро обнаружит ошибку, он сможет восстановить удаленные файлы. В Windows удаленные файлы обычно помещаются в «мусорную корзину», откуда их можно при необходимости из-

[1] В MS-DOS первый символ имени удаленного файла заменяется символом 0xE5, а блоки-секторы, занимаемые файлом, освобождаются. Создаваемый после этого новый файл может занять эти блоки-секторы и запись в каталоге. Но сразу после удаления файл может быть восстановлен по сохранившемуся (кроме первого символа имени) описателю в каталоге. — *Примеч. перев.*

влечь. При этом свободное пространство на диске не увеличивается до тех пор, пока корзина не будет очищена.

5.3.5. Производительность файловой системы

Доступ к диску значительно медленнее, чем к оперативной памяти. Чтение слова из памяти может занять около 10 нс. Чтение с жесткого диска может выполняться со скоростью 10 Мбайт/с, что в сорок раз медленнее, но к этому следует добавить 5–10 мс на поиск нужного цилиндра и ожидание поворота диска. Если требуется прочитать или записать всего одно слово, то оперативная память оказывается примерно в миллион раз быстрее жесткого диска. Поэтому во многих файловых системах применяются различные методы оптимизации, увеличивающие производительность. В данном разделе мы рассмотрим три из них.

Для минимизации количества обращений к диску применяется *блочный кэш* или *буферный кэш*. (Термин «кэш» происходит от французского слова cacher, что значит «скрывать».) В данном контексте кэшем называется набор блоков, логически принадлежащих диску, но хранящихся в оперативной памяти по соображениям производительности.

Существуют различные алгоритмы кэширования. Обычная практика заключается в перехвате всех запросов чтения к диску и проверке наличия требующихся блоков в кэше. Если блок присутствует в кэше, то запрос чтения блока может быть удовлетворен без обращения к диску. В противном случае блок сначала считывается с диска в кэш, а оттуда копируется по нужному адресу памяти. Последующие обращения к тому же блоку могут удовлетворяться из кэша.

Когда требуется загрузить блок в заполненный до предела кэш, какой-либо другой блок должен быть из него удален (и записан на диск, если он был модифицирован в кэше). Эта ситуация очень похожа на страничную организацию памяти, и к ней применимы все обычные алгоритмы замены, описанные в главе 4, такие как FIFO (First in First Out — первым прибыл — первым обслужен), «вторая попытка» и LRU (Least Recently Used — с наиболее давним использованием). Одно приятное отличие кэширования от страничной организации памяти состоит в том, что обращения к кэшу производятся относительно нечасто, что позволяет хранить все блоки в точном LRU-порядке с однонаправленными списками.

К сожалению, здесь есть одна загвоздка. Теперь, когда мы можем реализовать точное соблюдение алгоритма LRU, оказывается, что алгоритм LRU является нежелательным. Вызвано это тем, что его буквальное применение снижает надежность файловой системы и угрожает ее непротиворечивости (обсуждавшейся в предыдущем разделе). Если в кэш считывается критический блок и далее модифицируется, например, блок *i*-узла, но не записывается сразу же на диск, то компьютерный сбой может привести к тому, что файловая система окажется в некорректном состоянии. Если блок *i*-узла поместить в конец цепочки LRU, может пройти довольно много времени, прежде чем этот блок попадет в ее начало и будет записан на диск.

Более того, к некоторым блокам, таким как блоки *i*-узлов, программы редко обращаются дважды в течение короткого интервала времени. Исходя из этих соображений, мы приходим к модифицированной схеме LRU, принимая во внимание два следующих фактора.

1. Насколько велика вероятность того, что данный блок скоро снова понадобится?
2. Важен ли данный блок для непротиворечивости файловой системы?

Для ответа на каждый из поставленных вопросов блоки можно разделить на такие категории, как блоки *i*-узлов, «косвенные» блоки (косвенной адресации), блоки каталогов, блоки, целиком заполненные данными, и блоки, частично заполненные данными. Блоки, которые, вероятно, не потребуются снова в ближайшее время, помещаются в начало списка LRU, чтобы занимаемые ими буферы могли вскоре освободиться. Блоки, вероятность повторного использования которых в ближайшее время высока (например, частично заполненные записываемые блоки), заносятся в конец списка LRU, что позволяет им оставаться в кэше более долгое время.

Второй вопрос не связан с первым. Если блок представляет важность для непротиворечивости файловой системы (обычно это все блоки, кроме блоков данных), и такой блок модифицируется, то его следует немедленно сохранить на диске, независимо от его положения в списке LRU. Своевременно записывая критические блоки, мы значительно снижаем вероятность того, что сбой компьютера повредит файловую систему. Пользователь вряд ли будет рад потере одного из своих файлов из-за какого-то сбоя. Еще сильнее он огорчится, если при этом испорченной окажется вся файловая система.

Даже при принятии всех перечисленных выше мер предосторожности по поддержанию в рабочем состоянии файловой системы слишком долгое хранение в кэше блоков с данными является нежелательным. Представьте себе автора будущей книги, подготавливаемой на персональном компьютере. Даже если наш писатель периодически велит текстовому редактору сохранять редактируемый файл на диске, есть большая вероятность, что все блоки останутся в кэше. Если произойдет сбой, структура файловой системы не пострадает, но труд целого дня будет потерян.

Эта ситуация случается не слишком часто, если только с очень невезучими пользователями. Для решения данной проблемы обычно применяется два метода. В системе UNIX есть вызов sync, принуждающий сохранить все модифицированные блоки кэша на диске. При загрузке операционной системы запускается фоновая задача, обычно под названием update, вся работа которой заключается в периодическом (обычно через каждые 30 с) обращении к системному вызову sync. В результате при любом сбое будет потеряно не более полминуты работы.

В системе MS-DOS практикуется другой подход, состоящий в том, что каждый модифицированный блок записывается на диск сразу же. Кэш, в котором все модифицированные блоки немедленно записываются на диск, называется *сквозным кэшем* или *кэшем со сквозной записью*. При использовании сквозного

кэша количество обращений ввода/вывода к диску больше, чем при применении обычного кэша. Чтобы лучше понять разницу в этих двух подходах, представьте себе программу, сохраняющую блок размером в 1 Кбайт по одному символу. Система UNIX будет собирать все символы в кэше и записывать этот блок на диск каждые 30 с, или когда блок будет удален из кэша. Система MS-DOS будет обращаться к диску для каждого символа. Конечно, в большинстве программ применяется внутренняя буферизация, поэтому обычно они обращаются к системному вызову write не с одним символом, а с целыми строками или большими единицами данных.

Результатом различия стратегий кэширования оказывается тот факт, что простое удаление (гибкого) диска из системы UNIX, без выполнения системного вызова sync, почти всегда приведет к потере данных и часто также к повреждению файловой системы. В MS-DOS такой проблемы не возникает. Такое различие в стратегиях связано с тем, что UNIX разрабатывалась в среде, в которой все диски были жесткими и постоянными, тогда как система MS-DOS изначально предназначалась для работы со сменными носителями. Когда жесткие диски стали нормой, более эффективный метод, присущий UNIX, также стал нормой и теперь используется в Windows для жестких дисков.

Кэширование является единственным способом увеличения производительности системы. Другой важный метод состоит в уменьшении затрат времени на перемещение блока головок. Достигается это помещением блоков, к которым высока вероятность доступа в течение короткого интервала времени, близко друг к другу, желательно на одном цилиндре. Когда записывается выходной файл, файловая система должна зарезервировать место для чтения таких блоков за одну операцию. Если свободные блоки учитываются в битовом массиве, а весь битовый массив помещается в оперативной памяти, то довольно легко выбрать свободный блок как можно ближе к предыдущему блоку. В случае когда свободные блоки хранятся в списке, часть которого находится в оперативной памяти, а часть на диске, сделать это значительно труднее.

Однако даже при использовании списка свободных блоков может быть выполнена определенная кластеризация данных. Хитрость заключается в том, чтобы учитывать место на диске не в блоках, а в группах последовательных блоков. Если сектор состоит из 512 байт, система может использовать блоки размером в 1 Кбайт (два сектора), но выделять пространство на диске в единицах по два блока (четыре сектора). Это не то же самое, что использование двухкилобайтовых дисковых блоков, так как кэш все также рассчитан на килобайтовые блоки, и дисковые операции чтения и записи будут по-прежнему работать с килобайтовыми блоками. Однако при последовательном чтении файла количество операций поиска цилиндра уменьшится вдвое, что значительно увеличит производительность. Вариация этой же темы сводится к попытке системы учесть позицию блока в цилиндре.

Производительность файловых систем снижается еще в силу того, что при оперировании *i*-узлами или чем-либо эквивалентным им, особенно при чтении коротких файлов, требуется два обращения к диску вместо одного: одно для *i*-уз-

ла и одно для блока данных. Обычное размещение *i*-узлов на диске показано на рис. 5.17, *а*. Здесь все *i*-узлы располагаются в начале диска, откуда среднее расстояние между *i*-узлом и его блоками будет составлять около половины количества цилиндров, то есть при доступе практически к каждому файлу потребуются значительные перемещения блока головок.

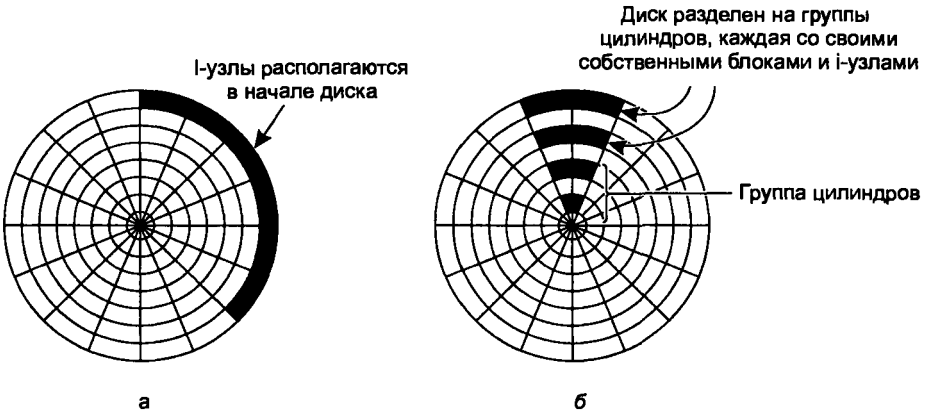


Рис. 5.17. *а* — i-узлы, размещенные в начале диска; *б* — диск, разделенный на группы цилиндров, каждая со своими собственными блоками и i-узлами

Один из способов подъема производительности заключается в помещении *i*-узлов в середину диска. Таким образом уменьшается средний путь блоков головок в два раза. Другая идея, отраженная на рис. 5.17, *б*, заключается в разбиении диска на группы цилиндров, каждая со своими *i*-узлами, блоками и списком свободных блоков. Когда создается новый файл, может быть выбран любой *i*-узел, но предпринимается попытка найти блок в той же группе цилиндров, что и *i*-узел. Если эта попытка заканчивается неудачей, используется блок в соседней группе цилиндров.

5.3.6. Файловые системы с журнальной структурой, LFS

Изменения в технологии оказывают влияние на современные файловые системы. В частности, центральные процессоры становятся все быстрее, емкость дисков увеличивается, цена их снижается (но скорость увеличивается не столь значительно), а размеры оперативной памяти растут экспоненциально. Единственным параметром, не меняющимся столь стремительно, является время поиска цилиндра диска. В результате во многих файловых системах появляется узкое место. В университете Беркли были проведены исследования, направленные на снижение остроты этой проблемы при помощи создания совершенно новой файловой системы LFS (Log-structured File System — файловая система с журналь-

ной структурой). В этом разделе мы кратко опишем, как работает система LFS. Дополнительные сведения можно узнать в [65].

В основе системы LFS лежит идея того, что по мере ускорения центральных процессоров и экстенсивного расширения оперативной памяти выгода от кэширования дисков все увеличивается. Поэтому становится возможным удовлетворить весьма существенную часть всех дисковых запросов прямо из кэша файловой системы без обращения к диску. Из этого следует, что в будущем большинство обращений к диску будут составлять обращения записи, поэтому алгоритм опережающего чтения, применявшийся в некоторых файловых системах, уже не дает большого выигрыша производительности.

Ситуация усложняется тем, что в большинстве файловых систем операции записи выполняются над очень маленькими блоками данных. Такие операции записи оказываются крайне неэффективными, поскольку самой физической записи, занимающей 50 мкс, часто предшествует поиск цилиндра в течение 10 мс и 6-миллисекундная задержка вращения. При таких параметрах эффективность диска падает до 1 %.

Чтобы понять, из чего складываются все эти мелкие операции записи, рассмотрим создание файла в операционной системе UNIX. Здесь необходимо произвести запись в *i*-узел каталога, блок каталога, *i*-узел файла и, наконец, блок самого файла. В принципе, эти операции записи могут быть отложены на некоторое время, но тем самым не исключаются серьезные проблемы с целостностью файловой системы в случае сбоя компьютера прежде, чем запись на диск будет выполнена. По этой причине *i*-узлы обычно сохранятся на диск без промедления.

Учитывая все вышесказанное, разработчики файловой системы LFS решили реализовать файловую систему UNIX таким образом, чтобы достичь максимума эффективности использования диска, несмотря на рабочую нагрузку, состоящую из большого количества случайных мелких операций записи. Замысел состоит в использовании диска как журнала. Периодически, когда возникает необходимость, все буферизированные в памяти блоки, подлежащие записи, собираются вместе в единый сегмент, и он сбрасывается на диск одним непрерывным куском в конец журнала. Записываемый сегмент может содержать *i*-узлы, блоки каталогов и блоки данных, перемешанные друг с другом. В начале каждого сегмента создается его оглавление. Если довести средний размер сегмента до 1 Мбайт, то можно задействовать почти всю пропускную способность диска.

При такой организации *i*-узлы существуют и имеют ту же структуру, что и в UNIX, но теперь они, вместо того чтобы располагаться в фиксированной позиции на диске, рассредоточены по всему журналу. Тем не менее, когда программа находит *i*-узел, определение расположения блоков выполняется обычным способом. Конечно, здесь обнаружить *i*-узел намного сложнее, так как его адрес не определяется по его номеру, как это было в UNIX. Чтобы можно было найти *i*-узел, создается массив *i*-узлов, индексированный по *i*-номерам. Элемент *i* массива указывает на *i*-узел *i* на диске. Массив хранится на диске, но также содержится и в кэше, а это означает, что наиболее часто востребуемые части этого массива постоянно находятся в оперативной памяти.

Таким образом, все операции записи буферизируются в памяти и периодически данные из буфера записываются на диск в виде единых сегментов в конец журнала. Чтобы открыть файл, используется массив, позволяющий обнаружить *i*-узел в файле. Как только *i*-узел обнаружен, могут быть определены номера блоков файла. Все эти блоки также располагаются в сегментах где-то в журнале.

Если бы диски были бесконечного размера, на этом все бы и заканчивалось. Однако существующие диски имеют ограниченный размер, поэтому рано или поздно журнал распухнет на весь диск. К счастью, многие сегменты могут содержать уже ненужные блоки. Например, если файл был перезаписан, его *i*-узел будет указывать на новые блоки, но старые блоки будут все также занимать место в записанных ранее сегментах.

Для решения проблемы повторного использования блоков в старых сегментах в файловую систему LFS встроена нить *чистильщика*, в обязанности которого входит постоянное сканирование журнала с целью сделать последний более компактным. Чистильщик начинает с того, что считывает содержимое самого первого сегмента журнала с целью определить, какие *i*-узлы и файлы в нем находятся. Затем он смотрит в текущий массив *i*-узлов, проверяя, являются ли *i*-узлы все еще текущими, и используются ли все еще блоки файлов. Если нет, эта информация отбрасывается, а используемые *i*-узлы и блоки считываются в память, чтобы записать их в следующий сегмент. Исходный сегмент помечается как свободный, поэтому журнал может помещать в него новые данные. Таким образом чистильщик двигается по журналу, удаляя старые сегменты с диска и помещая всю имеющую ценность информацию в память для перезаписи в следующий сегмент. В результате диск представляет собой большой кольцевой буфер, в котором пишущий поток добавляет новые сегменты с одного конца, а чистящий процесс удаляет старые сегменты с другого.

Учет расположения блоков здесь весьма нетривиален, поскольку, когда блок файла записывается в новый сегмент, *i*-узел файла (где-то в журнале) должен быть найден, обновлен и помещен в буфер для записи в следующий сегмент. При этом массив *i*-узлов также должен быть обновлен, чтобы элемент массива указывал на новую копию. Тем не менее администрирование такой системы вполне возможно, а увеличение производительности показывает, что все эти сложности были не напрасны. Результаты измерений показали, что файловая система с журнальной структурой (LFS) превосходит файловую систему UNIX при множестве небольших записей на порядок, а при чтении и больших записях обладает сходной или лучшей производительностью.

5.4. Безопасность

Многие компании располагают ценной информацией, которую они тщательно охраняют. Таким образом, защита информации от несанкционированного доступа является главной заботой всех операционных систем. В следующих разделах мы рассмотрим различные вопросы, связанные с безопасностью и защитой, рав-

но относящиеся как к системам с разделением времени, так и к сетевым и персональным компьютерам, работающим в сети.

5.4.1. Понятие безопасности

Термины «безопасность» и «защита» иногда смешиваются. Тем не менее часто бывает полезно провести границу между общими проблемами, связанными с гарантией того, что файлы не читаются и не модифицируются неавторизованными лицами, с одной стороны, и специфическими механизмами операционной системы, привлекаемыми для обеспечения безопасности, с другой стороны. Чтобы избежать путаницы, мы будем применять термин *безопасность* для обозначения общей проблемы и термин *механизмы защиты* при описании специфических механизмов операционной системы, используемых для обеспечения информационной безопасности в компьютерных системах. Однако граница между этими двумя терминами определена не четко. Сначала мы познакомимся с вопросами безопасности, чтобы понять природу проблемы. Затем мы рассмотрим механизмы защиты и модели, способствующие обеспечению безопасности.

Проблема безопасности многогранна. Тремя ее наиболее важными аспектами являются природа угроз, природа злоумышленников и случайная потеря данных. Все эти вопросы не будут обойдены вниманием в этой главе. К наиболее распространенным причинам случайной потери данных относятся:

1. «Форс-мажор»: пожары, наводнения, землетрясения, войны, восстания, животные, питающиеся магнитными лентами или гибкими дисками.
2. Аппаратные и программные ошибки: сбои центрального процессора, нечитаемые диски или ленты, ошибки при передаче данных, ошибки в программах.
3. Человеческий фактор: неправильный ввод данных, неверные вставленный диск или заправленная лента, запуск не той программы, потерянные диск или лента и т. д.

Большая часть этих проблем может быть разрешена при помощи своевременного создания соответствующих резервных копий, хранимых на всякий случай вдали от оригинальных данных.

Проблема злоумышленников представляется более интересной. В литературе по безопасности *злоумышленником* и иногда *неприятелем* называют человека, сующего свой нос в чужие дела. Злоумышленники подразделяются на два вида. Пассивные злоумышленники просто пытаются прочитать файлы, которые им не разрешено читать. Активные злоумышленники пытаются незаконно изменить данные. При разработке защиты системы от злоумышленников важно знать врага, против которого нужно возводить укрепления, в лицо. Наиболее распространенными категориями злоумышленников являются:

1. Случайные любопытные пользователи, не применяющие специальных технических средств. У многих людей есть компьютеры, соединенные с общим файловым сервером. И если не установить специальной защиты, благодаря естественному любопытству многие люди станут читать чужую

электронную почту и другие файлы. Например, во многих системах UNIX новые только что созданные файлы по умолчанию доступны для чтения всем желающим.

2. Члены организации, занимающиеся шпионажем. Студенты, системные программисты, операторы и другой технический персонал часто считают взлом системы безопасности локальной компьютерной системы святым долгом. Как правило, они имеют высокую квалификацию и готовы посвящать достижению поставленной перед собой цели значительное количество времени.
3. Те, кто совершают решительные попытки личного обогащения. Некоторые программисты, работающие в банках, предпринимали попытки украсть деньги у банка, в котором они работали. Их схемы варьировались от изменения способов округления сумм в программах, для сбора, таким образом, с миру по нитке, до шантажа («Заплатите мне, или я уничтожу всю банковскую информацию»).
4. Лица, занимающиеся коммерческим и военным шпионажем. Шпионаж представляет собой серьезную и хорошо финансированную попытку конкурента или другой страны украсть программы, коммерческие тайны, ценные идеи и технологии, схемы микросхем, бизнес-планы и т. д. Часто такие попытки включают подключение к линиям связи или установку антенн, направленных на компьютер для улавливания его электромагнитного излучения.

Очевидно, что попытка предотвратить кражу военных секретов враждебным иностранным государством отличается от противостояния шалостям студентов, встраивающих забавные сообщения в систему. Необходимые для поддержания секретности и защиты усилия зависят от предполагаемого противника.

Еще один аспект проблемы безопасности касается права пользователя на конфиденциальность личной информации. Это довольно запутанная тема, связанная с различными юридическими и моральными вопросами. Должно ли и имеет ли право правительство собирать досье на всех и каждого, чтобы поймать лиц, уклоняющихся от налогов или получающих незаконные пособия? Должна ли полиция иметь возможность слежки за всем и вся, пытаясь победить организованную преступность? Есть ли права доступа к частной информации у работодателей и страховых компаний? Что происходит, когда эти права вступают в конфликт с индивидуальными правами граждан? Все эти вопросы имеют чрезвычайно большое значение, но они находятся за пределами рассмотрения данной книги.

5.4.2. Знаменитые дефекты систем безопасности

Подобно знаменитым «Титанику» и «Гинденбургу» в транспортной индустрии, в области компьютерной безопасности также были моменты, о которых эксперты предпочли бы не вспоминать. У утилиты lpr системы UNIX, печатающей файл на

принтере, есть входной параметр, позволяющий удалять печатаемый файл после вывода на принтер. В ранних версиях системы UNIX любой пользователь мог распечатать и удалить с помощью этой утилиты файл паролей.

Другой способ взлома системы UNIX заключался в установке связи с файлом паролей файла с именем core, находящимся в рабочем каталоге пользователя. Затем взломщик намеренно вызывал сбой с сохранением образа памяти SETUID-программы, который система записывала в файл core, то есть поверх файла паролей. Таким образом, пользователь мог заменить содержимое файла паролей, указав в новом файле несколько строк по собственному усмотрению (задавая их в качестве аргументов команды).

Еще один механизм обхода защиты системы UNIX включал использование команды

```
mkdir foo
```

Утилита mkdir была SETUID-программой, владельцем которой являлась система (root). Эта программа создавала *i*-узел для каталога foo при помощи системного вызова mknod, после чего изменяла владельца каталога foo со своего идентификатора UID (то есть root) на UID пользователя. Когда системы были медленными, пользователю иногда удавалось успеть быстро удалить *i*-узел каталога и создать связь с файлом паролей под именем foo после системного вызова mknod, но до вызова chown. Если утилита mkdir выполняла системный вызов chown, она делала пользователя владельцем файла паролей. Поместив необходимые команды в файл сценария оболочки, взломщик мог пытаться выполнить эту операцию снова и снова, пока трюк не срабатывал.

Операционная система TENEX была очень популярна на машинах DEC-10. Теперь она уже не используется, но эта система навечно останется в анналах по компьютерной безопасности благодаря следующей ошибке разработчиков. Система TENEX поддерживала страничную организацию памяти. Чтобы пользователи могли отслеживать поведение собственных программ, можно было запрограммировать систему на вызов функции пользователя при каждом страничном прерывании.

Для защиты файлов в системе TENEX также использовались пароли. Для получения доступа к файлу программа должна была указать операционной системе правильный пароль во время открытия файла. Операционная система проверяла пароль символ за символом, останавливаясь, как только видела, что пароль неверен. Чтобы взломать защиту системы TENEX, злоумышленник мог аккуратно расположить пароль в памяти так, как указано на рис. 5.18, *а*, поместив первый символ пароля в конце одной страницы, а остальные символы — в начале следующей страницы.

Затем необходимо гарантировать отсутствие в памяти второй страницы, для чего можно, например, много раз обратиться к другим страницам. После этого программа пыталась открыть файл жертвы с помощью тщательно подбираемого пароля. Если первый символ пароля угадан пользователем неверно, система остановит проверку на первом же символе, ответив соответствующим сообщением, и страничного прерывания не произойдет. Если же первый символ верен, опера-

ционная система перейдет к проверке следующего символа, вызывая страничное прерывание, отслеживаемое программой пользователя.

Таким образом, программа может перебирать символ за символом, отгадывая пароль буква за буквой, и сдвигая пароль при каждом отгаданном символе на один символ вниз относительно границы страниц (рис. 5.18, *в*). Для нахождения всего пароля требуется перебор менее 128 символов (набор ASCII) для каждой позиции в пароле. Следовательно, для нахождения пароля длиной в n символов требуется всего $128\,n$ попыток вместо 128^n.

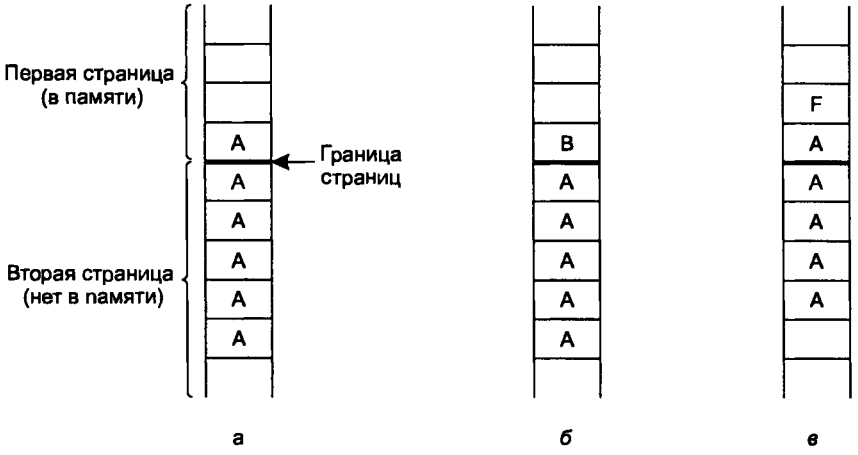


Рис. 5.18. Взлом пароля в системе TENEX: *а* — начальное состояние; *б* — в момент перебора символов; *а* — сдвиг пароля, если есть такая буква в этом слове

Теперь остановимся на дефектах безопасности в системе OS/360. Описание слегка упрощено, но в нем сохранена суть проблемы. В данной системе можно было запустить чтение магнитной ленты, а затем продолжить вычисления во время считывания данных с ленты в пространство пользователя. Трюк заключался в том, чтобы начать чтение ленты, а затем обратиться к системному вызову, которому требовалась структура данных пользователя, например файл и его пароль.

Операционная система сначала проверяла верность пароля для данного файла. После этого она считывала имя того же файла снова уже для его чтения. (Это имя могло бы храниться в системе, но теория теорией...) К сожалению, как раз перед тем, как операционная система собиралась считать имя файла во второй раз, поверх этого имени считывалось новое имя с магнитной ленты. Таким образом, операционная система, убедившись в подлинности пароля для одного файла, считывала другой файл, для которого у пользователя пароля не было. Чтобы точно рассчитать время обращений к системным вызовам, требовалась определенная практика, но это было не так уж сложно.

Время текло, и в дополнение к этим примерам появились новые уязвимости и способы их атак. Одним из вариантов атаки изнутри является *троянский конь*,

представляющий собой невинную с виду программу, содержащую процедуру, выполняющую неожиданные и нежелательные функции. Этими функциями могут быть копирование файлов пользователя туда, где их впоследствии может получить взломщик, или даже отсылка их взломщику или во временное укромное место по электронной почте или с помощью протокола FTP.

Существует еще одна разновидность атаки системы изнутри, называемая *логической бомбой*. Программа типа логической бомбы тайно закладывается в операционную систему обычно одним из сотрудников компании. До тех пор пока программист каждый день входит в систему под своим именем и паролем, эта программа не предпринимает никаких действий. Однако если программиста внезапно увольняют и физически удаляют из помещения без предупреждения, то на следующий день (или на следующую неделю) логическая бомба, не получив своего ежедневного пароля, начинает действовать. Существует множество вариаций на эту тему. В одном знаменитом случае программа проверяла платежную ведомость. Если личный номер программиста не появлялся в двух последовательных ведомостях подряд, бомба «взрывалась».

Взрыв логической бомбы может заключаться в форматировании жесткого диска, удалении файлов в случайном порядке, внесении с трудом обнаруживаемых изменений в ключевые программы или в шифровании важных файлов. В последнем случае компания оказывается перед сложным выбором: вызвать полицию (в результате чего много месяцев спустя злоумышленника, возможно, арестуют и признают виновным, но файлы придется создавать заново) или сдаться шантажисту и снова нанять на работу этого программиста в качестве «консультанта» с астрономическим окладом для восстановления системы (и надеяться, что он при этом не заминирует систему еще раз).

Наверное, самым масштабным случаем несанкционированного доступа стали события вечера 2 ноября 1988 года, когда аспирант университета Корнелла в штате Нью-Йорк Роберт Таппан Моррис выпустил написанную им программу-червя в Интернет. В результате этого действия были заражены тысячи компьютеров в университетах, корпорациях и правительственных лабораториях по всему миру, прежде чем эту программу удалось выследить и удалить.

Технически червь состоял из двух программ: начального загрузчика и собственно червя. Начальный загрузчик представлял собой 99 строк на языке C (файл назывался l1.c). Этот загрузчик компилировался и исполнялся на атакуемом компьютере. Будучи запущенной, личинка связывалась с машиной, с которой поступила, загружала тело основного червя и запускала его. После некоторых действий, направленных на попытки скрыть свое существование, червь заглядывал в таблицы маршрутизации нового хоста, определяя компьютеры, с которыми тот был соединен, после чего пытался распространить начальный загрузчик на эти машины.

Попав в систему, червь пытался взломать систему паролей. Для этого Моррису не понадобилось предпринимать собственных исследований. Все, что ему потребовалось, — это попросить у собственного отца, эксперта в области безопасности в Управлении национальной безопасности США, профессионально занимающегося взломом кодов, перепечатку классической статьи, написанной десятилетием

раньше Моррисом старшим и Кеном Томпсоном в лаборатории Bell Labs [61]. Каждый взломанный пароль позволял червю зарегистрироваться на любой машине, на которой у владельца пароля были учетные записи.

Морриса поймали, когда один из его друзей разговаривал с репортером из компьютерной редакции *Нью-Йорк Таймс*, Джоном Марковым, и пытался убедить репортера, что все это лишь несчастный случай, что червь безобиден и автор весьма сожалеет. Друг по неосторожности упомянул, что регистрационное имя нарушителя rtm. Преобразовать rtm в физическое имя владельца было несложно — все, что Маркову надо было сделать, — это запустить процедуру finger. На следующий день эта история оказалась на первых полосах всех газет, вытеснив оттуда даже информацию о предстоящих через три дня президентских выборах.

Моррис предстал перед федеральным судом и был приговорен к штрафу в размере 10 000 долларов, 3 годам испытательного срока и 400 часам общественных работ. Его судебные издержки, вероятно, превысили 150 000 долларов. Приговор породил множество разногласий. В компьютерном обществе многие считали, что Моррис был блестящим студентом, чья опасная шалость вышла из-под контроля. Написанная им программа не содержала ничего, что бы свидетельствовало о намерениях Морриса украсть какие-либо данные или причинить какой-либо намеренный ущерб. Другие считали его серьезным преступником, которому место в тюрьме.

Одним из результатов этого инцидента было создание группы компьютерной «скорой помощи» *CERT* (Computer Emergency Response Team), основными задачами которой являются доклады о попытках взлома в Интернете, а также анализ проблем безопасности и разработка методов их решения. При необходимости эта группа рассылает свою информацию тысячам системных администраторов по Интернету. К сожалению, этой информацией, содержащей сообщения об ошибках в системах, могут воспользоваться также и злоумышленники (возможно, притворяющиеся системными администраторами).

5.4.3. Атака системы безопасности

Обычный способ проверить надежность системы безопасности заключается в приглашении группы экспертов, называемых *командой «тигров»*, или *группой проникновения*, чтобы посмотреть, смогут ли они взломать защиту. Иногда в качестве такой команды приглашались аспиранты [42]. За несколько лет подобные команды обнаружили множество областей, в которых операционные системы проявляют свою слабость. Ниже будут перечислены наиболее распространенные методы атак, часто завершающиеся успехом. Хотя изначально эти методы разрабатывались для атаки систем разделения времени, они часто могут применяться и для нападения на серверы локальных сетей и другие совместно используемые машины. При разработке таких систем убедитесь, что им под силу выдержать атаки следующих типов:

- запросите страницы памяти, место на диске или магнитной ленте и просто считайте. Многие системы не очищают память при ее выделении пользо-

вателю, поэтому память и диски могут содержать много интересной информации, записанной предыдущим владельцем;

- попытайтесь обратиться к несуществующим системным вызовам или к имеющимся системным вызовам, но с неверными параметрами, например не того типа или слишком большой длины. Многие системы не выдерживают подобного обращения с ними;
- начните регистрацию, а затем нажмите клавишу DEL, RUBOUT или BREAK посреди процесса регистрации. В некоторых системах подобным образом удается уничтожить процесс, осуществляющий проверку пароля, и регистрация считается успешной;
- попытайтесь модифицировать сложные структуры операционной системы, хранящиеся в области памяти пользователя (если таковые имеются). В некоторых системах (особенно на мэйнфреймах), для того чтобы открыть файл, программа формирует большую структуру, содержащую имя файла и множество других параметров, которую передает операционной системе. При чтении и записи файла операционная система иногда сама обновляет эту структуру. Модификация некоторых полей может иметь разрушительное воздействие на безопасность;
- прочитайте руководство и попытайтесь найти фразы, гласящие: «Не делайте X». Попытайтесь проделать «X» в различных комбинациях;
- убедите системного администратора добавить потайную дверь с обходом важных этапов проверки безопасности для любого пользователя с вашим именем;
- если ничего другого не получилось, попытайтесь найти секретаршу системного администратора и притвориться несчастным пользователем, забывшим свой пароль. В качестве альтернативы можно попытаться подкупить секретаршу. У секретарши, как правило, есть доступ к самой разнообразной и очень интересной информации, а зарплата обычно невелика. Не следует недооценивать человеческий фактор.

Подобные и другие методы атак обсуждаются в [56]. Хотя эта статья вышла уже более четверти века тому назад, многие руководства к действию, вошедшие в нее, все еще работают.

Вирусы

Большие проблемы у многих пользователей вызвали компьютерные вирусы, представляющие собой особый тип атаки системы безопасности. *Вирус* является фрагментом программного кода, который присоединяется к нормальным программам, чтобы заражать другие программы. От червя вирус отличается только тем, что он паразитирует на существующих программах, в то время как червь — вещь в себе. И черви, и вирусы самостоятельно размножаются и могут причинить серьезные повреждения.

Типичный вирус работает следующим образом. Автор вируса сначала делает какую-либо полезную программу, например игру для MS-DOS. В коде этой про-

граммы прячется код вируса. Затем игра либо помещается в свободный доступ, либо продается за умеренную цену. Создать вирус не так просто, их авторы зачастую весьма изобретательны. Поэтому игра скоро становится достаточно популярной.

При запуске программа немедленно начинает искать на жестком диске все исполняемые файлы, проверяя, заражены ли они. Когда обнаруживается неинфицированная программа, она заражается, для этого к ней пристыковывается код вируса, а первая команда заменяется на переход на этот код. Код вируса, завершив работу, сначала исполняет инструкцию, которая ранее была первой, а затем переходит на вторую инструкцию исходной программы. Таким образом, при каждом запуске инфицированной программы она пытается заражать другие программы.

Помимо того, чтобы заражать программы, вирус может делать и другие вещи, например удалять, модифицировать или шифровать файлы. Один вирус даже вымогал деньги, выводя на экран сообщение с предложением переслать 500 долларов на абонентский ящик в Панаме, в противном случае угрожая уничтожить данные и повредить оборудование.

Вирус может также изменить загрузочный сектор диска, последствия чего для компьютера разрушительны. Такой вирус может просить ввести пароль, который за некоторую сумму в мелких непомеченных банкнотах может предложить его автор.

Проблем, вызываемых вирусами, легче избежать, чем пытаться лечить зараженную систему. Безопаснее всего, конечно, покупать программное обеспечение только в заслуживающих доверия магазинах. Использование бесплатных программ, загруженных из сети, или взятых у друзей пиратских копий приводит к риску заражения. Существуют коммерческие антивирусные пакеты, но некоторые из них просматривают содержимое только на указанные известные вирусы.

Более радикальный способ лечения сводится к тому, чтобы полностью форматировать диск, включая и загрузочный сектор. Затем нужно установить заслуживающие доверия программы и для каждого файла подсчитать контрольную сумму. Алгоритм вычисления контрольной суммы не важен, главное, чтобы сумма имела достаточное количество битов (как минимум, 32). Контрольные суммы нужно сохранить в безопасном месте, либо на дискету, либо в зашифрованном виде. После этого при каждой загрузке системы нужно заново вычислять контрольные суммы файлов, сверяя их с контрольными значениями. Если какой-либо файл меняется, он, вероятно, заражен. Такая методика не позволяет предотвратить заражение, но она хотя бы позволяет на раннем этапе его обнаружить.

Можно помешать заражению, если сделать каталог, содержащий исполняемые файлы, недоступным на запись для обычных пользователей. Вирусу, благодаря этому, будет сложнее заразить другие исполняемые файлы. Такой подход годится для UNIX, хотя в MS-DOS не работает, так как в ней невозможно полностью запретить запись в каталог.

5.4.4. Принципы проектирования систем безопасности

Теперь читателю должно быть ясно, что разработка хорошо защищенной операционной системы представляет собой нетривиальную задачу. Над этой проблемой без особого успеха билось множество людей. В 1975 году исследователи определили несколько общих принципов, которых необходимо придерживаться при проектировании надежных систем [67]. Ниже приведен краткий обзор некоторых из этих идей (основанных на опыте работы в системе MULTICS), значимость которых за последние четверть века не изменилась.

Во-первых, устройство системы не должно быть секретом. Предположение, что взломщики не знают, как работает система, может только ввести разработчиков в заблуждение. Рано или поздно злоумышленники узнают нужную им информацию, и если защита системы окажется скомпрометированной этой утечкой информации, это значит, что такая система безопасности ни на что не годится.

Во-вторых, по умолчанию доступ не должен предоставляться. Об ошибках, в результате которых пользователям было отказано в законном доступе, сообщат значительно быстрее, чем о случаях ошибочного предоставления несанкционированного доступа. Когда есть сомнение, говорите «Нет».

В-третьих, необходимо проверять текущее состояние прав доступа. Система не должна, проверив наличие прав доступа и убедившись, что доступ разрешен, затем сохранять эту информацию для последующего использования. Многие системы проверяют разрешение доступа при открытии файла, но не после его. Это означает, что пользователь, открывший файл и держащий его открытым неделями, будет продолжать обладать доступом к файлу, даже если владелец файла с тех пор уже давно изменил защиту файла или, может, даже пытался удалить этот файл.

В-четвертых, предоставляйте каждому процессу как можно меньше привилегий. Если у программы-редактора есть доступ только к редактируемому файлу (указанный при вызове), то редакторы в виде троянских коней с ахейцами в брюхе не смогут причинить большого вреда. Применение этого принципа предполагает схему защиты высокой степени детализации. Подобные схемы будут рассматриваться позднее в данной главе.

В-пятых, механизм защиты должен быть простым, одинаковым для всех и встроенным в самые нижние уровни системы. Попытка установить систему безопасности на существующую незащищенную систему практически невыполнима. Безопасность, как и целостность, не является свойством, которое можно добавить потом.

В-шестых, выбранная схема должна быть психологически приемлемой. Если пользователи почувствуют, что для защиты файлов требуется затратить слишком много усилий, они не станут этого делать. Тем не менее они будут громко жаловаться, если что-либо пойдет не так. Ответы типа «это ваша вина», как правило, не будут восприниматься с пониманием.

5.4.5. Аутентификация пользователей

Когда пользователь регистрируется на компьютере, операционная система, как правило, желает определить, кем является данный пользователь, и запускает процесс, называемый *аутентификацией пользователя*. Большинство методов аутентификации основаны на распознавании чего-то, известного пользователю, чего-то, имеющегося у пользователя, или чего-то, чем является пользователь.

Парольная аутентификация

В наиболее широко применяемой форме аутентификации пользователю предлагается ввести имя и пароль. Защита пароля легко реализуется. Самый простой способ реализации паролей заключается в поддержании централизованного списка пар (имя регистрации, пароль). Вводимое имя отыскивается в списке, а указанный пользователем пароль сравнивается с хранящимся в списке. Если пароли совпадают, регистрация в системе разрешается, если нет — в регистрации пользователю отказывается.

Большинство взломщиков проникают в систему, просто перебирая множество комбинаций имени и пароля, пока не находят комбинацию, которая работает. Многие пользователи используют в качестве регистрационного имени свое собственное имя в той или иной форме. Например, для пользователя по имени Ellen Ann Smith разумными кандидатами регистрационного имени являются ellen, smith, ellen_smith, ellen-smith, ellen.smith, esmith, easmith и eas. Вооружившись одной из книг типа «4096 имен для вашего новорожденного», плюс телефонной книгой, полной фамилий, взломщик без труда составит компьютеризированный список потенциальных регистрационных имен, соответствующих стране, в которой он собирается произвести атаку (имя ellen_smith может быть полезным в США или Великобритании, но вряд ли поможет в Японии).

Конечно, угадать регистрационное имя — это еще не все. Также требуется подобрать пароль. Насколько это сложно? Проще, чем вы думаете. Классический труд по вопросу безопасности паролей был написан Моррисом и Томпсоном в 1979 году на основе исследований систем UNIX [61]. Авторы данной книги скомпилировали список вероятных паролей: имя и фамилия, названия улиц, городов, слова из словарей среднего размера (также написанные задом наперед), автомобильные номера и короткие строки случайных символов. Затем они сравнили свой полученный таким образом список с системным файлом паролей, чтобы посмотреть, есть ли совпадения. Как выяснилось, более 86 % от общего количества паролей в файле оказались в их списке.

Если бы все пароли состояли из 7 символов, случайным образом выбранных из 95 печатных символов набора ASCII, их количество было бы равно 95^7, что примерно равно 7×10^{13}. Если выполнять 2000 шифрований в секунду, на построение списка для сверки пароля потребуется около 2000 лет. Более того, под хранение такого списка ушло бы 20 млн магнитных лент. Даже требование того, чтобы пароли содержали как минимум один символ нижнего регистра, один символ верхнего регистра, один специальный символ и были бы как минимум

семь или восемь символов в длину, заметно улучшает ситуацию по сравнению с тем случаем, когда пароль выбирается пользователем без всяких ограничений.

Даже для случая, когда по политическим причинам невозможно заставить пользователей выбирать разумные пароли, Моррис и Томпсон описали технологию, делающую их собственный метод атаки (заранее зашифровать большое число паролей) практически бесполезным. Идея состоит в том, чтобы ассоциировать с каждым паролем n-битное случайное число. Это случайное число меняется при каждом изменении пароля. Оно хранится в файле паролей в незашифрованном виде, и каждый может его видеть. Пароль же сначала объединяется со случайным числом, а только затем шифруется и записывается в файл.

Рассмотрим, к каким последствиям приведет эта техника для злоумышленника, пытающегося составить список вероятных паролей, зашифровать их и сохранить в отсортированном виде в файл f, где их затем легко будет найти. Если злоумышленник считает, что слово Marilyn может быть паролем, ему теперь недостаточно закодировать только это слово и записать его в f. Он должен зашифровать и записать $2n$ строк, Marilyn0000, Marilyn0001, Marilyn0002 и т. д. Благодаря этой технике, называемой добавлением *«соли»* в файл паролей, размер файла f увеличивается в $2n$ раз. В UNIX применяется значение n, равное 12. В некоторых версиях UNIX сам файл паролей сделан недоступным для чтения, а для работы с ним предоставлена программа, которая просматривает запрошенные записи, внося дополнительную задержку, достаточную, чтобы сильно замедлить атаку.

Добавление случайных чисел к файлу паролей защищает систему от взломщиков, пытающихся заранее составить большой список зашифрованных паролей и таким образом взломать несколько паролей сразу. Однако данный метод бессилен помочь в том случае, когда пароль легко отгадать, например если пользователь David использует пароль David. Взломщик может просто попытаться отгадать пароли один за другим. Обучение пользователей в данной области помогает, но оно редко проводится. Но, помимо обучения пользователей, в вашем распоряжении помощь компьютера. На некоторых системах устанавливается программа, формирующая случайные, легко произносимые бессмысленные слова, такие как fotally, garbungy или bipitty, подходящие в качестве паролей (желательно с чередованием прописных и строчных букв и с разбавлением специальными символами). Программа, вызываемая пользователем для установки или смены пароля, может также выдать предупреждение при выборе «слабого» сочетания символов.

Снисходительная программа может просто брюзжать, строгая программа может отвергать пароль и требовать ввода лучшего варианта. Программа установки пароля может также предложить свой вариант, как уже обсуждалось выше.

Некоторые операционные системы требуют от пользователей регулярной смены паролей, чтобы ограничить ущерб от утечки пароля. Крайность здесь — *одноразовые пароли*. В этом случае пользователь получает блокнот, содержащий список паролей. Для каждого входа в систему используется следующий пароль в списке. В итоге, если взломщику и удастся узнать уже отработавший пароль, он

ему не пригодится. Предполагается, что пользователь постарается не терять выданный ему блокнот.

Еще один вариант идеи паролей заключается в том, что для каждого нового пользователя создается длинный список вопросов и ответов, который хранится на сервере в надежном виде (например, в зашифрованном). Вопросы должны выбираться так, чтобы пользователю не нужно было их записывать. Примеры:

1. Как зовут сестру Марджолин?
2. На какой улице расположена ваша начальная школа?
3. Что преподавала мисс Воробьофф?

При регистрации сервер задает один из этих вопросов, выбирая его из списка случайным образом, и проверяет ответ. Однако чтобы такая схема могла работать, потребуется большое количество пар вопросов и ответов.

Другой вариант называется *«оклик-отзыв»*. Он работает следующим образом. Пользователь выбирает алгоритм, идентифицирующий его как пользователя, например x^2. Когда пользователь входит в систему, сервер посылает ему некое случайное число, например 7. В ответ пользователь отправляет серверу число 49. Алгоритм может отличаться утром и вечером, в различные дни недели и т. д.

Аутентификация по физическому объекту

Второй метод аутентификации пользователей заключается в проверке некоторого физического объекта, который есть у пользователя, а не информации, которую он знает. Сегодня этим физическим объектом часто является пластиковая карта, вставляемая в специальное устройство чтения, подключенное к терминалу или компьютеру. Как правило, пользователь должен не только вставить карту, но также ввести пароль, чтобы предотвратить использование потерянной или украденной карты. С этой точки зрения операции с банкоматом (ATM, Automatic Teller Machine) начинаются с того, что пользователь регистрируется на компьютере банка с удаленного терминала (банкомата) при помощи пластиковой карты и пароля. Сегодня в большинстве стран применяется PIN-код (PIN, Personal Identification Number — личный идентификационный номер), состоящий всего из 4 цифр, что позволяет избежать необходимости установки полной клавиатуры на банкоматы.

Третий метод проверки подлинности основан на измерении физических характеристик пользователя, которые трудно подделать. Они называются *биометрическими* параметрами. Например, для идентификации пользователя может использоваться специальное устройство считывания отпечатков пальцев или распознавания тембра голоса. Визуальная идентификация пока не встречается, но со временем и она может появиться.

Другой метод идентификации заключается в анализе подписи. Пользователь ставит подпись специальным пером, соединенным с терминалом, и компьютер сверяет ее с оригиналом, хранящимся на удаленном сервере или в смарт-карте. Лучше всего сравнивать не саму подпись, а движения пера и давление пера на

бумагу. Хороший специалист по подделке подписей может нарисовать довольно точную копию подписи, но не обязательно угадает, в каком порядке выполняются движения, а также он не в силах в точности воспроизвести параметры скорости, ускорений и давления пера.

На удивление часто на практике используется измерение длины пальцев. При этом каждый терминал оснащается устройством вроде показанного на рис. 5.19. Пользователь засовывает в него руку, и устройство измеряет длину его пальцев и сравнивает с информацией, хранящейся в базе данных.

Однако такое «чисто конкретное» измерение «распальцовки» обладает существенным недостатком. Эту систему легко одурачить, подсунув ей гипсовый (или из другого материала) слепок, возможно даже с настраиваемой длиной выдвижных пальцев.

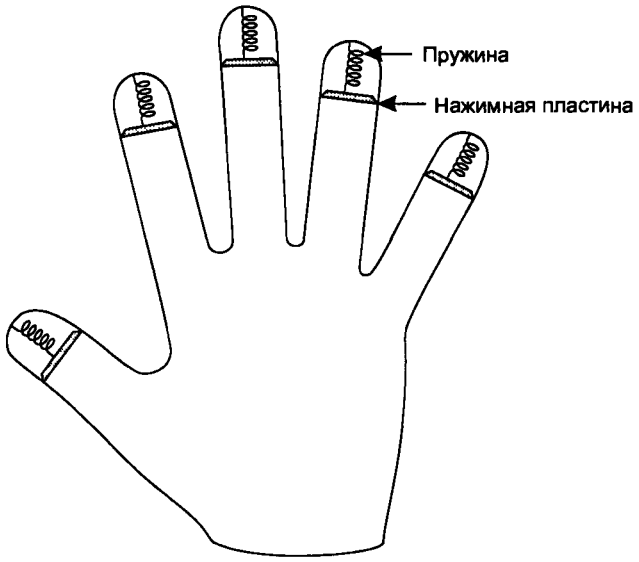


Рис. 5.19. Устройство для измерения длины пальцев

Подобные примеры биометрик можно приводить еще и еще, но мы остановимся только на двух, которые помогут сделать важное замечание по поводу здравого смысла. Кошки и другие животные мочой метят периметр своих владений. Очевидно, кошки могут идентифицировать друг друга подобным образом. Представьте себе, что кому-нибудь удастся создать небольшое устройство, способное производить мгновенный анализ мочи, обеспечивая, таким образом, надежную идентификацию. Раз так, значит, каждый терминал можно снабдить подобным устройством, вывесив табличку: «Для регистрации помочитесь сюда, пожалуйста». Возможно, таким образом удалось бы создать абсолютно надежную систему, хотя она вряд ли понравилась бы пользователям. То же самое можно сказать о системе, состоящей из иголки и небольшого спектрометра. Поль-

зователю предлагается проколоть палец иголкой, предоставив таким образом каплю крови для анализа.

Дело в том, что любая схема аутентификации должна быть психологически приемлема для сообщества аутентифицируемых. Измерение длины пальцев, возможно, не вызовет проблем, но даже такая безобидная процедура, как снятие отпечатков пальцев, может оказаться неоправданной, так как у многих это действие ассоциируется с обвинением в преступлении.

Контрмеры

В некоторых компьютерных системах, серьезно относящихся к вопросу безопасности (отношение к этому вопросу, как правило, кардинально меняется на следующий день после того как злоумышленник вломился в систему и причинил серьезный ущерб), часто предпринимаются шаги, призванные усложнить несанкционированный вход в систему. Так, компания может установить политику, разрешающую регистрацию сотрудников патентного отдела только с 8 часов утра до 5 вечера с понедельника по пятницу и только с машины, находящейся в патентном отделе. Любая попытка сотрудника патентного отдела зарегистрироваться в другие часы или не с того компьютера будет расцениваться как попытка взлома системы.

Коммутируемые телефонные линии также можно сделать более безопасными. Например, всем разрешается регистрироваться в системе через модем по телефонной линии, но после успешной регистрации система немедленно прерывает соединение и сама звонит пользователю по заранее условленному номеру. Такая мера означает, что взломщик не вломится в систему с любой телефонной линии. Для работы в системе подойдут только линии зарегистрированных пользователей. В любом случае, с применением данной техники или без нее, система обязана выдерживать паузу по крайней мере в 5 с при проверке пароля и увеличивать этот временной интервал по факту каждой неуспешной регистрации, чтобы снизить частоту попыток взломщика. После трех неуспешных попыток регистрации линия должна отключаться на 10 мин, а персонал уведомляться о попытке несанкционированного входа в систему.

Все попытки входа в систему должны регистрироваться. Когда пользователь регистрируется, система должна сообщать ему дату и время последней регистрации, а также терминал, использованный для входа, чтобы пользователь мог заметить взлом системы злоумышленником.

Еще один вариант защиты заключается в установке ловушки для взломщика. Простая схема ловушки представляет собой специальное имя регистрации с простым паролем (например, имя guest и пароль guest). При каждом входе в систему с таким именем системные специалисты в области безопасности немедленно уведомляются. Все команды, вводимые взломщиком, немедленно отображаются на мониторе руководителя службы безопасности, чтобы он мог видеть, что намеревается сделать взломщик.

Другие ловушки могут представлять собой легко обнаруживаемые ошибки в операционной системе и тому подобные вещи, намеренно встроенные с целью отлавливания злоумышленников на месте преступления. В 1989 году Столл на-

писал занимательный доклад о ловушках, установленных им с целью поймать шпиона, вломившегося на университетский компьютер в поисках военных секретов [75].

5.5. Механизмы защиты

В предыдущих разделах мы рассмотрели множество проблем, некоторые из них были техническими, тогда как другие — нет. В следующих разделах мы сконцентрируемся на некоторых технических деталях методов защиты файлов, используемых в операционных системах. Во всех этих методах проводится четкое разграничение между политикой (от кого и чьи данные должны защищаться) и механизмом (как система проводит данную политику). Отделение политики от механизма обсуждается в [69]. Мы уделим особое внимание именно механизму, а не политике.

В некоторых системах защита реализуется при помощи программы, называющейся *монитором обращений*. При каждой попытке доступа к некоторому ресурсу система сначала просит монитор обращений проверить законность данного доступа. Монитор обращений смотрит в таблицы политики и принимает решение. Ниже будет описано окружение, в котором работает монитор обращений.

5.5.1. Домены защиты

Компьютерная система подразумевает множество «объектов», которые требуется защищать. Это может быть аппаратура (например, центральный процессор, память, диски или принтеры) или программное обеспечение (процессы, файлы, базы данных или семафоры).

У каждого объекта есть уникальное имя, по которому к нему позволено обращаться, и набор операций, которые вправе выполнять с объектом процессы. Так, к файлу применимы операции read и write, с семафором имеют смысл операции up и down.

Очевидно, что для ограничения доступа к объектам требуется определенный механизм. Более того, этот механизм должен предоставлять возможность не полного запрета доступа, а ограничения в пределах подмножества разрешенных операций. Например, процессу A может быть разрешено читать файл F, но не писать в него.

Чтобы обсудить различные механизмы защиты, полезно ввести концепцию домена. *Домен* защиты представляет собой множество пар (объект, права доступа). Каждая пара указывает объект и некоторое подмножество операций, разрешенных с ним. *Права доступа* означают в данном контексте разрешение выполнить одну из операций. Домен может соответствовать одному пользователю или группе пользователей.

На рис. 5.20 показаны три домена, содержащие объекты и разрешения (RWX — Read, Write, eXecute — чтение, запись выполнение) для каждого объекта. Обратите внимание, что объект Printer1 одновременно присутствует в двух доменах.

Хотя это и не отражено в данном примере, один и тот же объект может иметь в различных доменах разные разрешения.

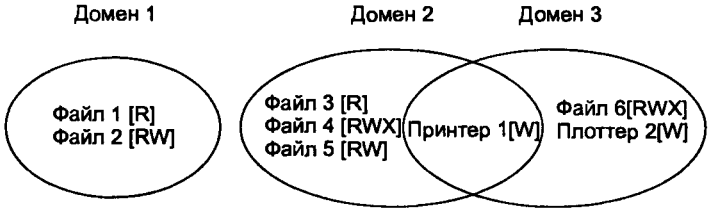


Рис. 5.20. Три домена защиты

В каждый момент времени каждый процесс работает в каком-либо одном домене защиты. Другими словами, имеется некоторая коллекция объектов, к которым он вправе получить доступ, и для каждого объекта у него есть определенный набор разрешений. Во время выполнения процессы имеют право переключаться с одного домена на другой домен. Правила переключения между доменами в большой степени зависят от системы.

Чтобы идея домена защиты выглядела более конкретно, рассмотрим пример из системы UNIX. В UNIX домен процесса определяется идентификаторами UID и GID процесса. По заданной комбинации (UID, GID) можно составить полный список всех объектов (файлов, включая устройства ввода/вывода, представленные в виде специальных файлов и т. д.), к которым процесс может получить доступ с указанием типа доступа (чтение, запись, исполнение). Два процесса с одной и той же комбинацией (UID, GID) будут иметь абсолютно одинаковый доступ к одинаковому набору объектов.

Более того, каждый процесс в UNIX состоит из двух частей: пользовательской и системной. Когда процесс обращается к системному вызову, он переключается из пользовательской части в системную. Пользовательская и системная части процесса различаются по уровню доступа к различным множествам объектов. Например, системная составляющая может иметь доступ ко всем страницам в физической памяти, ко всему диску и ко всем другим защищенным ресурсам. Таким образом, системный вызов осуществляет переключение доменов защиты.

Когда процесс выполняет системный вызов **exec** с файлом, у которого установлен бит **SETUID** или **SETGID**, процесс может получить новые идентификаторы UID и GID. При новой комбинации UID и GID процесс получает новый набор доступных файлов и операций. Запуск программы с установленным битом **SETUID** или **SETGID** также представляет собой переключение домена, так как права доступа при этом изменяются.

Важным вопросом является то, как система отслеживает, какой объект какому домену принадлежит. Можно себе представить большую *матрицу*, в которой рядами являются домены, а колонками — объекты. На пересечении располагаются ячейки, содержащие права доступа для данного домена к данному объекту. Матрица для рис. 5.20 показана на рис. 5.21. При наличии подобной матрицы

операционная система может для каждого домена определить разрешения к любому заданному объекту.

Переключение между доменами также легко реализуется при помощи все той же матрицы, если считать домены объектами, над которыми разрешена операция enter (вход). На рис. 5.22 снова изображена та же матрица, что и на предыдущем рисунке, но с тремя доменами, выступающими и в роли объектов. Процессы в состоянии переключаться с домена 1 на домен 2, но обратно вернуться уже не могут. Эта ситуация моделирует выполнение программы с установленным битом SETUID в UNIX. Другие переключения доменов в данном примере не разрешены.

Домен	Объект							
	Файл 1	Файл 2	Файл 3	Файл 4	Файл 5	Файл 6	Принтер 1	Плоттер 2
1	Чтение	Чтение Запись						
2			Чтение	Чтение Запись Исполнение	Чтение Запись		Запись	
3						Чтение Запись Исполнение	Запись	Запись

Рис. 5.21. Матрица защиты

Домен	Объект										
	Файл 1	Файл 2	Файл 3	Файл 4	Файл 5	Файл 6	Принтер 1	Плоттер 2	Домен 1	Домен 2	Домен 3
1	Чтение	Чтение Запись								Enter	
2			Чтение	Чтение Запись Исполнение	Чтение Запись		Запись				
3						Чтение Запись Исполнение	Запись	Запись			

Рис. 5.22. Матрица защиты с доменами в роли объектов

5.5.2. Списки управления доступом

На практике, однако, разрешения доступа редко оформляются в виде матрицы, показанной на рис. 5.22, поскольку такая матрица была бы очень большой и практически пустой. Поэтому, как правило, такие матрицы хранятся по рядам или столбцам, причем хранятся только непустые элементы. Эти два подхода, как ни странно, различаются между собой. В данном разделе мы рассмотрим матрицу защиты в варианте по столбцам, а в следующем разделе познакомимся с ее построчной реализацией.

Согласно первому методу, с каждым объектом ассоциируется список (упорядоченный), содержащий все домены, которым разрешен доступ к данному объекту, а также тип доступа. Такие списки называются *ACL-списками* (Access Control List — список управления доступом. Если бы данный способ управления доступом реализовывался в UNIX, проще всего было бы для каждого файла завести отдельный дисковый блок, хранящий список, и включить номер этого блока в *i*-узел файла. Так как здесь хранятся только заполненные ячейки матрицы доступа, места потребуется гораздо меньше, чем для хранения всей матрицы.

Как работают списки управления доступом, мы рассмотрим на примере UNIX, где домен определяется парой (UID, GID). Действительно, в системе MULTIX, по образу которой строилась UNIX, ACL работали примерно так же, поэтому пример не столь уж гипотетический.

Предположим, имеется четыре пользователя (то есть четыре UID): Jan, Els, Jelle и Maarike, которые входят соответственно в группы system, staff, student, student. Далее, предположим, что имеются файлы со следующими ACL:

File0: (Jan, *, RWX)

File1: (Jan, system, RWX)

File2: (Jan, *, RW–), (Els, staff, RW–), (Maarike, *, RW–)

File3: (*, student, R––)

File4: (Jelle, *, –––), (*, student, R––)

Каждая из записей ACL состоит из трех частей: UID, GID и разрешенные способы доступа: Read (чтение), Write (запись), eXecute (исполнение). Звездочка означает любую группу пользователей или любого пользователя. Тогда файл File0 могут читать, записывать и исполнять те процессы, у которых UID = Jan и с любым GID. Доступ к файлу File1 могут получить только процессы с UID = Jan и GID = system. Процесс, у которого UID = Jan и GID = staff, может обращаться к файлу File0, а к файлу File1 — нет. Файл File2 вправе читать и изменять процессы с UID = Jan, независимо от GID, кроме того, его разрешено читать процессам с UID = Els и GID = staff, или с UID = Maarike и любым GID. Файл File3 могут читать любые студенты (группа student). Особенно интересен файл File4. ACL этого файла гласит, что любой процесс с UID = Elle, независимо от GID, не имеет к файлу никакого доступа, а всем остальным студентам он доступен. Благодаря ACL можно запретить доступ к объекту для отдельных пользователей или групп, в то же время разрешая доступ остальным представителям того же класса.

Итак, достаточно говорить о том, чего нет в UNIX. Давайте теперь изучим то, что есть. В UNIX выделяются группы из трех битов для владельца файла, группы владельцев и для остальных. Эта схема, опять же, является ACL, но, на этот раз, ужатым до 9 бит. Это ассоциированный с файлом список, описывающий, кому и что разрешено делать с файлом. Хотя принятая в UNIX 9-битная схема, очевидно, имеет меньше возможностей, чем полноценная система с ACL, на практике она вполне адекватна, а ее реализация намного проще и дешевле.

Владелец объекта вправе в любой момент изменить его ACL, благодаря чему можно легко запретить доступ тем, кому он ранее был разрешен. Единственная

проблема в том, что изменение ACL почти наверняка не затронет тех пользователей, которые используют объект в текущий момент (то есть держат файл открытым).

5.5.3. Мандаты

Матрица, показанная на рис. 5.22, может также храниться по рядам. Здесь с каждым процессом ассоциирован список разрешенных для доступа объектов вместе с информацией о том, какие операции разрешены, другими словами, это домен защиты объекта. Такой список называется *списком полномочий*, а его элементы называются **возможностями**.

Типичный пример списка полномочий приведен в табл. 5.3. У каждого мандата есть поле Type, сообщающее тип объекта, поле Rights, являющееся битовой картой, описывающей допустимые для данного типа объектов операции, и поле Object, содержащее указатель на сам объект (например, номер его *i*-узла). Списки полномочий сами являются объектами, на которые позволено ссылаться из других списков, тем самым упрощается совместное использование поддоменов. На мандаты часто ссылаются по номеру в перечне. Например, процесс может сказать что-нибудь вроде: «прочитать 1 Кбайт из файла, на который ссылается мандат 2». Такой способ адресации подобен тому, как в UNIX используются дескрипторы.

Таблица 5.3. Список полномочий для домена 2 на рис. 5.21

#	Тип	Мандат	Объект
0	Файл	R--	Указатель на файл File3
1	Файл	RWX	Указатель на файл File4
2	Файл	RW-	Указатель на файл File5
3	Указатель	-W-	Указатель на принтер Printer1

Очевидно, что списки полномочий (или, как они сокращенно называются, *С-списки*) должны быть защищены от искажения их пользователями. Известны три способа защиты. Для первого требуется *теговая архитектура*, то есть аппаратно реализованная структура памяти, в которой у каждого слова памяти есть дополнительный (теговый) бит, сообщающий, содержит ли данное слово памяти мандат или нет. Теговый бит не используется в обычных командах процессора, таких как арифметические или команды сравнения. Изменен он может быть только программой, работающей в режиме ядра (то есть операционной системой).

Особенность второго способа в том, что список хранится внутри операционной системы. При этом к элементам списка можно обращаться по их позиции в списке. Такая форма адресации также напоминает использование дескрипторов файла в UNIX. Именно таким образом работала система Hydra [88].

Третий способ заключается в хранении списка мандатов в пространстве пользователя, но в зашифрованном виде, так, чтобы пользователь не сумел изменить эту информацию. Эта схема была разработана для распределенной операционной системы Amoeba и активно в ней применялась [81].

Помимо специфических разрешений, зависящих от конкретного объекта, например чтение и исполнение, мандаты (как системные, так и защищенные шифрованием) включают в себя, как правило, *общие права*, то есть разрешения выполнения действий, применимых ко всем объектам. Примерами таких действий являются:

- ✦ копирование элемента списка: создание нового элемента для того же объекта;
- ✦ копирование объекта: создание дубликата объекта с новым мандатом;
- ✦ удаление элемента списка: удаление записи в нем, объект при этом не затрагивается;
- ✦ удаление объекта: удаление объекта и элемента списка.

Напоследок стоит отметить, что аннулирование доступа к объекту в мандатных системах, реализованных на уровне ядра, довольно сложно. Системе трудно найти все мандаты для конкретного объекта, чтобы забрать их, так как они могут храниться по всему диску. Один из методов заключается в том, что элемент списка должен указывать не на сам объект, а на косвенно адресованный объект. Система может в любой момент разорвать связь между объектом и указывающим на него косвенным объектом, таким образом аннулируя все полномочия. (Когда мандат позднее появляется в системе, пользователь обнаружит, что объект косвенной адресации теперь указывает на нулевой объект.)

В системе Amoeba аннулирование разрешений выполняется легко. Все, что для этого требуется, — это изменить контрольное поле, хранимое с объектом. «Одним щелчком» все существующие мандаты объявляются недействительными. Однако ни одна схема не обеспечивает выборочного аннулирования разрешений, то есть невозможно, например, отменить полномочия Джона, не затронув всех остальных пользователей. Этот недостаток присущ всем мандатным системам.

5.5.4. Секретные каналы

Даже в корректно реализованной системе, в основе которой лежит правильная модель безопасности и безопасность которой была доказана, не исключены проколы защиты. В данном разделе мы обсудим, как информация может утекать на сторону даже в системах, для которых строго математически было доказано, что подобные утечки невозможны. Идеи, изложенные ниже, были высказаны Лэмпсоном в [52].

Модель Лэмпсона изначально была сформулирована в терминах единой системы разделения времени, но те же идеи применимы к локальной сети, а также в других многопользовательским средах. В ее чистом виде в эту модель входят три процесса, работающих на одной защищенной машине. Первый процесс представляет собой клиента, который доверяет выполнить некоторое задание второму процессу, серверу. Клиент и сервер доверяют друг другу не полностью. Например, работа сервера заключается в том, чтобы помочь клиентам заполнить их налоговые декларации. Клиенты беспокоятся, что сервер запишет в тайную тет-

радь их финансовую информацию, а затем, например, продаст эти сведения. Сервер озабочен тем, что клиенты украдут ценную программу подсчета налогов.

Третий процесс, называемый в данной модели «сообщником» (буквально collaborator, то есть «партнер»), намеревается украсть конфиденциальные сведения клиента. Владельцем этого процесса и сервера, как правило, является один и тот же человек. Все три процесса показаны на рис. 5.23. Цель данной модели — помочь разработать систему, в которой невозможна утечка информации, полученной сервером у клиента, к процессу-«подельнику». Лэмпсон назвал это *проблемой ограждения.*

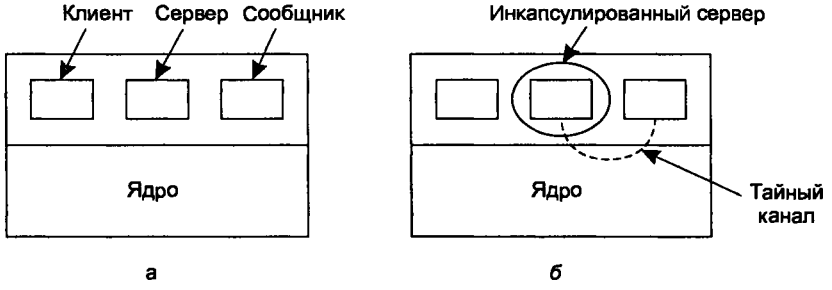


Рис. 5.23. а — клиент, сервер и «сообщник»; *б* — от инкапсулированного сервера информация все равно может утекать «сообщнику» по тайному каналу

С точки зрения разработчика системы задача заключается в инкапсуляции или ограждении сервера таким образом, чтобы он не мог передать информацию «сообщнику». С помощью схемы матрицы защиты несложно гарантировать, что сервер не сможет общаться с «сообщником», записывая данные в файл, к которому у «сообщника» есть доступ для чтения. Вероятно, можно также гарантировать отсутствие общения между сервером и «сообщником» при помощи системного механизма межпроцессного взаимодействия.

К сожалению, утечка информации может происходить по секретным каналам. Например, сервер передает последовательность нулей и единиц, кодируя единицы интервалами высокой активности процессора, а нули обозначая интервалами бездействия. «Сообщник» пытается принять этот поток, тщательно отслеживая время отклика сервера. Как правило, время отклика будет меньшим у простаивающего сервера, а именно нулевое. Этот канал связи называется секретным каналом. Он показан на рис. 5.23, *б*.

Конечно, тайный канал по определению зашумлен, но если применить помехоустойчивое кодирование (например, код Хэмминга), информация может приниматься «сообщником» с высокой степенью надежности. Пропускная способность такого канала будет невелика, но это не снижает его опасности. Очевидно, что ни одна из моделей защиты, основанных на матрицах объектов и доменов, не в силах предотвратить данный тип утечки информации.

Модуляция не является единственным вариантом тайного канала. Информация также может кодироваться страничными прерываниями, например, много прерываний в течение определенного интервала времени будет означать 1,

а отсутствие страничных прерываний — 0. В принципе для этой цели подойдет любой способ снижения производительности системы в течение определенного интервала времени. Если система позволяет блокировать файлы, тогда сервер может блокировать некий файл, что будет означать, например, 1, и разблокировать его, кодируя, таким образом, 0. Некоторые системы позволяют процессу определить, что файл блокирован, даже если у него нет доступа к этому файлу.

Для передачи скрытых сигналов также применимы захват и освобождение внешних ресурсов (например, магнитофонов, плоттеров и т. д.). В системе UNIX сервер может создавать файл, что будет означать 1, и удалять его, передавая 0. «Сообщник» может проверять наличие файла при помощи системного вызова access. Этот системный вызов будет работать, даже если у «сообщника» нет доступа к создаваемому сервером файлу. К сожалению, помимо таких «семафоров», существует множество других возможностей создания скрытого канала.

Лэмпсон также упомянул о еще одном возможном потайном канале связи, но уже между сервером и человеком. Например, сервер может сообщать, сколько работы он сделал для клиента, выставляя ему счет. Если настоящий счет составляет $100, а доход клиента составил $53 000, то сервер может сообщить об этом в виде счета в $100,53.

Обнаружить все скрытые каналы чрезвычайно трудно, не говоря уже об их блокировке. На практике в наших силах немногое. Добавление процесса, вызывающего страничные прерывания случайным образом или снижающего производительность системы другим способом, с целью снизить пропускную способность скрытых каналов, в качестве варианта решения проблемы не импонирует.

5.6. Обзор файловой системы MINIX

Как и любая другая файловая система, файловая система MINIX обязана решать рассмотренные нами задачи. Она должна выделять и освобождать пространство для файлов, следить за блоками на диске и за свободным местом, предоставлять какие-либо средства защиты от несанкционированного доступа и т. д. В оставшейся части главы мы более подробно коснемся того, как эти вопросы решаются в MINIX.

В первой части главы мы, ради большей общности, часто ссылались на UNIX, а не на MINIX, хотя внешний интерфейс этих двух систем практически идентичен. Теперь же мы сосредоточимся на внутреннем устройстве MINIX. Информацию о внутреннем устройстве UNIX вы можете почерпнуть из «монографий» Томпсона (1978), Баха (1981), Лайонса (1996) и Вахалия (1996).

В MINIX файловая система — это просто большая программа на языке C, работающая в пользовательском пространстве. Чтобы прочитать или записать файл, пользовательские процессы отправляют файловой системе сообщения, говорящие, что нужно сделать. Файловая система выполняет свою работу и отправляет обратно ответ. Фактически такая система представляет собой сетевой файловый сервер, оказавшийся на той же машине, что и обращающийся к нему процесс.

Такое устройство имеет несколько важных следствий. Прежде всего, файловую систему можно модифицировать, экспериментировать с ней и тестировать ее практически независимо от остальных частей MINIX. Далее, файловую систему можно легко перенести на другой компьютер, где есть компилятор C, скомпилировать ее там и использовать как отдельный удаленный UNIX-подобный файловый сервер. Единственные изменения коснутся того, как отправляются и принимаются сообщения, поскольку это делается по-разному на разных платформах.

В последующих разделах мы представим обзор многих ключевых областей устройства файловой системы. Особое внимание будет уделено сообщениям, структуре файловой системы, битовым картам, *i*-узлам, кэшированию блоков, путям и каталогам, дескрипторам файлов, блокированию файлов и специальным файлам (а также каналам). Рассмотрев эти темы, мы покажем, как все это работает вместе, проследив, что происходит, когда пользовательский процесс выполняет системный вызов read.

5.6.1. Сообщения

Файловая система понимает 39 типов сообщений, запрашивающих различные действия. Все они, кроме двух, соответствуют системным вызовам MINIX. Исключения составляют два сообщения, генерируемые другими компонентами MINIX. Что касается системных вызовов, то 31 из них принимаются от пользовательских процессов, 6 сообщений соответствуют системным вызовам, которые сначала обрабатываются менеджером памяти, который затем, чтобы завершить работу, вызывает файловую систему. Еще два сообщения также обрабатываются файловой системой. Все эти сообщения перечислены в табл. 5.4.

Таблица 5.4. Сообщения файловой системы. Имя файла всегда передается как указатель на строку. Ответное значение, отмеченное как «состояние», равно OK или ERROR

Сообщения от пользователя	Входные параметры	Ответное значение
ACCESS	Имя файла, режим доступа	Состояние
CHDIR	Имя нового рабочего каталога	Состояние
CHMOD	Имя файла, новая спецификация доступа	Состояние
CHOWN	Имя файла, новый владелец и группа	Состояние
CHROOT	Имя нового корневого каталога	Состояние
CLOSE	Дескриптор закрываемого файла	Состояние
CREAT	Имя создаваемого файла, режим	Дескриптор файла
DUP	Дескриптор файла (DUP2 требует два дескриптора)	Новый дескриптор файла
FCNTL	Дескриптор файла, код функции, аргументы	Зависит от функции
FSTAT	Имя файла, буфер	Состояние
IOCTL	Имя файла, код функции, аргументы	Состояние

Сообщения от пользователя	Входные параметры	Ответное значение
LINK	Имя файла, на который создается ссылка, имя ссылки	Состояние
LSEEK	Дескриптор файла, смещение, место, откуда считать смещение	Новое положение
MKDIR	Имя файла, спецификация доступа	Состояние
MKNOD	Имя каталога или признак специального файла, режим доступа и адрес	Состояние
MOUNT	Имя специального файла, точка монтирования, флаг только для чтения	Состояние
OPEN	Имя открываемого файла, флаг r/w	Дескриптор файла
PIPE	Указатель на два файловых дескриптора (модифицируются)	Состояние
READ	Дескриптор файла, буфер, сколько байтов	Количество прочитанных байтов
RENAME	Имя файла, имя файла	Состояние
RMDIR	Имя файла	Состояние
STAT	Имя файла, буфер для сохранения информации	Состояние
STIME	Указатель на текущее значение времени	Состояние
SYNC	Нет	Всегда OK
TIME	Указатель на место, куда будет записано текущее значение времени	Состояние
TIMES	Указатель на буфер для записи времени работы процесса и потомков	Состояние
UMASK	Дополнение к маске режимов доступа	Всегда OK
UMOUNT	Имя демонтируемого специального файла	Состояние
UNLINK	Имя файла	Состояние
UTIME	Имя файла, значения времени для него	Всегда OK
WRITE	Дескриптор файла, буфер, сколько байтов	Количество записанных байтов
Сообщения от менеджера памяти	**Входные параметры**	**Ответное значение**
EXEC	PID (идентификатор процесса)	Состояние
EXIT	PID	Состояние
FORK	PID родителя, PID потомка	Состояние
SETGID	PID, действительное и эффективное значения GID	Состояние
SETSID	PID	Состояние
SETUID	PID, действительное и эффективное значения UID	Состояние
Прочие сообщения	**Входные параметры**	**Ответное значение**
REVIVE	Процесс, который будет оживлен	Ответного сообщения нет
UNPAUSE	Процесс, который нужно проверить	(Пояснения в тексте)

Файловая система имеет ту же основную структуру, что и менеджер памяти или задачи ввода/вывода. В ней есть главный цикл, ожидающий сообщений. Когда сообщение принято, определяется его тип, который затем используется как индекс в таблице указателей на функции, обрабатывающие различные системные вызовы. Затем найденная процедура вызывается, она выполняет свои действия и возвращает код завершения. Далее файловая система отправляет ответное сообщение процессу, сделавшему вызов, и возвращается в начало цикла ожидать следующее сообщение.

5.6.2. Структура файловой системы

Файловая система MINIX представляет собой самостоятельную логическую сущность с *i*-узлами, каталогами и блоками данных. Она может храниться на любом блочном устройстве, например на дискете или жестком диске (или разделе жесткого диска). В любом случае, файловая система имеет одну и ту же структуру. Рисунок 5.24 иллюстрирует эту структуру на примере дискеты объемом 360 Кбайт с размером блока 1 Кбайт и 128 *i*-узлами. У файловой системы большего объема или с другим числом *i*-узлов или с иным размером блока будут те же самые шесть частей, только, возможно, их относительные размеры будут другими.

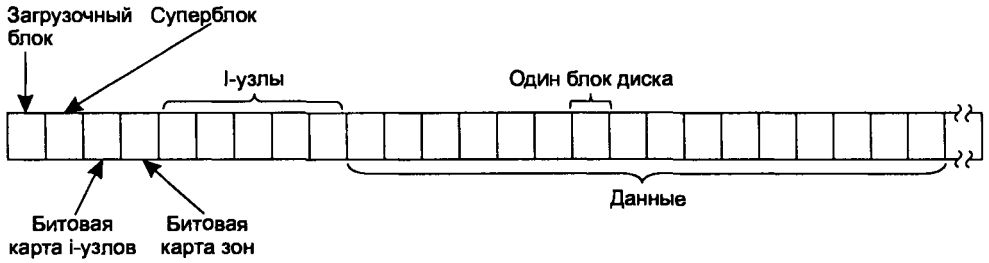


Рис. 5.24. Структура диска в простейшем случае: дискета 360 Кбайт, 128 i-узлов, размер блока 1 Кбайт (то есть два подряд идущих сектора по 512 байт рассматриваются как один блок)

Любая файловая система начинается с *загрузочного блока*. Этот блок содержит исполняемый код. При включении компьютера его аппаратное обеспечение считывает содержимое этого блока в память и исполняет его. Код в загрузочном блоке инициирует процесс загрузки самой операционной системы. После того как система загружена, загрузочный блок больше не используется. Для загрузки системы подходит не каждый дисковый накопитель, но ради единообразия структуры на каждом блочном устройстве резервируется загрузочный блок. Худшее, к чему может привести такая стратегия, — это потеря одного блока. Чтобы помешать аппаратному обеспечению пытаться загрузиться с устройства, не предназначенного для загрузки, в известное заранее место загрузочного блока записывается *сигнатура* («магическое число») в том и только том случае, если блок содержит исполняемый код. Аппаратура (а в действительности, код BIOS) отка-

жется загружаться с устройства, если загрузочный блок не имеет такой сигнатуры. Таким образом, случайный мусор не будет запущен как программа.

Суперблок содержит информацию, описывающую структуру файловой системы. Его строение показано на рис. 5.25.

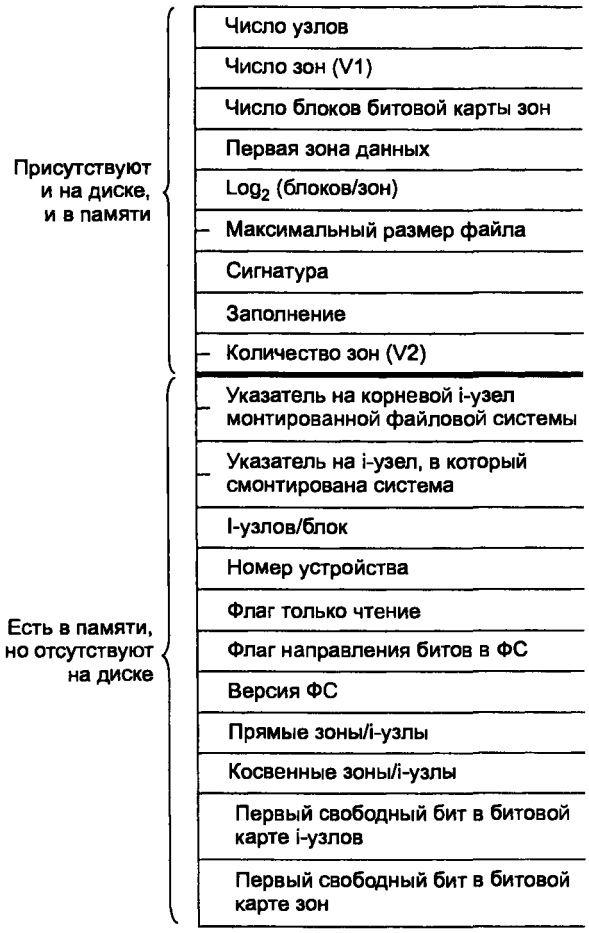


Рис. 5.25. Строение суперблока MINIX

Его основное назначение — сообщить файловой системе, насколько велики отдельные ее части. Зная размер блока и число *i*-узлов, несложно подсчитать размер битовой карты *i*-узлов и количество блоков *i*-узлов. Например, если блок имеет размер 1 Кбайт, каждый блок битовой карты будет содержать 1 Кбайт (8 Кбит), то есть будет использован для отслеживания состояния 8192 *i*-узлов. (Строго говоря, первый блок битовой карты описывает только 8191 *i*-узел, так как 0-го *i*-узла нет, но бит в битовой карте для него все равно выделяется.) Если всего имеется 10 000 *i*-узлов, то потребуется два блока для хранения битовой

карты. Так как каждый *i*-узел занимает 64 байта, блок размером 1 Кбайт может содержать до 16 *i*-узлов. При наличии 128 используемых *i*-узлов для их хранения потребуется 8 дисковых блоков.

Позже мы подробно объясним различие между дисковыми блоками и зонами, сейчас же достаточно сказать, что место на диске может выделяться частями (зонами) из 1, 2, 4, 8 или, в общем случае, 2^n блоков. Битовая карта зон отслеживает свободное пространство в зонах, а не в блоках. Для стандартных гибких дисков, используемых MINIX, размер зоны совпадает с размером блока (1 Кбайт), поэтому для таких устройств в первом приближении можно считать, что зона — это то же самое, что и блок. Пока мы позже в этой главе не приступим к обсуждению деталей выделения места, вы можете считать, что эти понятия эквивалентны.

Обратите внимание, что количество блоков в зоне не хранится в суперблоке, так как оно нигде не востребуется. Все, что нужно, — это логарифм по основанию два от этого числа, который используется как значение сдвига при преобразовании блоков в зоны, и наоборот. Например, если зона содержит 8 блоков, $\log_2 8 = 3$. То есть, чтобы найти зону, содержащую блок 128, нужно сдвинуть 128 на три бита вправо (получится зона 16).

Битовая карта зон содержит только зоны, занимаемые данными (то есть блоки, хранящие битовые карты и *i*-узлы, в нее не попадают), причем первая зона данных соответствует биту 1 в битовой карте. Как и в случае с картой *i*-узлов, нулевой бит не используется, а, значит, первый блок битовой карты описывает 8191 зон, а последующие — 8192 каждый. Если вы посмотрите на битовые карты только что отформатированного диска, вы увидите, что в обеих битовых картах, зон и *i*-узлов, установлено два бита. Первый из них соответствует несуществующей 0-й зоне или *i*-узлу. Второй соответствует *i*-узлу и зоне корневого каталога устройства, которая создается при создании файловой системы.

Информация, хранящаяся в суперблоке, избыточна, так как иногда она требуется в одной форме, а иногда в другой. Так как на размещение суперблока отводится 1 Кбайт места, имеет смысл хранить информацию во всех необходимых представлениях и не пересчитывать ее при работе. Например, номер первой зоны данных на диске можно вычислить, исходя из размера блока, размера зоны, числа *i*-узлов и числа зон, но быстрее просто хранить это значение в суперблоке. Оставшаяся часть суперблока все равно не используется, потому выделение в нем одного лишнего слова ничего не стоит.

Когда загружается ОС MINIX, суперблок с корневого устройства считывается в таблицу в памяти. Аналогичным образом, при монтировании других файловых систем их суперблоки также помещаются в память. В этой таблице есть несколько полей, которых нет на диске. Среди них флаг, индицирующий, что разрешено только чтение, флаг, позволяющий установить нестандартный порядок байтов, а также поля, предназначенные для ускорения доступа. Они указывают положение в битовых картах, ниже которого все биты установлены (то есть заняты). Кроме того, здесь есть поле, описывающее устройство, с которого пришел данный суперблок.

Прежде чем файловая система MINIX сможет использовать диск, он должен быть приведен в соответствие структуре, показанной на рис. 5.24. Для построе-

ния файловой системы имеется утилита mkfs. Она может быть вызвана, например, так:

```
mkfs /dev/fd0 1440
```

Такая команда создаст файловую систему из 1440 блоков на гибком диске в приводе 1. Ей можно указать файл-прототип с перечислением каталогов и файлов, подлежащих включению в созданную файловую систему. Эта же команда записывает в суперблок необходимую сигнатуру, идентифицирующую файловую систему как файловую систему MINIX. Эта сигнатура идентифицирует версию программы mkfs, при помощи которой создана файловая система, что позволяет в дальнейшем учесть различия между версиями. MINIX развивается, и в ранних версиях некоторые аспекты файловой системы (например, размер *i*-узла) были другими. При попытке монтировать дискету, имеющую другой формат, например дискету MS-DOS, системный вызов mount, проверяющий суперблок на наличие сигнатуры, сообщит об ошибке.

5.6.3. Битовые карты

MINIX следит за свободными и занятыми *i*-узлами и зонами при помощи двух битовых карт (см. рис. 5.25). Когда удаляется файл, несложно подсчитать, какой бит в битовой карте соответствует освободившемуся *i*-узлу, и найти его при помощи обычного механизма кэширования. В найденном блоке бит, соответствующий освободившемуся *i*-узлу, сбрасывается в 0. Зоны освобождаются точно так же, только используется битовая карта зон.

Логически, при создании файла файловая система должна последовательно просмотреть битовые карты, чтобы найти первый свободный *i*-узел. Фактически же, хранящийся в памяти суперблок содержит поле, указывающее на следующий свободный *i*-узел, поэтому потребуется искать свободный *i*-узел, начиная с этого положения (зачастую свободным оказывается следующий узел). Аналогичным образом, когда *i*-узел освобождается, проверяется, есть ли перед ним другие свободные, и при необходимости обновляется значение указателя. Если оказалось, что все *i*-узлы на диске заняты, процедура поиска указывает на 0-й элемент. Именно поэтому 0-й *i*-узел не используется (другими словами, он является индикатором заполнения диска). (Когда программа mkfs создает новую файловую систему, она обнуляет *i*-узел 0 и устанавливает младший бит в битовой карте, соответственно, файловая система никогда не пытается выделить этот блок.) Все, что только что было сказано для *i*-узлов, относится и к зонам. Когда необходимо дисковое пространство, логически ищется первая свободная зона, начиная с начала, а чтобы избежать ненужного последовательного поиска, поддерживается указатель на первую свободную зону.

Теперь мы можем объяснить разницу между зонами и блоками. В основе идеи зон лежит надежда гарантировать, что блоки одного файла расположены на одном цилиндре, с целью увеличения производительности последовательного чтения. Для этого используется подход, когда за один раз выделяется несколько блоков. Если, например, размер блока равен 1 Кбайт, а зоны — 4 Кбайт, битовая

карта зон отслеживает зоны, а не блоки. Диск объемом 20 Мбайт будет разбит на 5 К зон по 4 Кбайт, и в битовой карте зон, таким образом, будет 5 Кбит.

Большинство файловых систем работают с блоками. Обмен данными с диском всегда производится целыми блоками, кэш также оперирует блоками. Только та небольшая часть файловой системы, которая работает с физическим адресом (то есть с битовой картой зон и *i*-узлами), знает о зонах.

В процессе разработки файловой системы MINIX необходимо было принять некоторые решения о ее дизайне. В 1985 году, когда была задумана MINIX, диски имели небольшой объем, и ожидалось, что у многих пользователей будет только дисковод. Поэтому было решено в файловой системе V1 ограничить дисковый адрес 16 битами, чтобы впоследствии хранить большую их часть в блоках косвенной адресации. При 16-битном номере зоны и размере зоны в 1 Кбайт такая система позволяла работать с дисками объемом до 64 Мбайт. В те дни это был огромный объем, и казалось, что если диски станут больше, будет несложно переключиться на зоны размером 2 Кбайт или 4 Кбайт, не меняя размер блока. Благодаря 16-битному номеру зоны размер *i*-узла был ограничен 32 байтами.

Когда широко распространились диски значительно большей емкости, стало очевидно, что желательны изменения. Многие файлы имеют объем менее 1 Кбайт, поэтому увеличение размера блока привело бы к бесполезному расходованию пропускной способности диска из-за того, что записываются и считываются почти пустые блоки, и к бесполезной трате драгоценной оперативной памяти на кэш. Можно было бы увеличить размер зоны, но большие зоны означают менее эффективное использование свободного места на диске, при этом все равно желательно было бы сохранить эффективность работы с дисками маленького объема. Другой разумной альтернативой было бы задание разного размера зоны для больших и маленьких дисков.

В конце концов, было принято решение увеличить разрядность дискового указателя до 32 бит. Благодаря этому файловая система MINIX V2 могла бы работать с дисками объемом до 4 Тбайт, сохраняя размеры зоны и блока равным 1 Кбайт. Частично к этому выводу подталкивали и другие решения, касающиеся того, что должно храниться в *i*-узле. В результате стало разумным увеличение *i*-узла до 64 байт.

Зоны приводят еще к одной неожиданной проблеме, которую можно проиллюстрировать следующим простым примером, опять же, с зонами размером 4 Кбайт и блоками 1 Кбайт. Предположим, что имеется файл размером 1 Кбайт, для которого выделена одна зона. Последние три блока зоны содержат случайный мусор, оставшийся от предыдущих файлов, но ничего плохого в этом нет, так как в *i*-узле четко указано, что размер файла равен 1 Кбайт. Фактически, этот мусор не попадет даже в дисковый кэш, так как чтение производится блоками, а не зонами. Попытки прочитать информацию после конца файла всегда возвращают 0 и не возвращают никаких данных.

Предположим, что кто-то установил файловый указатель на 32 768 и дописал 1 байт. Теперь размер файла станет равным 32 769. Если после этого установить файловый указатель на 1 К и прочитать данные, удастся получить данные, которые были в данном блоке ранее, что является серьезной угрозой безопасности.

Решение состоит в том, чтобы в ситуации, когда производится запись после конца файла, явно обнулять все еще не выделенные блоки в зоне, которая ранее была последней. Хотя такая ситуация встречается достаточно редко, система должна ее отслеживать, в результате ее код несколько усложняется.

5.6.4. I-узлы

Схема *i*-узла MINIX показана на рис. 5.26. Она практически совпадает со строением *i*-узла в UNIX. Имеются девять 32-битных указателей на зоны диска, из которых семь прямых и два косвенных. В MINIX *i*-узел занимает 64 байта, как и в стандартной UNIX, поэтому остается место для дополнительного, десятого указателя, хотя в текущей версии файловой системы он не поддерживается. Время последнего доступа, модификации и изменения *i*-узла в MINIX хранятся стандартным образом, как в UNIX. Последний из этих параметров обновляется практически при каждой операции, за исключением чтения файла.

Когда файл открывается, его *i*-узел считывается в память и помещается в таблицу inode, где остается до закрытия файла. Записи в этой таблице имеют несколько дополнительных полей, которых нет на диске. Например, благодаря номеру *i*-узла и устройства, откуда он считан, файловая система знает, куда перезаписать *i*-узел, если он изменен в памяти. Кроме того, у каждого *i*-узла имеется счетчик. Если файл открывается дважды, вторая копия *i*-узла в память не копируется, вместо этого увеличивается на единицу значение счетчика. При закрытии файла счетчик декрементируется, и только тогда, когда он достигает нуля, *i*-узел удаляется из таблицы в памяти. Если за время нахождения в памяти он был изменен, он записывается на диск.

Главное предназначение *i*-узла в том, чтобы хранить сведения о положении блоков файла на диске. Первые семь номеров зон записываются в *i*-узел напрямую. Таким образом, при стандартных параметрах, когда зоны и блоки имеют размер 1 Кбайт, файлы размером до 7 Кбайт не требуют использования блоков косвенной адресации. После 7 Кбайт применяются «косвенные» блоки, подобно тому, как это показано на рис. 5.26, за тем исключением, что это только блоки первого и второго уровней косвенности. Если блоки и зоны имеют размер 1 Кбайт, косвенный блок первого уровня содержит 256 записей, что соответствует четверти мегабайта. Косвенный блок второго уровня ссылается на 256 блоков первого уровня, обеспечивая доступ к области в 64 Мбайт. Максимальный объем файловой системы MINIX составляет 1 Гбайт, и если потребуется реализовать работу с очень большими файлами, можно будет задействовать блоки третьего уровня косвенности, а также увеличить размер зоны.

Кроме того, *i*-узел хранит информацию о типе файла (обычный файл, каталог, специальный блочный файл, специальный символьный файл, канал ввода/вывода) и биты защиты, а также биты SETUID и SETGID. В поле link фиксируется, сколько разных каталогов ссылаются на данный файл, чтобы файловая система могла знать, когда следует освободить занимаемое им место. Не путайте этот счетчик со счетчиком количества открытий файла, обычно разными процессами (этот счетчик есть только в памяти в таблице inode).

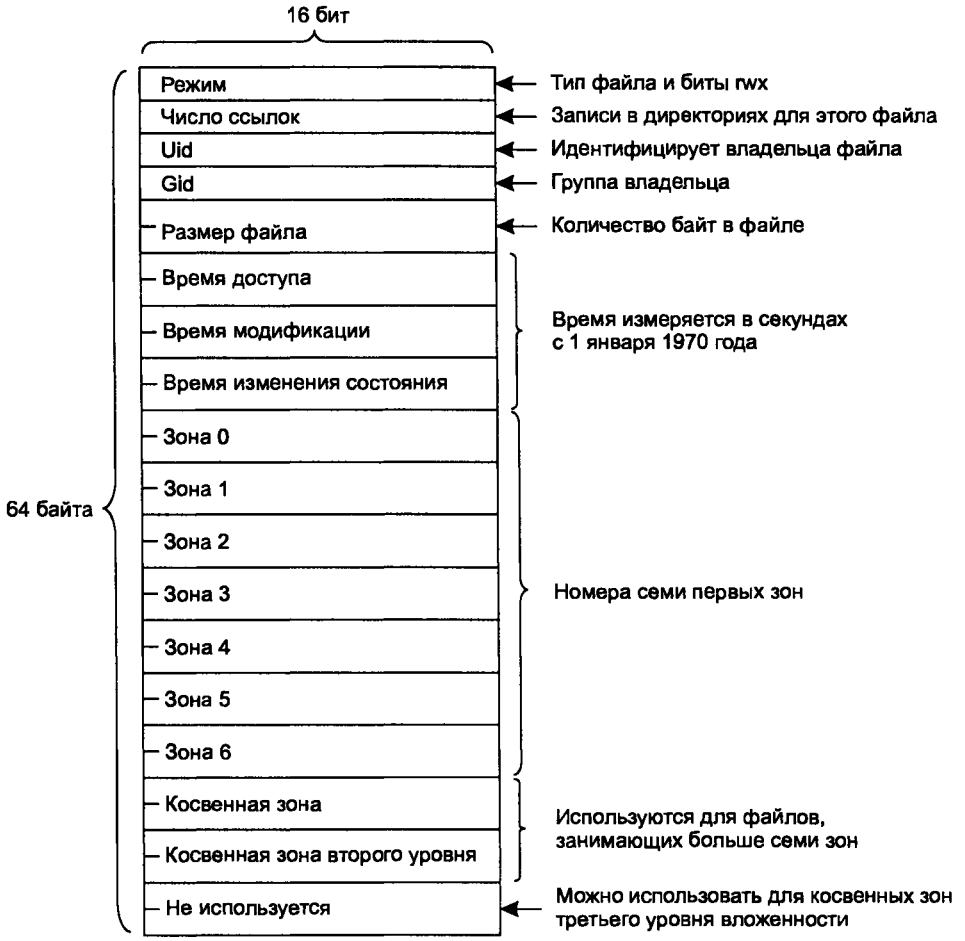


Рис. 5.26. Структура i-узла в MINIX

5.6.5. Кэш блоков

Для повышения производительности файловой системы в MINIX применяется кэширование блоков. Кэш реализован в виде массива буферов, каждый из которых состоит из заголовка, содержащего указатели, счетчики и флаги, и тела — области памяти для хранения одного блока. Все неиспользуемые буферы связаны друг с другом в двунаправленный список, отсортированный от недавно использованных (MRU) к последним использованным (LRU). Это изображено на рис. 5.27.

С целью быстрого извлечения из кэша нужного блока применяется цепочечный хеш. Все буферы, имеющие одинаковое хеш-значение k, сцеплены в однонаправленный список, на который ссылается запись k хеш-таблицы. Сделано так

по той причине, что применяемая хеш-функция просто извлекает n младших битов из номера блока, и хеш-значения могут совпадать. Каждый буфер принадлежит одной из цепочек. Конечно, сразу после загрузки MINIX все буферы свободны, поэтому все они находятся в одной цепочке, на которую ссылается нулевой элемент хеш-таблицы. Все остальные элементы таблицы в это время содержат нулевые указатели. Но после того, как система начнет работу, буферы будут исключаться из нулевой цепочки и будут формироваться новые цепочки.

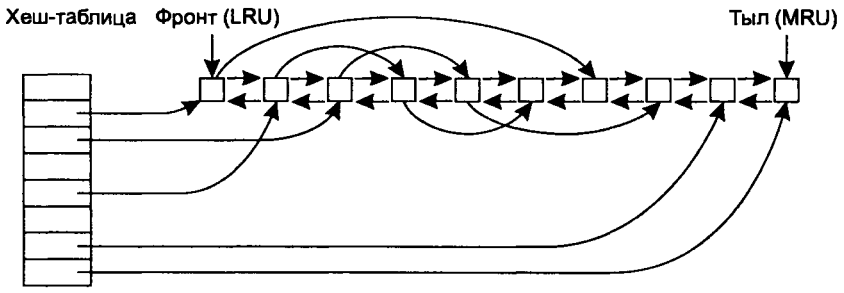


Рис. 5.27. Связный список кэша блоков

Если файловой системе необходим блок, она вызывает функцию get_block. Эта функция вычисляет хеш-код блока и ищет его в соответствующем списке. При вызове get_block ей передается как номер блока, так и номер устройства, и при поиске оба эти параметра сравниваются с соответствующими полями буферов из цепочки. Если нужный блок найден, увеличивается на единицу значение счетчика в его заголовке и возвращается указатель на него. Если затребованный буфер в кэше не обнаруживается, выбирается первый из LRU-списка; он гарантированно не используется, и содержащийся в нем блок может быть удален из оперативной памяти.

Когда удаляемый из памяти блок выбран, проверяется один из флагов в его заголовке, означающий, что блок был модифицирован после считывания. Если флаг установлен, блок записывается на диск. Затем с диска считывается нужный блок, для этого отправляется сообщение задаче диска. До тех пор пока запрошенный блок не будет прочитан, файловая система приостанавливается, а после этого возвращает указатель на прочитанный блок.

По окончании работы запросившей блок процедуры, чтобы освободить блок, она вызывает другую процедуру, put_block. Обычно блок используется сразу после выделения и затем освобождается, но так как существует возможность того, что перед освобождением блока будут сделаны дополнительные запросы, функция put_block сначала уменьшает на единицу счетчик использования и помещает блок обратно в LRU-список в том и только том случае, если счетчик достиг нуля. Если он имеет ненулевое значение, блок не освобождается.

Один из аргументов put_block описывает, к какому классу принадлежит освобождаемый блок (то есть i-узлы, каталоги, данные). В зависимости от этого значения принимаются два ключевых решения.

1. Поместить блок в начало или в конец LRU-списка.
2. Записать блок на диск немедленно (если он был модифицирован).

Блоки, которые с малой вероятностью скоро потребуются, например суперблоки, помещаются в начало списка, поэтому, когда в следующий раз потребуется свободный буфер, он будет взят из начала. Все остальные блоки присоединяются в конец списка, в полном соответствии с идеей сортировки по частоте использования.

Модифицированный блок не записывается до тех пор, пока не произойдет одно из двух следующих событий.

1. Блок достиг начала LRU-списка и изгоняется из памяти.
2. Выполнен системный вызов sync.

Вызов sync не перебирает элементы LRU-списка. Вместо этого он обрабатывает массив буферов и записывает модифицированные буферы на диск даже в том случае, если они еще используются.

Тем не менее есть одно исключение. Модифицированный суперблок записывается на диск немедленно. В старой версии MINIX суперблок модифицировался при монтировании системы, поэтому, чтобы уменьшить вероятность повреждения файловой системы по причине неожиданного сбоя, он сохранялся без промедления. Сейчас суперблок не модифицируется, и код, отвечающий за его срочную запись, является анахронизмом. В стандартной конфигурации никакие другие блоки не записываются сразу. Но это поведение можно изменить, варьируя значение параметра ROBUST в файле конфигурации системы, include/minix/ config.h. С его помощью можно сделать так, чтобы файловая система помечала *i*-узлы, каталоги, блоки косвенной адресации и блоки битовых карт как записываемые незамедлительно. Это должно сделать файловую систему более устойчивой, но ценой устойчивости является потеря производительности. Не совсем ясно и то, насколько это эффективно. Сбой питания, произошедший, когда еще не все блоки записаны на диск, вызовет головную боль, будь то блоки с данными или *i*-узлами.

Заметьте, что флаг в заголовке, означающий, что блок был модифицирован, устанавливается процедурой в файловой системе, которая его запросила и использует. Процедуры get_block и put_block занимаются только манипуляциями с однонаправленными списками. Они не имеют представления о том, какой процедуре файловой системы какой блок нужен и зачем.

5.6.6. Каталоги и пути

Другой важной составляющей файловой системы является система управления каталогами и путями. Многие системные вызовы, такие как open, в качестве аргумента принимают имя файла. Но в действительности нужен номер *i*-узла этого файла, поэтому файловая система должна просмотреть дерево каталогов и найти нужный *i*-узел.

В MINIX каталог состоит из файла, содержащего 16-байтовые записи. Первые два байта из шестнадцати формируют 16-битный номер *i*-узла, а оставшиеся 14 образуют имя файла. Это соответствует традиционной структуре каталога,

принятой в UNIX, которую вы видели на рис. 5.11. Если системе передан путь /usr/ast/mbox, она сначала ищет запись usr в корневом каталоге, затем — ast в /usr и, наконец, пытается найти mbox в /usr/ast. Как продемонстрировано на рис. 5.12, путь к файлу обрабатывается по одному компоненту за раз.

Единственная сложность возникает тогда, когда встречается смонтированная файловая система. В обычной конфигурации MINIX, как и во многих других UNIX-подобных операционных системах, имеется небольшая корневая файловая система, содержащая основные файлы, необходимые для запуска и обслуживания системы, а основное большинство файлов, включая пользовательские каталоги, находятся на отдельном устройстве, монтируемом в /usr.

Итак, настало время посмотреть, как работает монтирование. Когда пользователь вводит с терминала команду

```
mount /dev/hd2c /usr
```

файловая система, размещенная на втором разделе жесткого диска, монтируется в каталог /usr корневой файловой системы. Файловые системы до и после монтирования показаны на рис. 5.28.

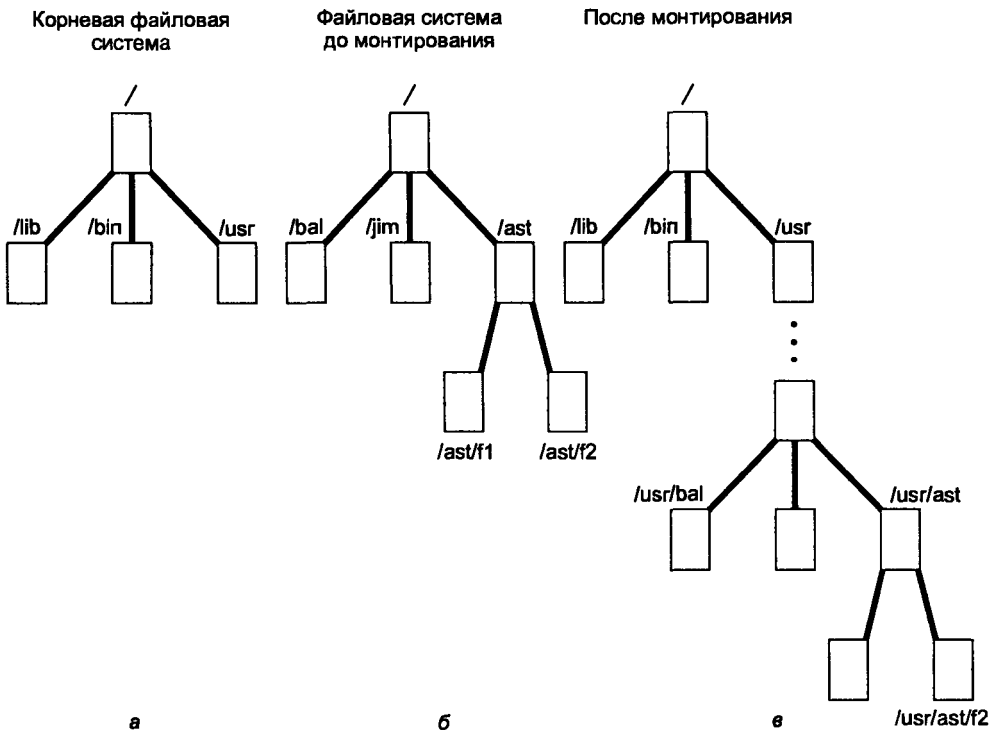


Рис. 5.28. *а* — корневая файловая система; *б* — не монтированная файловая система; *в* — результат монтирования файловой системы с (*б*) в каталог /usr

Ключевым моментом при монтировании является флаг, который устанавливается в хранящейся в памяти копии *i*-узла каталога /usr после успешного подсоединения. Этот флаг означает, что в данный каталог встроена файловая система. Кроме того, вызов mount загружает суперблок файловой системы в таблицу super_block и устанавливает два указателя на него. Дополнительно вызов записывает корневой *i*-узел смонтированной файловой системы в таблицу inode.

На рис. 5.25 можно увидеть, что в копии суперблока в памяти есть два поля, относящихся к монтированию файловых систем. В первое из них, i-node-of-the-mounted-file-system, записывается ссылка на корневой *i*-узел смонтированной файловой системы. Второе поле, i-node-mounted-on, хранит номер *i*-узла точки монтирования файловой системы, например, в нашем случае это будет *i*-узел /usr. Два этих поля служат для соединения смонтированной и корневой файловых систем и представляют собой тот самый «клей», который удерживает их вместе (на рис. 5.28, *в* это изображено точками). Благодаря этим полям и работают смонтированные файловые системы.

Когда файловой системе передается путь наподобие /usr/ast/f2, она увидит установленный в *i*-узле каталога /usr флаг и поймет, что продолжать поиск файла нужно с *i*-узла файловой системы, смонтированной в каталоге /usr. Возникает вопрос: «Как же она найдет этот корневой *i*-узел?»

Ответ прост. Система перебирает все хранящиеся в памяти суперблоки, пока не встретит тот, у которого поле i-node-mounted-on указывает на /usr. Это должен быть суперблок файловой системы, смонтированный в каталог /usr. Обнаружив его, несложно определить корневой *i*-узел соответствующей файловой системы. После этого можно продолжить поиск файла. В данном случае файловая система перейдет в каталог ast в корневом каталоге второго раздела жесткого диска.

5.6.7. Дескрипторы файлов

После того как файл открыт, пользователю возвращается дескриптор этого файла, который впоследствии может использоваться в системных вызовах read или write. В данном разделе мы рассмотрим, как файловая система обращается с дескрипторами.

Как и менеджер памяти и задача системы, файловая система поддерживает в своем адресном пространстве собственную часть таблицы процессов. Особенно интересны три поля этой таблицы. Первые два из них содержат указатели на корневой и рабочий каталоги. Поиск файла всегда начинается с одного из указанных каталогов, в зависимости от того, задан абсолютный или относительный путь. Значения этих указателей можно обновить при помощи системных вызовов chroot и chdir, которые изменяют соответственно текущие корневой и рабочий каталоги.

Третье интересное нам сейчас поле представляет собой массив, индексами в котором служат номера файловых дескрипторов. Этот массив служит для того, чтобы по данному дескриптору определить соответствующий ему файл. На первый взгляд, достаточно того, чтобы элементы этого массива представляли собой

указатели на *i*-узел файла. В конце концов, *i*-узел загружается в память при открытии файла и хранится там до закрытия файла, то есть *i*-узел гарантированно будет доступен.

К несчастью, этот простой план обречен на провал, так как в MINIX (как и в UNIX) можно совместно работать с файлом. Возникающая проблема связана с 32-битным числом, индицирующим, какой байт будет считан следующим. Это тот самый номер, называемый *указателем файла* (или указателем позиции в файле), который меняется системным вызовом lseek. Проблема выражается следующим вопросом: «Где должен храниться указатель файла?»

Первый вариант — поместить его в *i*-узел. Но, к сожалению, если два (или более) процесса будут в одно и то же время держать файл открытым, им придется завести свои собственные указатели, так как иначе вызов lseek, сделанный одним процессом, будет влиять на следующую операцию чтения другого процесса. Вывод: указатель файла нельзя хранить в *i*-узле.

А что насчет того, чтобы поместить его в таблицу процессов? Почему бы не взять второй массив, параллельный массиву файловых дескрипторов, в котором хранить файловый указатель для каждого файла? Эта идея также не работает, хотя и по более тонкой причине. Основная подоплека проблемы кроется в семантике системного вызова fork. Когда процесс ветвится, то и родительский, и дочерний процессы должны разделять единый указатель для всех открытых файлов.

Чтобы лучше понять проблему, представим себе сценарий оболочки, в котором стандартный вывод перенаправлен в файл. Когда оболочка ответвляет от себя первую программу, позиция в файле стандартного вывода для нее равна 0. Это значение наследуется потомком, который выводит в стандартный вывод, предположим, 1 Кбайт данных. После завершения потомка значение указателя должно быть равно 1 К.

Пусть теперь оболочка прочитала следующую команду сценария и ответвила следующую программу. Важно, чтобы эта программа унаследовала от оболочки текущее положение в файле, равное 1 К, чтобы начать вывод в файл с того места, на котором остановилась предшественница. Если оболочка не будет разделять значение указателя с дочерними программами, вторая программа, вместо того чтобы дописать свои данные в конец файла, перекроет ими вывод первой.

Таким образом, и в таблице процессов нельзя хранить файловые указатели. Они действительно должны разделяться. Для решения этой проблемы в MINIX используется новая разделяемая таблица, называемая filp. В ней хранятся значения всех файловых указателей. Применение этой таблицы демонстрируется на рис. 5.29. Благодаря тому что файловые указатели действительно используются совместно, можно корректно реализовать семантику вызова fork, и сценарии оболочки заработают правильно.

Единственным обязательным значением, которое необходимо хранить в таблице filp, является значение файлового указателя, но для удобства туда же записывается и указатель на *i*-узел. Массив файловых дескрипторов в таблице процессов при этом содержит указатели на записи в массиве filp. Кроме того, в таблице

filp находятся значения битов управления доступом, некоторые флаги, которые могут указывать, что файл открыт в каком-либо специальном режиме, и счетчик количества процессов, использующих данный указатель. Счетчик необходим для того, чтобы файловая система могла определить, когда завершится последний процесс, обращающийся к данной ячейке, и освободить ее.

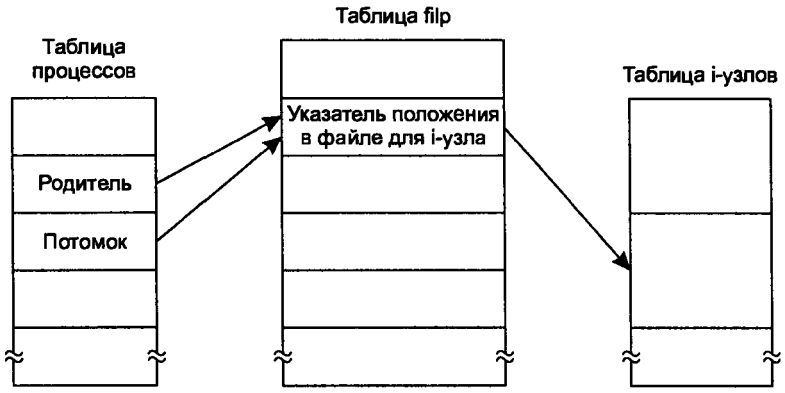


Рис. 5.29. Совместное использование файловых указателей родительским и дочерним процессами

5.6.8. Захват файлов

Специальная таблица нужна для реализации еще одного аспекта управления файловой системой. Это захват файлов. В MINIX поддерживается описываемый POSIX механизм *добровольной блокировки файла* (advisory file locking). Данный механизм позволяет отметить весь файл, его часть или несколько его частей как захваченные. Операционная система не мешает получать доступ к блокированным данным, но ожидается, что процессы будут вести себя прилежно и проверять наличие блокировки, прежде чем делать что-либо, что может привести к конфликту с другим процессом.

Причина, по которой для этого выделена отдельная таблица, аналогична оправданиям, приведенным в предыдущем разделе для таблицы filp. Один процесс может одновременно удерживать несколько файлов, и в то же время несколько частей одного файла может быть захвачено разными процессами (хотя захваченные области, конечно же, не перекрываются). Поэтому ни таблица процессов, ни filp не подходят для хранения этих данных. Так как на одном файле может быть установлено несколько блокировок, *i*-узел тоже не годится.

В MINIX для хранения сведений о всех блокировках вводится таблица file_lock. В каждой ячейке этой таблицы есть поля, указывающие тип захвата (захвачен ли файл на чтение или на запись), идентификатор процесса-владельца, указатель на *i*-узел блокированного файла, а также смещения первого и последнего байтов удерживаемой области.

5.6.9. Каналы ввода/вывода и специальные файлы

Каналы и специальные файлы имеют важное отличие от обычных файлов. Когда процесс пытается читать дисковый файл или писать в него, в худшем случае операция займет несколько сотен миллисекунд. Максимально может потребоваться два или три доступа к диску, не больше. При чтении из канала ситуация иная: если в канале данных нет, читающий из него процесс будет ждать до тех пор, пока кто-то другой не поместит их туда, на что могут потребоваться часы. Аналогично, при чтении с терминала процесс будет ждать до момента, пока пользователь что-нибудь не введет с клавиатуры.

Как следствие, обычное правило обслуживать запрос до его полного выполнения здесь не работает. Такие запросы нужно приостанавливать и запускать позже. Когда процесс делает попытку чтения из канала ввода/вывода, операционная система вправе немедленно проверить его состояние и выяснить, может ли операция быть завершена. Если да, операция выполняется, но если нет, файловая система сохраняет аргументы системного вызова в таблице процессов, чтобы перезапустить его, когда придет время.

Заметьте, что файловая система не предпринимает никаких действий, чтобы приостановить сделавший вызов процесс. Все, что она делает, — воздерживается от отправки ответного сообщения, в результате процесс остается заблокированным в ожидании ответа. Таким образом, после того как процесс приостановлен, файловая система возвращается в основной цикл, ожидая следующего системного вызова. Как только другой процесс приведет канал ввода/вывода в такое состояние, что операция, запрошенная приостановленным процессом, сможет выполниться, файловая система устанавливает флаг, чтобы при следующем проходе основного цикла извлечь из таблицы процессов аргументы системного вызова и выполнить его.

С терминалами и прочими символьными устройствами ситуация немного иная. У каждого специального файла в *i*-узле хранятся два числа, это старшее устройство и младшее устройство. Первое из этих чисел характеризует класс устройства (то есть RAM-диск, жесткий диск, терминал). Оно используется как индекс в одной из таблиц файловой системы, которая ставит в соответствие этому номеру задачу ввода/вывода (другими словами, драйвер). В результате главный номер определяет, какой драйвер вызвать. Второе число передается драйверу как аргумент. Оно определяет, какое устройство использовать, например терминал 2 или диск 1.

В некоторых случаях, особенно это касается терминалов, в младшем номере закодирована информация о категории устройств, обслуживаемых задачей. Например, основная консоль MINIX, /dev/console, имеет номера 4, 0. Виртуальные консоли обслуживаются той же частью драйвера. Это устройства /dev/ttyc1 (4, 1), /dev/ttys2 (4, 2) и т. д. Для обслуживания терминалов на последовательных линиях требуется другое низкоуровневое программное обеспечение, и этим устройствам присваиваются следующие номера: 4, 16 (/dev/tty00) и 4, 17 (/dev/tty01). Аналогичным образом, для обслуживания сетевых псевдотерминалов нужен от-

дельный код низкого уровня, и эти устройства, ttyp01 и ttyp02, получают такие номера, как 4, 128 и 4, 129. С каждым из псевдоустройств ассоциировано устройство ptyp1, ptyp2 и так далее с номерами 4, 192; 4, 193... Эти номера выбираются так, чтобы задаче терминала было проще вызывать низкоуровневые функции, необходимые для работы с каждой из групп объектов. Никто не ожидает, что когда-нибудь кому-нибудь придет в голову оснастить систему с MINIX более чем 192 терминалами.

Когда процесс читает специальный файл, файловая система извлекает из его *i*-узла старший и младший номера и по старшему номеру при помощи таблицы в файловой системе определяет номер задачи. Определив задачу, файловая система отправляет ей сообщение, параметрами которого являются младший номер, выполняемая операция, номер сделавшего вызов процесса, адрес буфера и количество байтов. Формат совпадает с показанным в табл. 3.4, с той разницей, что поле POSITION не используется.

Если драйвер устройства может выполнить работу немедленно (то есть строка уже введена в терминал), он копирует данные из собственного внутреннего буфера в пользовательский и отправляет файловой системе сообщение об успешном завершении работы. Затем файловая система посылает ответное сообщение пользователю, и вызов завершается. Обратите внимание: драйвер устройства не передает данные файловой системе. При работе блочных устройств данные передаются через кэш блоков, но со специальными символьными файлами это не так.

Если же драйвер не может сразу же выполнить работу, он записывает параметры сообщения в свои внутренние таблицы и рапортует файловой системе о своей недееспособности. В этот момент файловая система попадает в ту же ситуацию, которая возникает, когда кто-либо пытается прочитать из пустого канала ввода/вывода. Она фиксирует тот факт, что процесс был приостановлен, и ждет следующего сообщения.

Когда драйвер получит достаточно данных для выполнения запроса, он переносит их в буфер все еще приостановленного пользовательского процесса и информирует файловую систему о сделанном. Все, что осталось сделать файловой системе, — отправить пользовательскому процессу ответ, чтобы разблокировать его и сказать, сколько байтов передано.

5.6.10. Пример: системный вызов read

Как мы скоро увидим, большая часть кода файловой системы посвящена обслуживанию системных вызовов. Следовательно, естественно было бы заключить данный обзор кратким наброском того, как работает один из самых важных системных вызовов, read.

Когда пользовательская программа, чтобы прочитать обычный файл, выполняет команду

```
n = read (fd, buffer, nbytes);
```

вызывается библиотечная подпрограмма read с тремя аргументами. Она формирует из этих значений сообщение, помещает в него код операции read и отправ-

ляет его файловой системе, блокируясь в ожидании ответа. Получив сообщение, файловая система выбирает из таблицы адрес функции-обработчика, пользуясь типом сообщения как индексом. В данном случае будет выбрана процедура, ответственная за чтение.

Процедура извлекает из сообщения дескриптор файла, определяет по нему соответствующую запись в filp, а затем *i*-узел файла, который будет считываться (см. рис. 5.29). Затем запрос разбивается на такие части, чтобы каждая из них уместилась в один блок. Например, если текущее значение файлового указателя равно 600 и запрошен 1 Кбайт данных, запрос будет разбит на две части, от 600 до 1023 и от 1024 до 1623 (если размер блока равен 1 Кбайт).

Затем проверяется наличие каждого из этих блоков в кэше. Если блок в кэше отсутствует, файловая система выбирает последний из использовавшихся незанятых буферов и передает задаче диска сообщение перезаписать его, если в нем есть мусор. Затем задаче диска передается просьба считать блок.

После того как блок оказался в кэше, файловая система посылает задаче системы сообщение, чтобы она поместила данные в соответствующее место в пользовательском буфере (то есть байты от 600 до 1023 скопировать в начало буфера, а байты от 1024 до 1623 — по смещению 424 от начала). Выполнив копирование, задача системы отправляет пользователю сообщение, указывая в нем, сколько байтов было обработано.

Когда ответ достигает пользователя, библиотечная подпрограмма read извлекает из него код возврата и передает его значение в вызвавший процесс.

Тут есть один дополнительный шаг, который в действительности не является частью самого системного вызова read. Файловая система, выполнив чтение и отправив ответ, инициирует чтение следующего блока, если считывание производится с блочного устройства и удовлетворены некоторые условия. Так как чаще всего производится последовательное чтение, разумно ожидать, что при следующем чтении будет запрошен следующий блок файла, который, таким образом, заранее окажется в кэше.

5.7. Реализация файловой системы MINIX

Код файловой системы MINIX относительно велик (более 100 страниц кода на C), но довольно прямолинеен. В нем запрос на выполнение системного вызова принимается, выполняется и отправляется ответ. В последующих разделах мы пройдемся по этому коду файл за файлом, указывая на основные моменты. Сам код также содержит немало комментариев, чтобы облегчить задачу читателя.

Рассматривая код остальных составляющих MINIX, мы обычно сначала изучали главный цикл процесса, а затем процедуры, обслуживающие различные типы сообщений. Здесь же мы возьмем за основу другой подход. Сначала мы изучим основные подсистемы (работу с кэшем, *i*-узлами и т. д). Затем рассмотрим главный цикл и системные вызовы, работающие с файлами. Потом настанет пора системных вызовов для управления каталогами, и, наконец, мы обсудим оставшиеся системные вызовы.

5.7.1. Заголовочные файлы и глобальные структуры данных

Как и в ядре и менеджере памяти, в файловой системе применяется большое количество структур и таблиц, определяемых заголовочными файлами. Некоторые из этих структур помещены в общесистемные заголовочные файлы, лежащие в каталоге include/ и его подкаталогах. Например, файл include/sys/stat.h описывает формат представления информации *i*-узла для других программ, а структура данных каталога определяется файлом include/sys/dir.h. Оба этих файла предписаны стандартом POSIX. Управляют файловой системой множество глобальных определений, расположенных в файле include/minix/config.h, например, макрос ROBUST определяет, будут ли важные структуры данных записываться на диск немедленно, а NR_BUFS и NR_BUFS_HASH отвечают за размер кэша блоков.

Заголовочные файлы файловой системы

Собственные заголовочные файлы файловой системы расположены в каталоге src/fs/. Имена многих из них уже знакомы вам из изучения других частей системы MINIX. Главный заголовочный файл, fs.h, очень сходен с src/kernel/kernel.h и src/mm/mm.h. Он включает остальные заголовочные файлы, необходимые всем другим файлам кода на C из файловой системы. Как и везде, в главный заголовочный файл включаются локальные заголовочные файлы const.h, type.h, proto.h и glo.h. Их мы и рассмотрим далее.

В файле const.h определяются некоторые константы, такие как размеры таблиц или флаги, используемые далее по всей файловой системе. У MINIX уже есть история. В предыдущих версиях применялась другая файловая система, и для тех пользователей, которые хотят обращаться к файлам, записанным предыдущей версией, предоставлена поддержка обеих версий, как V1, так и текущей, V2. Суперблок файловой системы содержит *сигнатуру*, при помощи которой операционная система может определить тип файловой системы. Константы SPER_MAGIC и SUPER_V2 определяют эти сигнатуры. О поддержке предыдущих версий не только приходится иногда читать в теоретических статьях, об этом всегда приходится задумываться разработчику новой версии программного обеспечения. Ему необходимо решить, сколько прикладывать усилий для упрощения жизни пользователей старой реализации.

В файле type.h описаны как структуры предыдущей версии V1, так и новой, V2, в том виде, в котором они записываются на диск. Размер нового *i*-узла вдвое больше, чем в старой системе, которая разрабатывалась как компактная система для машин без жесткого диска и дискетами по 360 Кбайт. В новой версии выделено место для всех трех полей времени, которые есть в UNIX-системах. В *i*-узле версии V1 было всего одно поле времени, а вызовы stat и fstat, возвращающие структуру stat, содержащую все три поля, имитировали их наличие. При поддержке двух версий файловых систем есть небольшое затруднение, касающееся поля d2_gid структуры d2_inode. Старое программное обеспечение MINIX рассчитывает на то, что тип gid_t восьмибитный, поэтому поле d2_gid необходимо явно объявлять как u16_t.

В файле proto.h в форме, приемлемой как для стандартного ANSI C компилятора, так и для старых компиляторов а ля Керниган и Ричи, описаны прототипы функций. Это большой, но не очень интересный файл. Тем не менее тут нужно указать на один момент: так как файловая система обслуживает такое большое количество системных вызовов, код различных процедур do_xxx рассеян по нескольким файлам. Описания в proto.h организованы в порядке файлов, поэтому им удобно пользоваться, если вы хотите узнать, в каком из файлов находится интересующая вас функция.

Наконец, в glo.h описываются глобальные переменные. Здесь же находятся буферы входящих и исходящих сообщений. При описании переменных применяется уже знакомый вам трюк с макросом EXTERN, поэтому они доступны из всех частей файловой системы. Как и в других компонентах MINIX, место в памяти для этих переменных выделяется при компиляции файла table.c.

Принадлежащая файловой системе часть таблицы процессов содержится в файле fsproc.h. В нем при помощи макроса EXTERN объявляется массив fsproc. В последнем хранятся маска режима доступа, указатели на *i*-узлы текущих рабочего и корневого каталогов, массив файловых дескрипторов, UID, GID и номер терминала для каждого процесса. Здесь же можно найти идентификатор процесса и группы процессов, которые дублируют поля таблицы процессов, принадлежащие ядру и менеджеру памяти.

Несколько полей отведены под хранение аргументов остановленных на полпути системных вызовов, таких как чтение из пустого канала. Для полей fp_suspended и fp_received в действительности требуется только один бит, но для символа компиляторы практически всегда генерируют лучший код. Кроме того, есть поле для битов FD_CLOEXEC, требуемых стандартом POSIX. Их назначение в том, чтобы указать, что файл должен быть закрыт, когда делается системный вызов exec.

Теперь мы приступим к файлам, в которых описываются остальные таблицы, используемые файловой системой. Прежде всего, это файл buf.h, где задается кэш блоков. Все структуры в нем описаны при помощи EXTERN. Все буферы хранятся в массиве bufs, и каждый из них содержит область данных b, полный указателей заголовок и счетчики. Область данных объявлена как объединение пяти типов, поскольку иногда к блоку удобнее обращаться как к массиву символов, иногда как к каталогу и т. д.

Так, чтобы обратиться к области данных буфера 3 как к массиву символов, нужно использовать запись buf[3].b.b__data, так как buf[3].b ссылается на всю область данных, из которой выделяется поле b__data. Хотя такой синтаксис и правилен, он неуклюж, поэтому далее создается макрос b_data, позволяющий использовать запись buf[3].b.b_data. Обратите внимание: поле записывается как b__data, с двумя символами подчеркивания, а макрос b_data — с одним. Далее задаются аналогичные макросы для других способов обращения к данным блока.

Далее, сразу после этих макросов задается хеш-таблица буфера, buf_hash. Каждый из ее элементов ссылается на список буферов, которые сначала пусты. Макросы в конце файла buf.h определяют различные типы блоков. Когда использованный блок возвращается обратно в кэш, с ним передается одно из этих значений, по которому менеджер кэша решает, поместить ли блок в начало или

в конец списка LRU, а также записать его на диск сразу или нет. Бит WRITE_IMMED сигнализирует о том, что измененный блок должен быть перезаписан на диск немедленно. В обычных условиях этим битом помечается только суперблок. А что насчет других структур, помеченных MAYBE_WRITE_IMMED? Если в файле include/minix/config.h макрос ROBUST инициирован, это определение будет равно WRITE_IMMED, а в противном случае — нулю. В стандартной конфигурации MINIX ROBUST равен 0, поэтому блоки с такой пометкой будут записываться тогда же, когда и обычные блоки с данными.

Наконец, в последней строке на основе значения NR_BUF_HASH из include/minix/config.h задается параметр HASH_MASK. Это значение побитово объединяется с номером блока по принципу логического И (AND), с целью выяснить, в каком из списков массива buf_hash следует искать буфер блока.

В следующем файле, dev.h, определяется таблица dmap. Сама эта таблица описана в table.c с начальными значениями, поэтому ее описание не может быть использовано в других файлах. Чтобы иметь возможность обращаться к ней из других мест, и создан файл dev.h. В нем таблица dmap описана с ключевым словом extern вместо EXTERN. Назначение данной таблицы в том, чтобы отображать старший номер устройства на номер соответствующей задачи.

Файл file.h содержит промежуточную таблицу filp (описанную как EXTERN), применяемую для хранения текущего положения в файле и указателя на его *i*-узел (см. рис. 5.29). Эта же таблица дает ответ на вопрос, был ли файл открыт на чтение и/или на запись, а также сколько файловых дескрипторов в текущий момент ссылаются на ее элемент.

Таблица file_lock (объявленная как EXTERN) из файла lock.h хранит сведения о захвате файлов. Размер этого массива определяется параметром NR_LOCKS, который определяется в файле const.h и по умолчанию равен 8. Если когда-либо потребуется реализовать на основе MINIX многопользовательскую базу данных, это значение необходимо будет увеличить.

В inode.h объявляется (опять же, как EXTERN) массив *i*-узлов inode. В нем хранятся все используемые в текущий момент *i*-узлы. Как уже говорилось ранее, *i*-узел файла загружается в память при его открытии и хранится там до тех пор, пока файл не будет закрыт. В структуре inode есть информация, присутствующая в памяти, но отсутствующая на диске. Заметьте, что здесь описывается только одна версия и нет никаких зависящих от версии файловой системы особенностей. Различия между файловыми системами V1 и V2 учитываются при считывании *i*-узла с диска, и остальной системе не приходится задумываться о формате диска, по крайней мере до тех пор, пока не понадобится записать измененную информацию обратно.

К этому моменту вам должно быть понятно назначение большинства полей. Небольшого пояснения заслуживает только i_seek. Ранее упоминалось, что при последовательном чтении файловая система, в качестве оптимизации, пытается заранее помещать следующий блок в кэш, до того как он запрошен. При произвольном доступе к файлу опережающее чтение не нужно. Поэтому, когда делается вызов lseek, чтобы отключить опережающее чтение, устанавливается поле i_seek.

Файл param.h аналогичен файлу менеджера памяти с тем же именем. В нем определяются имена для полей сообщений с параметрами, чтобы, например, можно было писать buffer вместо m.m1_p1, имени одного из полей буфера сообщения m.

В файле super.h находится определение таблицы суперблоков. Она отвечает за загрузку суперблока корневой системы и сюда же записываются суперблоки монтируемых файловых систем. Как и другие таблицы, super_block объявлена с ключевым словом EXTERN.

Выделение памяти для данных файловой системы

Последний файл, который мы обсудим, table.c, не является заголовочным. Но, как и за разговором о менеджере памяти, имеет смыл рассмотреть его сразу после заголовочных файлов, поскольку все они присоединяются при компиляции table.c. Большинство упомянутых нами структур данных объявлены при помощи EXTERN, это касается как глобальных переменных, так и локальной части таблицы процессов. Так же как и в других частях MINIX, память для таких переменных в действительности выделяется при компиляции table.c. Кроме того, в этом файле содержатся два важных неинициализированных массива. Массив call_vector хранит указатели на функции, он нужен в главном цикле, чтобы определить, какая функция выполняет какой системный вызов. Похожую таблицу мы видели и внутри менеджера памяти.

Таблица dmap аналогов в менеджере памяти не имеет. Здесь есть строки для каждого из значений старшего номера устройства, начиная с нуля. Когда устройство закрывается или открывается или когда информация записывается на него или считывается с него, адрес процедуры, выполняющей соответствующую операцию, берется из этой таблицы. Все эти процедуры расположены в адресном пространстве файловой системы. Большинство из них ничего не делают, но некоторые передают вызов задаче, запрашивая ввод/вывод. Номер задачи тоже хранится в таблице dmap.

Когда в MINIX добавляется новый класс устройств, в эту таблицу необходимо включить соответствующую строку, указывающую, какие действия следует выполнять (если вообще выполнять) для открытия, закрытия, чтения или записи. Вот простой пример. Если в MINIX добавить поддержку накопителя на магнитной ленте, то процедура открытия из таблицы должна контролировать, занят ли привод. Чтобы сберечь силы пользователей, для автоматизации процесса изменения этой таблицы определен макрос DT.

В таблице dmap есть строки для всех возможных значений старшего номера устройства, и все эти строки оформлены как макросы. У обязательных устройств аргумент макроса enable равен 1. Некоторые из записей не используются либо потому, что соответствующий драйвер еще не готов, либо потому, что драйвер был удален. У таких записей параметр enable равен 0. Тем устройствам, наличие которых управляется файлом конфигурации, соответствуют записи, где в качестве аргумента enable подставлен макрос, включающий данное устройство, например ENABLE_WINI.

5.7.2. Таблицы

С каждой важной таблицей, будь то таблица блоков, *i*-узлов, суперблоков и т. д., связан файл, содержащий процедуры для работы с ней. Эти процедуры широко используются в остальном коде и образуют основной интерфейс между таблицами и файловой системой. Поэтому наше изучение кода файловой системы имеет смысл начать с этих файлов.

Управление блоками

С кэшем блоков работают подпрограммы из файла cache.c. Он содержит девять процедур, перечисленных в табл. 5.5. Первая из них, get_block, реализует стандартный способ получить блок данных. Когда процедуре файловой системы необходимо прочитать блок пользовательских данных, каталог, суперблок или блок любого другого типа, она вызывает функцию get_block, указывая ей номер блока и устройство.

Таблица 5.5. Процедуры управления блоками

Процедура	Действие
get_block	Извлекает блок для чтения или записи
put_block	Помещает обратно блок, ранее запрошенный с помощью get_block
alloc_zone	Выделяет новую зону (чтобы увеличить файл)
free_zone	Освобождает зону (когда файл удаляется)
rw_block	Перемещает блок между диском и кэшем
invalidate	Чистит все кэшированные блоки с одного из устройств
flushall	Сбрасывает на диск все измененные блоки для некоторого устройства
rw_scattered	Чтение разрозненных данных с устройства или их запись
rm_lru	Удаляет блок из LRU-списка

Когда вызывается функция get_block, она прежде всего смотрит, есть ли запрошенный блок в кэше. Если да, возвращается указатель на него. Иначе запрошенный блок необходимо считать. Блоки в кэше объединены в списки, всего NR_BUF_HASH списков. Этот параметр, NR_BUF_HASH, можно настраивать, как и параметр NR_BUFS, определяющий размер кэша блоков. Оба они устанавливаются в файле include/minix/config.h. В заключение скажем несколько слов об оптимизации размера кэша блоков и хеш-таблицы. Параметр HASH_MASK равен NR_BUF_HASH−1. Если имеется 256 списков, маска равна 255, и все блоки, в номерах которых совпадают последние 8 бит, попадают в один список. Получится 256 списков для номеров 00000000, 00000001, ..., 11111111.

При поиске блока на первом шаге выясняется, в какой из цепочек хеш-таблицы блок находится, хотя есть один особый случай, когда считывается свободное место из разреженного файла и поиск пропускается. Это причина проверки, выполняемой первым условным оператором в коде функции. Если данный особый случай не обнаружен, следующие две строки устанавливают указатель bp на начало той цепочки, в которой оказался бы нужный блок, если бы он был в кэше,

для чего на номер блока накладывается маска HASH_MASK. Следующий далее цикл перебирает элементы цепочки в поисках запрошенного блока. Если блок найден и в данный момент не используется, он исключается из списка LRU. Если он используется, то в LRU-списке его уже нет. Затем вызывающей программе возвращается указатель на найденный блок.

Если в списке нужный блок не обнаружен, то и в кэше его нет, потому из LRU-списка выбирается самый давно использованный блок. Выбранный блок исключается из хеш-цепочки, так как ему будет назначен новый номер, и он попадет в другую цепочку. Если информация в блоке изменена, она записывается на диск. Если это делать при помощи вызова flushall, будут сохранены все измененные блоки с того же устройства. Используемые в текущий момент блоки никогда не могут быть отданы для другого запроса, так как они отсутствуют в LRU-списке.

Как только получен новый буфер, во все его поля, включая поле b_dev, записываются новые значения, и можно считывать данные блока с диска. Есть только два случая, в которых считывать диск с блока необязательно. Функция get_block может быть вызвана с параметром only_search. Это может означать, что выполняется упреждающее выделение блока. Упреждающее выделение подразумевает, что содержимое обнаруженного буфера при необходимости перезаписывается на диск и ему назначается новый номер, но в поле b_dev заносится значение NO_DEV, как свидетельство того, что блок не содержит данных. Применение этого мы увидим, когда станем обсуждать функцию rw_scattered. Кроме того, only_search может использоваться тогда, когда блок нужен файловой системе для полной перезаписи его содержимого. Тогда считывание старых данных было бы пустым расточительством. И в этом и в том случае параметры блока обновляются, но реальное чтение не выполняется. Когда новый блок готов и при необходимости считан, get_block возвращает указатель на него в вызывающую программу.

Предположим, что файловой системе временно, чтобы найти имя файла, нужен блок каталога. Тогда, чтобы получить этот блок, она вызывает get_block. Обнаружив имя файла, она, чтобы вернуть блок в кэш, вызывает put_block, освобождая буфер про запас, в расчете что позднее он понадобится для другого блока.

Функция put_block отвечает за возврат блока в LRU-список, а также в некоторых случаях перезаписывает его содержимое на диск. Второй условный оператор в коде этой функции принимает решение о том, будет ли блок помещен в начало или в конец списка, в зависимости от флага block_type. Блоки, для которых в ближайшем времени ожидаются новые обращения, записываются в конец списка, где они будут храниться еще некоторое время. Блоки, проявление интереса к которым маловероятно, заносятся в начало списка, в расчете на их передачу другим операциям. Например, в начало списка записываются суперблоки.

После того как блок помещен в список, делается другая проверка, чтобы выяснить, нужно ли записать его на диск немедленно. В стандартной конфигурации сразу же сохраняются только суперблоки, но единственный случай, когда суперблок перед записью модифицируется, имеет место во время изменения размера RAM-диска при загрузке. Тогда запись производится на RAM-диск, хотя его суперблок вряд ли понадобится еще раз. Таким образом, данная возможность

мало востребована. С другой стороны, можно так отредактировать макрос ROBUST в файле include/minix/config.h, чтобы немедленно записывались *i*-узлы, блоки каталогов, а также другие блоки, важные для функционирования самой файловой системы.

По мере роста файла необходимо время от времени выделять новые зоны для хранения его данных. За выделение зон отвечает процедура alloc_zone. Она ищет свободную зону по битовой карте зон. Если это первая зона файла, то перебирать всю битовую карту не нужно, так как поле s_zsearch в суперблоке всегда указывает на первую свободную зону на диске. В противном случае, чтобы зоны располагались вместе, ищется ближайшая к последней зоне файла свободная зона. Соответственно, поиск начинается с последней зоны файла. В последней строке процедуры номер бита в битовой карте преобразуется в номер зоны (таким образом, что бит 1 соответствует первой зоне данных).

При удалении файла его зоны следует вернуть в битовую карту. Это действие находится в ведении функции free_zone. Вся ее работа сводится к вызовам функции free_bit, которой передается номер зоны и битовая карта. Функция free_bit применяется и для освобождения *i*-узлов, конечно, тогда ей в качестве первого аргумента передается битовая карта *i*-узлов.

При работе с кэшем требуется записывать и считывать блоки. Простой интерфейс для взаимодействия с диском обеспечивает функция rw_block. Она записывает или считывает один блок. Аналогично, функция rw_inode записывает или считывает один *i*-узел.

Следующая процедура называется invalidate. Она вызывается, например, при демонтировании диска, чтобы убрать из кэша все блоки, принадлежащие демонтируемой файловой системе. Если этого не сделать, при следующем использовании устройства (с другим гибким диском) система может увидеть старые блоки вместо новых.

Функцию flushall вызывает системный вызов sync, для того чтобы сбросить на диск все измененные буферы, принадлежащие какому-то одному монтированному устройству. Эта функция работает с кэшем буферов как с одномерным массивом, поэтому она может обнаруживать измененные буферы даже в том случае, если их нет в списке LRU. Весь массив буферов сканируется, и указатели на те из них, что принадлежат указанному устройству и содержат обновленные данные, добавляются в массив указателей dirty. Последний массив, чтобы не выделять его в стеке, объявлен как static. Затем массив передается rw_scattered.

Функция rw_scattered получает на входе идентификатор устройства, указатель на массив указателей на буферы, размер этого массива и флаг, указывающий, выполняется чтение или запись. Первым делом функция сортирует переданный ей массив указателей, чтобы передача данных происходила в наиболее эффективном порядке. С флагом WRITING ее может вызывать только описанная выше процедура flushall. В этом случае несложно понять происхождение номеров блоков, так как передаваемые буферы содержат данные, которые были ранее считаны, а теперь изменены. Для чтения rw_scattered вызывается только из функции rahead в файле read.c. Сейчас мы не будем углубляться в детали ее работы, скажем лишь, что перед вызовом rw_scattered несколько раз в режиме подготовки

вызывается get_block, тем самым резервируется группа буферов. Буферам назначается номер блока, а номер устройства — нет, но это не проблема, поскольку номер устройства передается в качестве одного из аргументов в rw_scattered.

Есть важное отличие между тем, как драйвер устройства реагирует на чтение и запись из rw_scattered. Запрос на запись некоторого количества блоков *обязательно* должен быть выполнен полностью, в то время как запрос на чтение обрабатывается разными драйверами по-разному, в зависимости от того, что более эффективно для данного конкретного драйвера. Функция rahead зачастую вызывает rw_scattered, передавая ей список буферов, которые в действительности могут быть не нужны. Поэтому лучше всего считать только те блоки, до которых легко добраться, не выискивая их по всему устройству и теряя драгоценное время. Например, драйвер гибкого диска может остановиться на границе дорожки, а другие драйверы будут читать только последовательные блоки. Выполнив чтение, rw_scattered помечает прочитанные блоки, вписывая в содержащие их буферы номер устройства.

Последняя функция из табл. 5.5, rm_lru, предназначена для того, чтобы удалять блок из LRU-списка. Она используется только внутри функции get_block, поэтому вместо PUBLIC объявлена как PRIVATE, с целью скрыть ее от процедур в других файлах.

Прежде чем закончить изучение кэша блоков, скажем несколько слов о его тонкой настройке. Значение параметра NR_BUF_HASH должно быть степенью двойки. Если оно больше, чем NR_BUFS, средняя длина хеш-цепочки будет меньше единицы. Но когда хватает памяти для большого количества буферов, ее хватит и для большого количества хеш-цепочек, поэтому значение этого параметра обычно выбирается равным ближайшей степени двойки, большей NR_BUFS. В обсуждаемом в тексте исходном коде заведено 512 буферов и 1024 хеш-цепочки. Оптимальный размер зависит от характера эксплуатации системы, поскольку определяет, сколько информации будет кэшироваться. Опытным путем было установлено, что увеличение числа буферов выше 1024 не дает прироста производительности при перекомпиляции MINIX, видимо, потому, что этого достаточно, чтобы хранить промежуточные файлы для всех проходов компилятора. Для некоторых задач более адекватной будет меньшая емкость, а другим может потребоваться увеличить кэш ради подъема быстродействия.

Скомпилированные двоичные файлы установочного пакета MINIX настроены на работу с кэшем гораздо меньшего объема, так как эта конфигурация рассчитана на работу на как можно большем количестве разных компьютеров. Требовалось создать систему, устанавливаемую в том числе на машину с всего двумя мегабайтами оперативной памяти, а система с кэшем на 1024 блока требует более 2 Мбайт памяти. Пакет также включает в себя все возможные драйверы жестких дисков и другие специфические драйверы. Мы полагаем, что большинству пользователей вскоре после установки системы наверняка захочется отредактировать include/minix/config.h, чтобы убрать ненужные драйверы и увеличить размер кэша, и перекомпилировать систему.

Говоря о кэше блоков, следует заметить, что на 16-разрядных машинах Intel размер сегмента памяти ограничен 64 Кбайт, что делает невозможной реализа-

цию большого кэша. В этом случае можно так настроить файловую систему, чтобы RAM-диск выступал в роли дополнительного кэша, хранящего блоки, изгнанные из основного. Мы не будем обсуждать здесь эту возможность, так как на 32-битных системах Intel такой необходимости нет, большой основной кэш дает наилучшую производительность. Тем не менее вспомогательный кэш, вероятно, окажется полезен на компьютере, где не хватает места для большого основного кэша в виртуальном адресном пространстве файловой системы (например, это может быть 286-я модель). Дополнительный кэш будет работать лучше, чем обычный RAM-диск, в силу того что кэш хранит только данные, которые хотя бы раз действительно потребовались, и достаточно вместительный кэш может значительно увеличить производительность. Какой объем должен иметь «достаточно большой» кэш, заранее сказать нельзя. Ответ дает экспериментирование с тем, приводит ли дальнейшее наращивание кэша к повышению производительности. Полезным инструментом для оптимизации системы является команда time, которая позволяет измерить время выполнения определенной программы.

Работа с i-узлами

Не только кэшу блоков требуются управляющие подпрограммы. Для работы с таблицей *i*-узлов они также необходимы. Многие из этих процедур аналогичны процедурам для работы с блоками. Сами процедуры перечислены в табл. 5.6.

Таблица 5.6. Процедуры для работы с i-узлами

Процедура	Назначение
get_inode	Помещает i-узел в память
put_inode	Возвращает более не нужный i-узел
alloc_inode	Выделяет новый i-узел (для нового файла)
wipe_inode	Очищает некоторые поля i-узла
free_inode	Освобождает i-узел (при удалении файла)
update_times	Обновляет поля времени в i-узле
rw_inode	Передает i-узел между памятью и диском
old_icopy	Преобразует данные i-узла, чтобы записать его на диск в формате V1
new_icopy	Преобразует данные i-узла, чтобы записать его на диск в формате V2
dup_inode	Указывает, что кто-то еще использует i-узел

Функция get_inode является аналогом get_block. Когда какой-либо части системы требуется получить *i*-узел, она вызывает эту функцию. Сначала get_inode ищет узел в таблице, чтобы узнать, не загружен ли он уже. Если да, счетчик использования *i*-узла увеличивается и возвращается указатель на него. Поиск производится большим циклом в коде функции. Если в памяти *i*-узел не найден, он загружается при помощи вызова rw_inode.

Когда процедура, которой потребовался *i*-узел, завершает свои действия с ним, то, чтобы возвратить *i*-узел, она вызывает процедуру put_inode, уменьшающую значение счетчика использования. Нулевое значение счетчика подразумевает,

что файл больше никому не нужен и может быть удален из таблицы. Если при этом *i*-узел оказался изменен, он перезаписывается на диск.

Если поле i_link равно нулю, значит, на файл не ссылается ни один из каталогов, поэтому все его зоны могут быть освобождены. Обратите внимание, что обнуление счетчика ссылок i_link и счетчика использования — разные события, они обусловлены разными причинами и приводят к разным следствиям. Если *i*-узел соответствует каналу ввода/вывода, все его зоны должны быть освобождены, даже если число ссылок не равно нулю. Такое может произойти, когда процесс, читающий из канала, освободит его. Нет никакого смысла поддерживать канал для одного процесса.

Когда создается файл, для него должен быть выделен новый *i*-узел при помощи процедуры alloc_inode. В MINIX устройства можно монтировать в режиме только для чтения, поэтому функция сначала смотрит в суперблоке, разрешена ли запись на устройство. В отличие от зон, которые выбираются в стремлении держать все зоны файла близко друг к другу, здесь подойдет любой *i*-узел. Чтобы уменьшить время поиска в битовой карте *i*-узлов, используются поля суперблока, в которых хранится положение первого свободного *i*-узла.

После того как новый *i*-узел получен, он загружается в таблицу в памяти при помощи get_inode. Затем, частично прямо на месте, а частично — при помощи процедуры wipe_inode, его поля инициализируются. Такое разделение труда выбрано здесь потому, что wipe_inode вызывается и в некоторых других частях системы для очистки отдельных полей *i*-узла.

При удалении файла его *i*-узел освобождается посредством процедуры free_inode. Эта подпрограмма просто сбрасывает соответствующий бит в карте *i*-узлов и обновляет ссылку на первый свободный *i*-узел в суперблоке.

Следующая функция, update_times, вызывается, чтобы получить от системных часов время и изменить нуждающиеся в обновлении значения полей времени. Она также вызывается из stat и fstat, поэтому объявлена как PUBLIC.

Процедура rw_inode аналогична процедуре rw_block. Ее назначение в том, чтобы считать *i*-узел с диска. Это действие она выполняет в четыре следующих этапа:

1. Определение блока, в котором находится необходимый *i*-узел.
2. Считывание блока при помощи get_block.
3. Извлечение *i*-узла и копирование его в таблицу inode.
4. Возврат блока посредством put_block.

Код rw_inode несколько сложнее, чем приведенная выше схема, так как от него требуются некоторые дополнительные действия. Во-первых, поскольку определение текущего времени является весьма дорогостоящей операцией, все поля *i*-узла, подлежащие обновлению значения времени, только помечаются путем установки соответствующих битов в поле *i*-узла в памяти i_update. Если это поле будет иметь ненулевое значение, при записи будет вызвана функция update_times.

Во-вторых, дополнительную сложность вносит история MINIX. В старой версии файловой системы V1 *i*-узлы на диске имели структуру, отличную от структуры в новой версии V2. Поэтому о преобразовании заботятся две функции,

old_copy и new_copy. Первая из них преобразует представление *i*-узла в памяти в формат старой версии V1. Вторая делает то же самое для новой версии V2. Обе функции используются только в пределах этого файла, поэтому они объявлены как PRIVATE. Обе они выполняют преобразование в двух направлениях, как из памяти на диск, так и обратно. ОС MINIX работает и на системах, где расположение байтов в слове отличается от такового в процессорах Intel. В каждой реализации информация на диске хранится в том порядке, какой принят в системе. А что это за порядок, система узнает из поля sp->native в суперблоке. Поэтому old_copy и new_copy при необходимости вызывают conv2 и conv4, с целью изменить порядок байтов.

Процедура dup_inode просто увеличивает на единицу счетчик использования *i*-узла. Она вызывается при повторном открытии файла, при этом *i*-узел не нужно еще раз считывать с диска.

Работа с суперблоками

В файле super.c находятся процедуры для работы с суперблоками и битовыми картами. Эти пять процедур перечислены в табл. 5.7.

Таблица 5.7. Процедуры для работы с суперблоками и битовыми картами

Процедура	Назначение
alloc_bit	«Выделяет» бит в битовой карте i-узлов или зон
free_bit	«Освобождает» бит в битовой карте i-узлов или зон
get_super	Ищет устройство в таблице суперблоков
mounted	Сообщает, принадлежит ли i-узел монтированной или корневой файловой системе
read_super	Считывает суперблок

Как мы видели выше, когда требуется *i*-узел или зона, вызывается функция alloc_inode или alloc_zone. Каждая из них, в свою очередь, вызывают alloc_bit, чтобы найти неустановленный бит в битовой карте. Поиск включает в себя три вложенных цикла, работающих следующим образом:

1. Внешний цикл перебирает все блоки битовой карты.
2. Промежуточный цикл перебирает все слова блока.
3. Внутренний цикл проверяет все биты в слове.

В промежуточном цикле выясняется, является ли текущее слово дополнением нуля, то есть состоит ли оно целиком из единиц. Если да, значит, в этом слове нет «свободных» битов, и проверяется следующее слово. Когда обнаруживается другое значение, где по крайней мере один бит равен 0, запускается внутренний цикл, который определяет его позицию в слове. Если оказалось, что проверены все блоки, но нулевых битов не нашлось, значит, свободных *i*-узлов или зон на диске нет, и возвращается код NO_BIT (0). Подобный поиск может потребовать много процессорного времени, но благодаря указателям на первый свободный

i-узел и зону, передаваемых из суперблока в alloc_bit как начальная позиция для поиска, его удается подсократить.

Обнулить «занятый» бит проще, чем установить, так как не требуется выполнять поиск. Функция free_bit вычисляет, какой блок битовой карты содержит сбрасываемый бит, и дальше вызывает get_block, обнуляет бит и завершает операцию вызовом put_block.

При помощи следующей процедуры, get_super, в таблице суперблоков ищется запись заданного устройства. Например, когда монтируется файловая система, нужно проверить, не делается ли это повторно. Для чего можно через посредство get_super попробовать найти устройство, на котором файловая система расположена. Если устройство не найдено, то и файловая система еще не смонтирована.

Функция mounted вызывается только при закрытии блочного устройства. Обычно при закрытии такого устройства все данные, сохраненные в кэше, сбрасываются. Но если оказалось, что устройство смонтировано, это нежелательно. Функции mounted передается указатель на *i*-узел устройства. Она проверяет, является ли устройство корневым или смонтированным, и если да, возвращает TRUE.

Наконец, перейдем к read_super. Эта функция частично подобна функциям rw_block и rw_inode, но выполняет только чтение. Записывать суперблок при нормальной работе системы не требуется. Считав суперблок, функция read_super узнает версию файловой системы и при необходимости выполняет форматное преобразование. Таким образом, копия суперблока в памяти всегда имеет стандартный формат, даже если она прочитана с диска с другой структурой.

Работа с дескрипторами файлов

В MINIX имеются специальные подпрограммы и для работы с файловыми дескрипторами и таблицами filp (см. рис. 5.29). Эти процедуры находятся в файле filedesc.c. Когда файл создается или открывается, с ним связываются свободный дескриптор и незанятая ячейка в таблице filp. Для поиска свободных ресурсов предназначена процедура get_fd. Она не помечает их как занятые, так как успешному завершению вызовов creat или open предшествует еще множество различных проверок.

Функция get_filp проверяет, находится ли файловый дескриптор в заданных пределах и возвращает его указатель, взятый из filp.

Последняя процедура в файле, find_filp, позволяет выяснить, что процесс пытается писать в неработающий канал (то есть канал, не открытый другим процессом на чтение). Она прямым перебором ищет в таблице filp процессы-читатели на другом конце канала. Если такой процесс найти не удалось, значит, канал разрушен и попытка записи в него обречена на провал.

Захват файлов

Функции POSIX для захвата файлов перечислены в табл. 5.8. Участок файла можно захватить на запись и чтение или только на запись при помощи системного вызова fcntl с запросом F_SETLK или F_SETLKW. Узнать, удерживается ли кем тот или иной участок файла, можно с помощью запроса F_GETLK.

Таблица 5.8. Операции захвата записей согласно POSIX

Операция	Действие
F_SETLK	Захватывает область на запись и чтение
F_SETLKW	Захватывает область на запись
F_GETLK	Проверяет, свободна ли область

В lock.c есть только две функции. К первой из них, lock_op, обращается системный вызов fcntl при выполнении каждой из операций, перечисленных в таблице. Она делает несколько тестов, гарантирующих, что участок задан корректно. Блокировка участка не должна конфликтовать с существующими, а другой захват не должен быть выполнен дважды. Для снятия захвата вызывается другая процедура из этого файла, lock_revive. Она разблокирует все процессы, ранее заблокированные в ожидании данного участка. Это компромиссная стратегия, потому для точного решения, какой процесс получит управление, потребовался дополнительный код. Те процессы, время которых еще не наступило, вновь блокируются после запуска. Такая стратегия выбрана исходя из того предположения, что захват файлов — операция редкая. Если на основе MINIX будет построена многопользовательская база данных, не исключено, что потребуется изменить данный алгоритм.

Процедура lock_revive также вызывается при закрытии захваченного файла, это может произойти, например, если процесс был завершен принудительно до того, как закончил работать с удерживаемым им файлом.

5.7.3. Основная программа

Код главного цикла файловой системы находится в файле main.c. По своей структуре он очень схож с главным циклом менеджера памяти и задач ввода/вывода. В нем вызывается функция get_work, ожидающая прибытия следующего запроса на обслуживание (если не может быть обслужен процесс, ранее приостановленный на чтении из канала или с терминала). Она же устанавливает значение глобальной переменной who, куда записывается номер ячейки в таблице процессов, принадлежащей вызывающему процессу, а также заполняет другую глобальную переменную, fs_call, записывая в нее номер сделанного системного вызова.

Как только управление возвращается в главный цикл, устанавливаются три флага: fp указывает на запись вызывающего процесса в таблице процессов, super_user говорит, принадлежит ли этот процесс суперпользователю, dont_reply инициализируется значением FALSE. Затем начинается главное действо: на передний план выходит процедура, выполняющая системный вызов. Адрес процедуры определяется по номеру системного вызова из таблицы указателей на процедуры, call_vector.

При возврате управления в главный цикл проверяется значение флага dont_reply. Если он установлен, значит, ответное сообщение не требуется (то есть процесс заблокирован по причине чтения из пустого канала). Иначе отправляется ответное сообщение. Последний оператор в цикле должен распознавать последо-

вательное чтение файла и, соответственно, пытается загрузить следующий блок до того, как в действительности придет запрос, в расчете на увеличение производительности.

Вызов get_work смотрит, имеются ли ранее заблокированные процедуры, которые могут продолжить работу. Если такие есть, они считаются приоритетнее новых сообщений. Только тогда, когда у файловой системы нет отложенных задач, она отправляет ядру запрос на получение следующего сообщения.

После того как системный вызов, успешно или с ошибкой, выполнен, при помощи reply отправляется ответ. Процесс может быть завершен по сигналу, и код состояния, возвращаемый ядром, игнорируется. В этом случае все равно ничего больше не сделаешь.

Функции инициализации

Оставшуюся часть файла main.c составляют функции инициализации, отрабатывающие при запуске системы. Перед входом в главный цикл файловая система инициализирует себя, вызывая fs_init, которая, в свою очередь, вызывает несколько других функций с целью подготовить кэш блоков, определить параметры загрузки ядра, при необходимости организовать RAM-диск, а также загрузить суперблок корневого устройства. На следующем шаге заполняется принадлежащая файловой системе часть таблицы процессов, в нее вносятся записи всех серверов и задач, до процесса init. Наконец, проверяются значения некоторых важных констант, чтобы узнать, осмысленны ли их значения, и задаче памяти отправляется сообщение, в котором передается адрес таблицы процессов в файловой системе. Впоследствии этот адрес может использоваться программой ps.

Сначала fs_init вызывает buf_pool, формирующую списки для кэширования блоков. На рис. 5.27 показано нормальное состояние кэша, где все блоки связаны друг с другом в LRU-список и хешированы. Полезно разобраться в том, откуда берется ситуация, показанная на рисунке. Сразу после инициализации процедурой buf_pool все буферы находятся в списке и все они сцеплены в нулевую хеш-цепочку, как показано на рис. 5.30, *а*. Когда запрашивается буфер, структура приходит в состояние, показанное на рис. 5.30, *б*, и остается в нем, пока буфер используется. На этом рисунке мы можем видеть, что блок исключен из LRU-списка и помещен в отдельную хеш-цепочку. Обычно же блок освобождается и возвращается в LRU-список немедленно. Эта ситуация иллюстрируется рис. 5.30, *в*. Здесь блок, хотя и не используется более, все еще содержит информацию, и при необходимости может быть извлечен из хеш-цепочки. После того как система поработает некоторое время, почти все блоки, скорее всего, окажутся случайно распределены между различными цепочками. Список LRU при этом будет выглядеть в соответствии с рис. 5.27.

Следующая функция, get_boot_parameters, запрашивает у задачи системы параметры загрузки ядра. Эти значения необходимы функции load_ram, выделяющей память для RAM-диска. Если при загрузке ядру передан параметр

```
rootdev = ram
```

корневая файловая система блок за блоком копируется на RAM-диск (устройство ramimagedev), различные структуры файловой системы не интерпретируются.

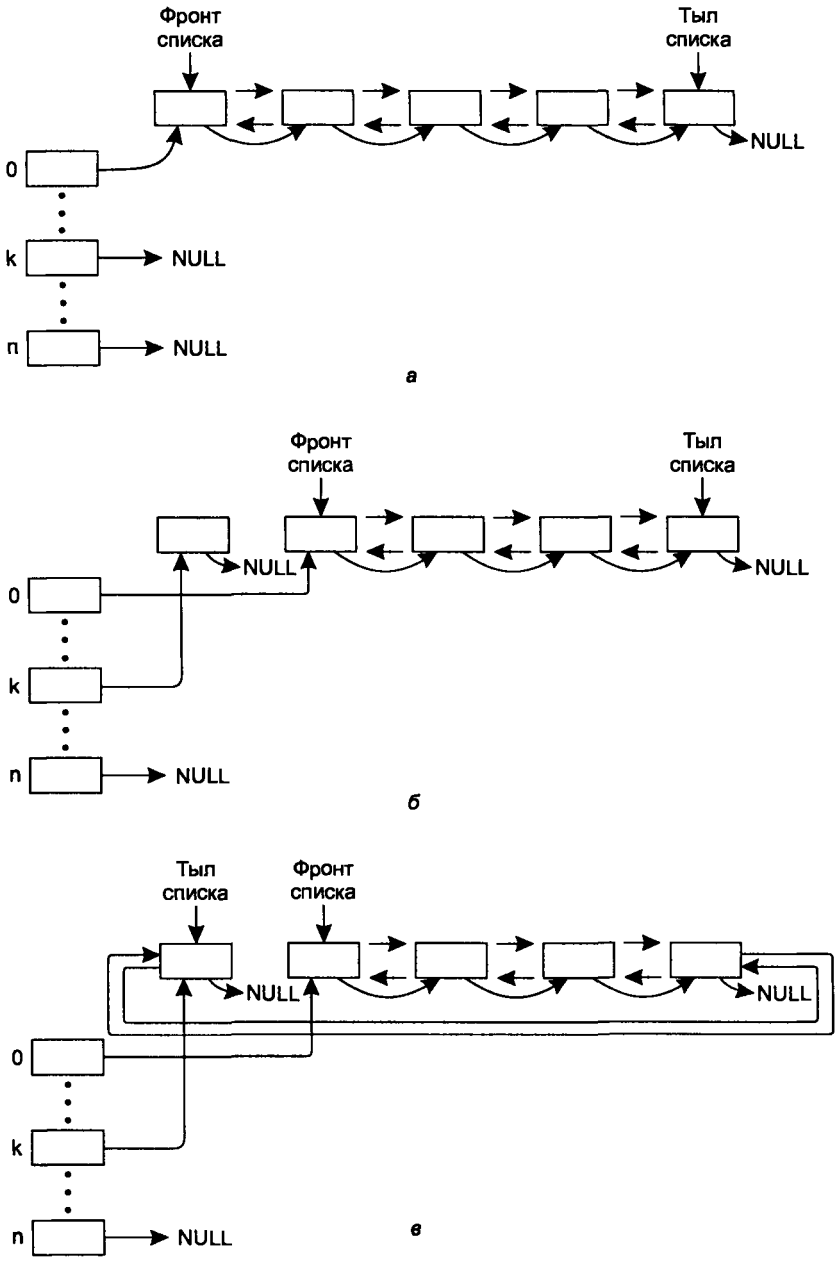


Рис. 5.30. Инициализация кэша блоков: *а* — начальное состояние; *б* — после запроса одного блока; *в* — после того как блок освобожден

Если значение параметра ramsize меньше объема корневой файловой системы, RAM-диск увеличивается, чтобы вместить ее. Если же RAM-диск вмещает всю

корневую файловую систему и на нем еще остается место, производится подстройка с целью подогнать размер диска под объем файловой системы. Чтобы записать новую информацию в суперблок, вызывается функция put_block. Это единственный случай, когда суперблок модифицируется.

Функция load_ram выделяет память для RAM-диска, когда параметр ramsize равен нулю. При этом на диск не копируется никаких структур файловой системы, и прежде чем работать с диском, их нужно создать при помощи команды mkfs. Пустой же RAM-диск можно использовать в качестве вторичного кэша, если поддержка такового включена при компиляции системы.

Последняя функция в main.c — load_super — выполняет инициализацию таблицы суперблоков и считывает суперблок корневого устройства.

5.7.4. Работа с отдельными файлами

В этом разделе мы рассмотрим системные вызовы по отношению к отдельным файлам (в противоположность тем, которые оперируют, скажем, каталогами). Начнем же с того, как файлы создаются, открываются и закрываются. Затем мы более подробно изучим механизм записи и чтения файлов. И на десерт у нас будут процедуры, работающие с каналами, и понимание того, чем их работа отличается от работы с файлами.

Создание, открытие и закрытие файлов

Файл open.c содержит код шести системных вызовов: creat, open, mknod, mkdir, close и lseek. Вызовы creat и open мы рассмотрим вместе, а затем перейдем к остальным.

В более старых версиях UNIX вызовы creat и open служили для разных целей. При попытке открыть несуществующий файл возникала ошибка, а новый файл должен был создаваться при помощи creat, другой задачей которого является установка размера файла в нулевое значение. Но в современной POSIX-системе два отдельных вызова не требуются. По POSIX, созданием файла или «обнулением» старого должен заниматься вызов open, поэтому возможности creat теперь составляют лишь часть потенциала open, и нужен он только для совместимости со старыми программами. Процедуры, выполняющие действия creat и open, называются соответственно do_creat и do_open (как и в случае с менеджером памяти, здесь соблюдается соглашение именовать обработчик системного вызова xxx как do_xxx). Открытие или создание файла включает три шага.

1. Поиск *i*-узла (или выделение и инициализация нового *i*-узла, если файл создается).
2. Поиск или создание записи в каталоге.
3. Настройка дескриптора файла и его возврат.

Оба вызова, creat и open, делают лишь то, что принимают имя файла и перекладывают задачи, общие для обоих вызовов, на плечи common_open.

Процедура common_open начинает с того, что проверяет наличие свободных файловых дескрипторов и ячеек в таблице filp. Затем, если при вызове было ука-

зано создать новый файл (то есть установлен бит O_CREAT), вызывается функция new_node. Если запись в каталоге уже присутствует, эта функция возвращает указатель на существующий *i*-узел, в противном случае она создает новую запись и *i*-узел. Если *i*-узел не может быть создан, функция записывает код ошибки в глобальную переменную err_code. Наличие кода ошибки не обязательно означает сбой. Когда new_node обнаруживает одноименный файл, она сигнализирует об этом через код ошибки, но в действительности здесь нет ничего противоправного. При неустановленном бите O_CREAT поиск *i*-узла осуществляется альтернативным методом, через посредство функции eat_pat из состава файла path.c, которую мы обсудим позже. На данном этапе важно лишь знать, что если *i*-узел не удалось ни найти, ни создать, работа common_open завершается с ошибкой прежде, чем начинается выделение файлового дескриптора. Иначе функция продолжает работу, назначая файлу дескриптор и запрашивая запись в таблице filp. Следующий блок кода пропускается, если файл создается именно сейчас.

Если же файл существовал ранее, файловая система должна определить, что это за файл, какие разрешения выставлены у него и т. д., чтобы выяснить, может ли он быть открыт. Проверку битов rwx (то есть битов, управляющих доступом) выполняет вызов forbidden. Если файл представляет собой обычный файл и при вызове был установлен бит O_TRUNC, длина файла приравнивается нулю и снова вызывается forbidden, на этот раз чтобы проверить доступность файла на запись. Когда запись разрешена, вызываются функции wipe_inode и rw_inode, очищающие *i*-узел и сохраняющие его на диске. Для других типов файлов (каталогов, специальных файлов и каналов ввода/вывода) делаются соответствующие проверки. Так, в случае устройства, при помощи структуры dmap выполняется вызов нужной процедуры, которая открывает его. За открытие именованного канала отвечает функция pipe_open, для него выполняются различные проверки, имеющие отношение к каналам.

В коде функции common_open, как и в тексте других процедур файловой системы, есть большое количество проверок различных ошибок и недопустимых комбинаций параметров. Хотя этот код и не изящен, он важен для получения устойчивой, свободной от врожденных дефектов файловой системы. Если произошла какая-либо ошибка, ранее выделенные файловый дескриптор и ячейка в таблице filp освобождаются. Функция common_open в этом случае возвращает отрицательное значение, говорящее об ошибке. Если проблем с файловым дескриптором не было, код возврата является положительным числом.

Отложим в сторону файлы и займемся каталогами. И для начала поглядим на работу функции new_node, что занимается выделением *i*-узла для вызовов creat и open. Обратившись к ней, мы вводим в рассмотрение путь к файлу. Она также используется системными вызовами mknod и mkdir, которые еще будут обсуждаться. Сначала эта функция разбирает путь к файлу, то есть просматривает его компонент за компонентом, пока не достигает конечного каталога. При помощи функции advance проверяется, может ли быть открыт последний компонент.

Например, если сделан вызов

```
fd = creat("/usr/ast/foobar", 0755);
```

то load_dir попытается загрузить в таблицы *i*-узел для /usr/ast и вернуть указатель на него. Если файл foobar еще не существует, этот *i*-узел скоро понадобится, чтобы добавить новый файл в каталог. Другие системные вызовы, добавляющие или удаляющие файлы, также используют load_dir.

Если new_node обнаруживает, что файл не существует, она вызывает alloc_inode, тем самым выделяя новый *i*-узел, и возвращает указатель на него. Если свободных *i*-узлов не осталось, назад вернется код NIL_INODE.

Если выделить *i*-узел удалось, его поля заполняются и он записывается обратно на диск, а в конечный каталог заносится запись с именем файла. Опять же, здесь мы видим, что файловая система обязана постоянно проверять, не произошло ли ошибки. Если ошибка происходит, необходимо без паники аккуратно освободить все ресурсы, такие как *i*-узел и удерживаемый блок. Если вместо того, чтобы, скажем, при нехватке *i*-узлов мы позволили бы MINIX просто завершиться с фатальной ошибкой вместо того, чтобы тщательно отменять все действия вызова и возвращать код ошибки, файловая система была бы ощутимо проще, но и, сами понимаете...

Ранее упоминалось, что каналы ввода/вывода требуют особых действий. Когда с каналом не связан хотя бы один процесс, читатель либо писатель, функция pipe_open приостанавливает процесс-инициатор вызова. В противном случае она вызывает процедуру release, которая ищет в таблице процессов те, что ожидают данный канал. Если такие процессы найдены, они запускаются.

Системный вызов mknod выполняется функцией do_mknod. Эта процедура аналогична do_creat, за исключением того, что она лишь создает *i*-узел и делает для него запись каталога. Фактически большую часть работы выполняет функция new_node (за три строки перед завершающим оператором return). Если *i*-узел уже существует, возвращается код ошибки. Код ошибки тот же самый, который был приемлемым при открытии файла, но в данном случае код возвращается в вызывающую программу, которая, предположительно, подобающим образом его интерпретирует. Подробный анализ отдельных случаев, который мы видели в common_open, здесь не нужен.

Функция do_mkdir выполняет системный вызов mkdir. Как и в случае с предыдущими системными вызовами, важную роль здесь играет new_node. Каталоги в отличие от файлов никогда не бывают пусты, так как любой каталог содержит, по крайней мере, две записи, «.» и «..», первая из которых ссылается на сам каталог, а вторая — на вышестоящий. Количество ссылок на один файл ограничено константой LINK_MAX (в файле include/limits.h для стандартной конфигурации MINIX ей присваивается значение 127). Поэтому, раз дочерний каталог содержит ссылку на родительский, функция do_mkdir сначала проверяет, можно ли добавить новую ссылку на родительский каталог. Когда эта проверка пройдена, вызывается new_node. Если и этот вызов удался, создаются записи для каталогов «.» и «..». Описанный алгоритм прямолинеен, но учитывает возможность сбоев (например, переполнение диска). Чтобы ничего не испортить, обеспечивается откат к исходному состоянию, если процесс не может быть завершен.

Закрыть файл всегда проще, чем открыть. Поэтому функции do_close для обычных файлов, фактически, требуется только уменьшить значение счетчика в filp

и, если оно достигло нуля, возвратить *i*-узел при помощи put_node. (Каналам же и специальным файлам требуется особое внимание.) На последнем шаге аннулируются все захваты, связанные с этим файлом, и пробуждаются все процессы, приостановленные по факту захвата.

Заметьте, что возврат *i*-узла означает только, что уменьшается счетчик в таблице inode, следовательно, в конце концов, он может быть удален из таблицы. Эта операция не имеет ничего общего с освобождением *i*-узла (то есть со сбросом бита в битовой карте *i*-узлов). Объект *i*-узла освобождается только тогда, когда файл больше не находится ни в одном из каталогов.

Последняя процедура в open.c, do_seek, вызывается, когда осуществляется переход на заданную позицию в файле. При этом блокируется упреждающее чтение, поскольку явное перемещение на заданную позицию в файле несовместимо с последовательным доступом.

Чтение файла

Открыв файл, его можно читать или записывать в него данные. Таких функций больше, чем достаточно, и все связанные с чтением можно найти в файле read.c. Мы обсудим сначала их, а затем перейдем к следующему файлу, write.c, чтобы взглянуть на код, предназначенный специально для записи. Чтение и запись во многом различны, но у них довольно много общего, поэтому все, что требуется от do_read, — это вызвать общую процедуру read_write с флагом READING. В следующем разделе мы увидим, что do_read столь же проста.

Первый условный оператор связан с обработкой особых действий, при помощи которых файловая система загружает целые сегменты для менеджера памяти. Обычная работа начинается с альтернативной ветви условного оператора. Затем следуют несколько проверок корректности операции (скажем, не делается ли попытка читать из файла, открытого только на запись) и инициализируются некоторые глобальные переменные. Чтение из специальных символьных файлов производится в обход кэша блоков, поэтому такие файлы фильтруются отдельным условным оператором.

Если файл обычный, выполняются следующие проверки, имеющие отношение только к записи в файлы, где есть опасность превышения емкости устройства, а также относительно записи *после* конца файла, что может привести к образованию «дыр». Как уже упоминалось в обзоре MINIX, наличие нескольких блоков в зоне приводит к некоторым проблемам, с которыми приходится бороться. Каналы ввода/вывода также являются особым случаем.

Следующий далее по коду цикл является сердцем механизма чтения, по крайней мере, для обычных файлов. В цикле данные разбиваются на куски, каждый из которых умещается в один дисковый блок. Порция начинается с текущей позиции и считается завершенной при выполнении одного из трех условий:

- считаны все байты;
- встретилась граница блока;
- достигнут конец файла.

Эти правила означают, что одна порция никогда не может занимать два блока. На рис. 5.31 показано, как определяется квота размера, для порций размером шесть, два и один байт соответственно.

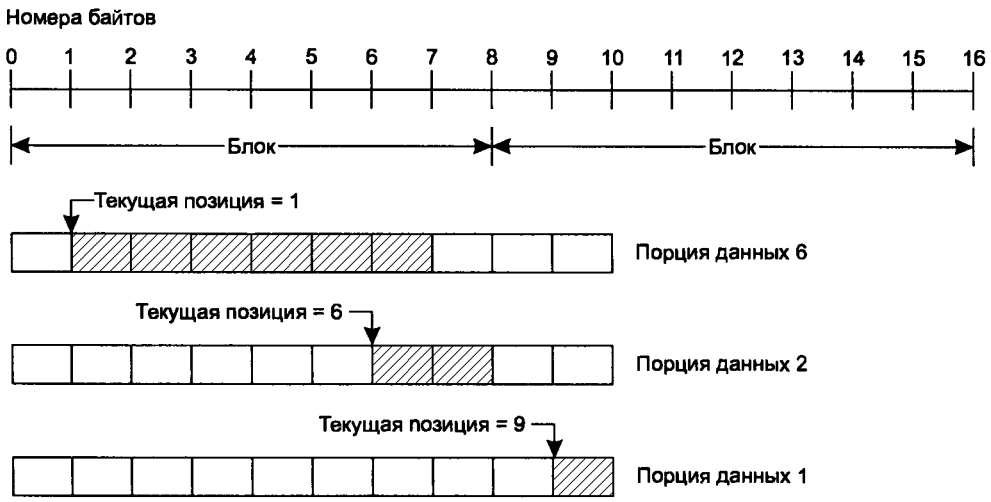


Рис. 5.31. Три примера того, как для 10-байтового файла определяется размер первой порции данных. Размер блока равен 8 байтам, запрашивается 6 байт. Порция отмечена штриховкой

Считывание выполняется функцией rw_chunk. Когда эта функция возвращает управление, увеличиваются значения различных счетчиков и продвигаются указатели, и начинается следующая итерация. После того как цикл завершится, текущая позиция в файле и другие переменные (например, указатели канала) могут быть обновлены.

Наконец, если было запрошено упреждающее чтение, позиция и *i*-узел, откуда его нужно начинать, сохраняются в глобальных переменных, чтобы файловая система, отправив пользователю ответное сообщение, могла бы начать считывать следующий блок. Часто при этом файловая система блокируется, ожидая считывания блока, и у пользовательского процесса есть время начать работать с только что полученными данными. Такое чередование обработки и ввода/вывода может значительно улучшить производительность.

Процедура rw_chunk получает на входе *i*-узел и позицию в файле, преобразует эти значения в физический номер блока на диске и запрашивает передачу этого блока (или его части) в область данных пользователя. Преобразование относительной позиции в файле в физический адрес на диске выполняется функцией read_map, которая осведомлена о структуре *i*-узлов и блоков косвенной адресации. Первый условный оператор функции read_map проверяет, является ли файл обычным, и если да, в переменные b и dev записываются соответственно физический номер блока и номер устройства. Далее, в 260-й строке вызывается функция

get_block, и обработчик кэша блоков находит нужный блок, считывая его при необходимости.

О копировании данных в адресное пространство пользователя из найденного блока заботится функция sys_copy на 270-й строке. Затем блок освобождается, чтобы позже он мог быть удален из кэша. (После того как get_block находит нужный блок, тот исключается из списка LRU и пребывает вне его до тех пор, пока счетчик использования в заголовке буфера не обнулится. Вызов put_block уменьшает значение этого счетчика и, если настало время, возвращает буфер в LRU-список). Код в 280-й строке файла определяет, заполнился ли блок при записи. Правда, сейчас это уже не принципиально, так как функция put_block теперь игнорирует значение, передаваемое ей во втором аргументе, и всегда добавляет освободившиеся блоки в конец LRU-последовательности.

Функция read_map преобразует логическое смещение в файле в физический номер блока, пользуясь для этого информацией *i*-узла. Те блоки, которые достаточно близки к началу файла, попадут в одну из первых семи зон (то есть в одну из тех зон, что хранятся в самом *i*-узле). При этом, чтобы узнать, какая из зон необходима, требуется только несложная арифметика. Для блоков, расположенных дальше, может понадобиться считать один или два блока косвенной адресации.

Функция rd_indir вызывается, когда необходимо считать «косвенный» блок. Это действие вынесено в самостоятельную процедуру потому, что данные на диске могут храниться в разных форматах, в зависимости от версии файловой системы и от оборудования, на котором файловая система была сформирована. Функция выполняет необходимые преобразования, чтобы все остальные подпрограммы видели данные только в одной форме.

Функция read_ahead преобразует логическое положение в физический адрес блока, вызывает get_block, в результате чего блок оказывается в кэше, и немедленно возвращает его. В конце концов, сделать с ним она все равно ничего не может. Задача read_ahead лишь в том, чтобы увеличить вероятность найти данные в кэше, если они скоро понадобятся.

Обратите внимание, что read_ahead вызывается только из главного цикла в main. Ее вызов не является частью выполнения read. Важно понять, что эта функция вызывается *после* того, как пользователю отправлен ответ, чтобы он мог продолжать свою работу, пока файловая система дожидается завершения упреждающего чтения.

Сама функция read_ahead написана так, чтобы запрашивать всего один блок. Реально трудится вызываемая ею подпрограмма rahead, последняя в файле read.c. В rahead заложена та жизненная концепция, что если немного больше — хорошо, то намного больше — еще лучше. Так как дискам и прочим накопителям часто требуется много времени, чтобы найти первый запрошенный блок, но зато они могут быстро считывать несколько смежных блоков, делается ставка на то, что получится считать много последовательных блоков ценой небольших дополнительных трудозатрат. Запрос на упреждающую выборку передается функции get_block, подготавливающей кэш блоков к получению нескольких блоков за раз. Затем вызывается rw_scattered, которой передается список блоков. Работу этой функции мы уже обсуждали. Вспомните, что rw_scattered передает запрос драй-

веру устройства, который в ответ имеет право обслужить столько запросов, сколько он способен выполнить эффективно. Все это звучит довольно витиевато, но зато позволяет на нужное «много» увеличить скорость приложений, считывающих с диска ожидаемое «побольше» количество данных.

На рис. 5.32 показаны взаимосвязи между основными подпрограммами, участвующими в чтении файла. В том числе указано, кто кого вызывает.

Рис. 5.32. Некоторые из процедур, участвующих в чтении файла

Запись

Код, обеспечивающий запись в файл, находится в write.c. Запись в файл в большинстве своем сходна с чтением, и do_write просто вызывает read_write с флагом WRITING. Основное отличие записи в том, что здесь может потребоваться выделение дополнительных блоков. Функция write_map аналогична read_map с той разницей, что, вместо поиска физического адреса блока по *i*-узлу и «косвенным» блокам, она добавляет новое (точнее, она добавляет номер новой зоны, а не блока).

Код функции write_map сложный и длинный, поскольку эта функция должна учитывать несколько различных ситуаций. Если новая зона вставляется в начало файла, она должна быть вставлена в его *i*-узел.

Самый же худший случай, когда по мере роста файла простых косвенных блоков перестает хватать и приходится применять косвенные блоки второго уровня. После этого выделяется косвенный блок первого уровня, и его адрес заносится в блок второго. Как и в случае с чтением, для этого предусмотрена отдельная процедура, wr_indir. Если косвенный блок второго уровня выделить удалось и из-за этого диск переполнился, а на косвенный блок первого уровня места уже не хватает, то блок второго уровня нужно вернуть, чтобы избежать повреждения битовой карты.

Опять же, если бы в случае неудачи можно было бы просто сообщить о фатальной ошибке ядра, код был бы намного компактнее. Но, с точки зрения пользователя, гораздо лучше, когда при переполнении диска write возвращает код ошибки вместо того, чтобы провоцировать крах, к тому же с порчей файловой системы.

Функция wr_indir записывает новый зонный в косвенный блок, а чтобы преобразовать данные в нужный формат (порядок следования байтов), она вызывает одну из подпрограмм преобразования, conv2 или conv4. Пусть имя этой функции не вводит вас в заблуждение. Помните, что действительную запись данных на диск выполняют функции, обслуживающие кэш блоков.

Далее в write.c следует функция clear_zone. Она занимается очисткой блоков, оказавшихся в середине файла. Такое может случиться, если записать некоторое количество данных *после* конца файла. К счастью, это случается не очень часто.

Функция new_block вызывается из rw_chunk, когда требуется новый блок. На рис. 5.33 показаны шесть последовательных стадий увеличения файла. В этом примере размер блока равен 1 Кбайт, а зоны — 2 Кбайт.

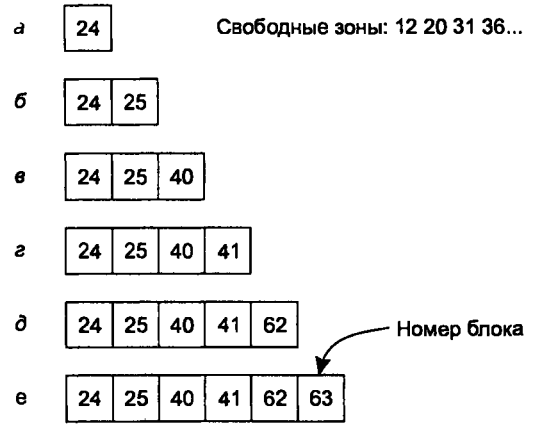


Рис. 5.33. Последовательное выделение блоков размером 1 Кбайт при размере зоны 2 Кбайт

Здесь при первом своем вызове new_block выделяет зону 12 (блоки 24 и 25). Затем она использует блок 25, который уже выделен, но еще не использован. При третьем вызове выделяется зона 20 (блоки 40 и 41) и т. д.

Функция zero_block очищает блок, стирая его предыдущее содержимое. Честно говоря, это описание длиннее, чем сам код.

Каналы ввода/вывода

Каналы (за исключением марсианских) во многих отношениях подобны файлам. В этом разделе мы сконцентрируем внимание на отличиях. Код, который мы будем обсуждать, находится в файле pipe.c.

Прежде всего, каналы создаются иначе, с помощью вызова pipe, а не creat. Вызов pipe выполняется функцией do_pipe. Вся работа do_pipe сводится к выделению *i*-узла для канала, после чего для него возвращаются два файловых дескриптора. Владельцем каналов является система, а не пользователь, и располагаются они на назначенном для этого устройстве, в качестве которого наиболее удобен RAM-диск, так как данные каналов не нуждаются в долговременном хранении.

Запись в канал и чтение из него отличаются от работы с файлом, поскольку канал имеет ограниченную пропускную способность. Попытка записать данные в переполненный канал приведет к приостановке пишущего процесса. Аналогичным образом, к блокировке приведет и попытка чтения из пустого канала. Итак, канал должен иметь два указателя, текущую позицию (она нужна читателям) и размер (интересный писателям). Все это вместе определяет, откуда и куда следуют данные.

Функция pipe_check делает различные проверки, выясняя, возможна ли работа с каналом. В дополнение к этим проверкам, которые могут привести к приостановке вызывающего процесса, pipe_check вызывает release, чтобы определить, может ли быть оживлен ранее приостановленный процесс, пытавшийся прочитать отсутствующие данные или злоупотребляющий записью. За возобновление пишущих процессов отвечает код на строке 113, читающих — код на строке 152. Здесь же обнаруживаются некорректно проложенные или «сломанные» (то есть без читателей) каналы.

За приостановку процесса отвечает функция suspend. Она просто сохраняет в таблице процессов параметры вызова и устанавливает в TRUE флаг dont_reply, чтобы файловая система не отправляла ответное сообщение.

Процедура release вызывается для проверки, может ли ранее приостановленный на канале процесс быть запущен. Если это так, вызывается функция revive, устанавливающая флаг, на который позже обратит внимание главный цикл. Эта функция не относится к системным вызовам, но работает через механизм передачи сообщений.

Последняя процедура в pipe.c называется do_unpause. Когда менеджер памяти пытается передать процессу сигнал, он обязан узнать, не приостановлен ли процесс на доступе к каналу или к специальному файлу (если это так, процесс будет разбужен с ошибкой EINTR). Так как менеджер памяти о каналах и специальных файлах никто в известность не ставил, он просит совета у файловой системы, передавая ей сообщение. Это сообщение обрабатывается функцией do_unpause, ко-

торая возобновляет работу процесса в том случае, если он заблокирован. Как и revive, do_unpause имеет сходство с системным вызовом, хотя и не относится к ним.

5.7.5. Каталоги и пути

Итак, мы завершили обзор того, как записываются и считываются файлы. Наша следующая задача — разобраться, как обрабатываются пути к файлам и каталоги.

Преобразование пути в i-узел

Многие системные вызовы принимают в качестве входного аргумента пути (то есть имена файлов). Это, например, вызовы open, unlink, mount. Большинству из них, перед тем как начать выполнять сам вызов, требуется определить *i*-узел указанного файла. Поэтому теперь мы подробно изучим, как имя файла преобразуется в *i*-узел. Общие контуры уже были обрисованы на рис. 5.12.

Разбором имен файлов занимается код в path.c. Первая его процедура, eat_path, получает на входе указатель на имя файла, разбирает имя, загружает нужный *i*-узел в память и возвращает указатель на узел. Она выполняет свою задачу при помощи функции last_dir, возвращающей *i*-узел каталога, в котором непосредственно находится файл. Затем, чтобы получить последний компонент пути, она вызывает advance. Если поиск объекта закончился неудачей, например, такое может произойти, если один из каталогов в пути не существует или существует, но недоступен для поиска, вместо указателя на *i*-узел возвращается значение NIL_INODE.

Имена файлов могут быть абсолютными или относительными и могут иметь произвольное число компонентов, разделенных слэшами. Распознаванием занимается last_dir. Сначала она проверяет первый символ пути с целью выяснить, абсолютный тот или относительный. Если путь абсолютный, в rip записывается указатель на корневой *i*-узел, для относительных путей в эту переменную помещается указатель на *i*-узел рабочего каталога.

С этого момента у last_dir есть путь и указатель на каталог, в котором нужно искать первый компонент. Затем начинается цикл, где путь разбирается компонент за компонентом. После конечной итерации возвращается указатель на конечный каталог.

Функция get_name является вспомогательной процедурой, извлекающей компоненты пути из строк. Более интересна функция advance, которая получает на входе указатель на каталог и строку и ищет указанную строку в каталоге. Если ей удается обнаружить в каталоге объект с таким именем, она возвращает указатель на его *i*-узел. Эта же функция учитывает особенности смонтированных файловых систем.

Хотя advance и управляет процессом поиска строки, самим сравнением строк с записями в каталоге занимается функция search_dir. Это единственный элемент во всей файловой системе, который непосредственно имеет дело с файлами каталогов. В функции есть два вложенных цикла, во внешнем перебираются блоки

каталогов, а во внутреннем просматриваются записи внутри каждого из блоков. Функция search_dir также вызывается для удаления записей из каталога или для внесения новых записей. Основные взаимосвязи между процедурами, участвующими в поиске файла по его пути, показаны на рис. 5.34.

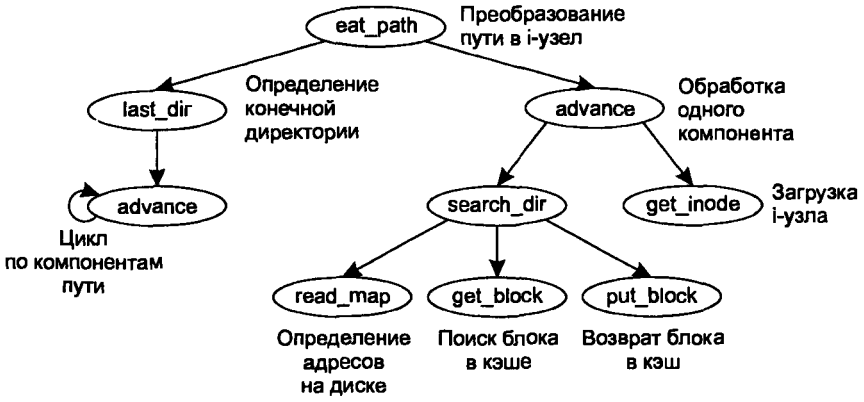


Рис. 5.34. Некоторые из процедур, участвующих в поиске файла по его пути

Монтирование файловых систем

Есть два системных вызова, оказывающих влияние на файловую систему в целом, это mount и umount. При помощи этих вызовов можно «склеивать» воедино отдельные файловые системы на различных дополнительных устройствах, образуя общую древовидную структуру. Как можно было видеть на рис. 5.28, при монтировании файловой системы ее суперблок и корневой *i*-узел считываются, и в суперблоке устанавливаются два указателя. Первый из них ссылается на точку монтирования (то есть *i*-узел, к которому присоединена монтированная файловая система), а второй указывает на корневой *i*-узел новой файловой системы. Эти два указателя и сцепляют вместе файловые системы.

Означенные указатели устанавливаются в последних строках функции do_mount из файла mount.c. Две страницы кода, которые предшествуют этому моменту, практически целиком посвящены проверке различных ошибок, ожидаемых при монтировании файловых систем. В том числе таких:

- указанный специальный файл не является блочным устройством;
- специальный файл представляет собой блочное устройство, но он уже смонтирован;
- у монтируемой файловой системы неправильная сигнатура;
- монтируемая файловая систем некорректна (то есть нет *i*-узлов);
- файл, к которому присоединяется файловая система, не существует или является специальным файлом;
- не хватает памяти для битовых карт монтируемой файловой системы;

- не хватает памяти для суперблока монтируемой файловой системы;
- не хватает памяти для корневого *i*-узла монтируемой файловой системы.

Хотя может показаться, что нет смысла повторяться, практика показывает, что в реальных операционных системах значительная часть кода выполняет рутинную работу, которая не слишком интересна, но имеет ключевое значение для работы системы. Если пользователь случайно, скажем, раз в месяц, будет пытаться монтировать поврежденную дискету и это будет приводить к тотальному сбою и порче файловой системы, он решит, что система нестабильна, и будет винить в этом не себя, а разработчика.

Томас Эдисон сделал одно замечание, которое применимо и к нашему случаю. Он сказал, что гений — это 1 % вдохновения и 99 % труда. Различие между хорошей и посредственной системой не в превосходном алгоритме планирования, а в том внимании, которое уделено деталям.

Демонтировать файловую систему проще, чем монтировать. Решает эту задачу функция do_umount. Ей важно позаботиться лишь о том, чтобы все файлы на демонтируемой файловой системе были закрыты и ни у одного процесса не было бы текущего каталога на ней. Проверка осуществляется тривиально: сканируется вся таблица *i*-узлов на предмет того, есть ли в памяти хотя бы один *i*-узел, принадлежащий демонтируемой файловой системе. Если да, umount рапортует об ошибке.

Последняя подпрограмма в файле mount.c носит имя name_to_dev. Она определяет старший и младший номера устройства по передаваемому ей на вход имени специального файла. Эти номера хранятся в самом *i*-узле, там, где у обычных файлов хранится информация о первой зоне. Это место пустует, поскольку у специальных файлов нет зон.

Создание и уничтожение ссылок

Следующий файл, который мы рассмотрим, называется link.c, он имеет дело со ссылками. Процедура do_link очень напоминает do_mount в том смысле, что практически весь ее код связан с проверкой ошибок. Ниже перечислены некоторые из возможных ошибок, которые могут произойти при вызове

```
link(file_name, link_name);
```

- файл file_name не существует или недоступен;
- у файла file_name уже есть максимальное количество ссылок;
- файл file_name является каталогом (создавать такие ссылки имеет право только суперпользователь);
- файл link_name уже существует;
- Link_name и file_name расположены на разных устройствах.

Если все в порядке, в каталоге создается новая запись с именем link_name и номером *i*-узла file_name. В коде имя name1 соответствует file_name, а name2 — link_name. Вносит новую запись в каталог функция search_dir, вызываемая из do_link.

Удаление файлов и каталогов происходит путем уничтожения ссылок на них. Поэтому оба системных вызова unlink и rmdir обслуживаются единственной функцией do_unlink. Опять же, в ней делается множество различных проверок, общий код убеждается, что файл существует и не является точкой монтирования, после чего, в зависимости от типа вызова, управление передается либо процедуре remove_dir, либо unlink_file. Мы вскоре обсудим эти две подпрограммы.

Код в файле link.c обеспечивает работу еще одного системного вызова, rename. Пользователям UNIX должна быть знакома команда оболочки mv, которая пользуется исключительно этим вызовом. Ее название отражает еще один из аспектов вызова, так как он способен не только менять имя файла, но и эффективно перемещать его из одного каталога в другой, причем операция перемещения атомарная, что позволяет избежать ситуации состязания. Этот вызов обрабатывается функцией do_rename. Перед выполнением команды проверяется множество условий, среди которых есть следующие:

- исходный файл должен существовать;
- новый путь не должен быть подкаталогом старого;
- новое имя не может быть ни «.», ни «..»;
- исходный и конечный каталоги должны находиться на одном устройстве;
- как исходный, так и конечный каталоги должны быть доступны на запись и на поиск файла и должны быть расположены на устройстве, доступном для записи;
- ни старое, ни новое имя не могут обозначать каталог, в который смонтирована файловая система.

Если одноименный файл существует, нужно проверить еще ряд дополнительных условий. Самое главное — должно быть разрешено удалить имеющийся файл.

В коде do_rename можно увидеть несколько приемов, служащих для снижения риска некоторых проблем. Если при переименовании файл с новым именем уже существует, то при заполненном диске может произойти ошибка, невзирая на то, что в конечном итоге дополнительное место не используется. Чтобы обойти эту проблему, старый файл сначала удаляется, за это отвечают строки кода с 260 по 266. По тем же соображениям из каталога сначала удаляется старое имя файла (в коде на 280-й строке), а затем в него записывается новое, во избежание выделения для каталога нового блока. Это соображение не применимо к тому случаю, когда новый и старый файлы расположены в разных каталогах, поэтому на 285-й строке, посвященной данной ситуации, сначала создается новое имя, а затем удаляется старое. Такой подход должен уменьшить риск повреждения файловой системы, если произойдет сбой, так как с точки зрения целостности гораздо лучше иметь две ссылки на один *i*-узел, чем *i*-узел, на который нет ссылок ни в одном из каталогов. Вероятность, что в процессе переименования кончится свободное место, невысока, а вероятность краха системы еще меньше, но в данном случае ничего не стоит приготовиться к худшему варианту.

Оставшиеся в файле link.c функции обеспечивают работу уже рассмотренных. В дополнение к ним, функция truncate вызывается и из некоторых других мест

в коде файловой системы. Она проходит по *i*-узлу зона за зоной, освобождая все найденные зоны, а также «косвенные» блоки. Функция remove_dir сначала делает ряд проверок, чтобы удостовериться, что каталог можно удалить, а затем вызывает unlink_file. Если никаких ошибок не произошло, запись каталога очищается и счетчик в *i*-узле декрементируется.

5.7.6. Прочие вызовы файловой системы

Оставшаяся группа вызовов представляют собой неоднородную смесь методов, затрагивающих состояние, каталоги, механизмы ограничения доступа, время и другие службы.

Управление состоянием каталогов и файлов

Файл stadir.c содержит код для четырех системных вызовов: chdir, chroot, stat и fstat. Изучая код last_dir, мы видели, что поиск файла по имени начинается с проверки первого символа пути. Если первый символ оказывался слэшем, брался указатель на корневой каталог, если нет — на рабочий.

Чтобы сменить текущий рабочий (или текущий корневой) каталог, нужно всего лишь изменить значения этих двух указателей в таблице процессов. То есть обратиться к функциям do_chdir и do_chroot соответственно. Обе они сначала выполняют необходимые проверки, а затем вызывают change с целью открыть новый каталог и заменить им старый.

В функции do_chdir есть код, не исполняемый пользовательскими процессами (с 35 строки по 51). Он предназначен специально для менеджера памяти, чтобы менять каталог при выполнении вызова exec. Когда пользователь запускает в своем рабочем каталоге файл, скажем, a.out, менеджеру памяти проще перейти в этот каталог, чем раздумывать, где он находится.

Оставшиеся два вызова, stat и fstat, одинаковы во всем, кроме способа задания файла. Первому из вызовов требуется имя файла, в то время как второму дескриптор открытого файла. Поэтому обе процедуры верхнего уровня, do_stat и do_fstat, вызывают stat_inode, которая и выполняет порученное. В функции do_stat перед вызовом stat_inode файл сначала открывается, с целью получить его дескриптор. Таким образом, обе эти функции передают в stat_inode дескриптор.

Вся работа stat_inode сводится к сбору информации о файле и записи ее в буфер. Затем этот буфер должен быть скопирован в пользовательское адресное пространство, так как он слишком велик и не помещается в сообщении.

Защита

Механизм ограничения доступа в MINIX основан на битах rwx. Это три набора битов, каждый из которых определяет доступность файла для его владельца, группы и для остальных. Значениями этих битов можно управлять при помощи системного вызова chmod, инициируемого функцией do_chmod из файла protect.c. Она сначала делает ряд проверок, а затем меняет режим доступа к файлу.

Вызов chown подобен вызову chmod в том, что он тоже меняет значения внутренних полей *i*-узла некоторого файла. Поэтому их реализация тоже достаточно

схожа, хотя do_chown позволяет менять владельца файла только суперпользователю. Обычные пользователи вправе применять этот вызов, чтобы менять принадлежность их собственных файлов к группе.

Вызов umask позволяет задавать маску (хранящуюся в таблице процессов), которая маскирует биты разрешений при последующих вызовах creat. Весь код уместился бы в одну строку, если бы не потребность в восстановлении старого значения маски. Это дополнительное требование утраивает объем кода.

При помощи системного вызова access процесс может выяснить, разрешен ли ему определенный способ доступа к файлу (например, на чтение). Этот вызов реализуется функцией do_access, которая считывает *i*-узел файла, а затем вызывает вспомогательную функцию, forbidden. Та, в свою очередь, проверяет UID и GID процесса, а также информацию в *i*-узле и в зависимости от этих данных принимает решение о том, что доступ разрешен или запрещен.

Небольшая процедура read_only проверяет файловую систему, на которой расположен переданный ей *i*-узел, и определяет, смонтирована ли она с доступом только на чтение. Эта функция необходима для предотвращения попыток записи на такую файловую систему.

Время

В MINIX есть несколько системных вызовов, манипулирующих временем, это utime, time, stime и times. Они представлены в табл. 5.9. Хотя три из четырех вызовов ничего не делают с файлами, они отнесены к файловой системе, поскольку информация о времени записывается в *i*-узлы файлов.

Таблица 5.9. Четыре системных вызова, связанных со временем

Вызов	Действие
utime	Устанавливает время последнего изменения файла
time	Устанавливает текущее реальное время в секундах
stime	Устанавливает часы реального времени
times	Запрашивает учетное время работы процесса

С каждым файлом связаны три 32-битных числа. Первые два из них хранят время последнего доступа к файлу и момент его последнего изменения. В третьем фиксируется время последнего изменения состояния самого *i*-узла. Это время будет меняться практически при каждом обращении к файлу, за исключением вызовов read и exec. Все значения хранятся в *i*-узле. При помощи системного вызова utime владелец файла или суперпользователь могут изменить время доступа и изменения. Это делается процедурой do_utime из файла time.c, которая получает *i*-узел и записывает в него значения времени. Затем сбрасываются флаги, индицирующие, что требуется обновить время, с целью избежать дорогостоящего и ненужного здесь вызова clock_time.

Файловая система не занимается поддержкой реального времени, это дело задачи часов в ядре. Следовательно, для файловой системы единственный способ узнать который час — послать сообщение ядру. Фактически, это именно то, что

делают обе функции do_time и do_stime. Реальное время измеряется в секундах, начиная с 1 января 1970 года.

Сведения о продолжительности работы процесса также хранятся в ядре. Каждый такт часов записывается на счет того или иного процесса. Чтобы получить эту информацию, нужно послать сообщение задаче системы, что и делает функция do_tims. Она не названа do_times потому, что многие компиляторы C добавляют знак подчеркивания перед каждым символом, а многие компоновщики усекают имена до восьми знакомест, в результате имя do_time становится неотличимым от do_times.

Все прочее

В файле misc.c размещен код системных вызовов, которые больше поместить было некуда. Вызов dup дублирует файловый дескриптор. Другими словами, он создает дескриптор, указывающий на тот же самый файл, что и аргумент вызова. У этого вызова есть вариант, называющийся dup2, и оба варианта выполняются процедурой do_dup. Обе эти устаревшие процедуры включены в MINIX лишь для поддержки старых скомпилированных программ. В MINIX с современной версией библиотеки C их место занял системный вызов fcntl.

Таблица 5.10. Параметры вызова fcntl по стандарту POSIX

Операция	Значение
F_DUPFD	Дублирует файловый дескриптор
F_GETFD	Получает значение флага «закрыть при exec»
F_SETFD	Устанавливает значение флага «закрыть при exec»
F_GETFL	Считывает флаги состояния
F_SETFL	Устанавливает флаги состояния
F_GETLK	Определяет владельца захваченного файла
F_SETLK	Устанавливает захват файла на чтение/запись
F_SETLKW	Устанавливает захват файла только на запись

Он и является рекомендуемым механизмом для работы с открытым файлом, а фактически выполняется функцией do_fcntl. Задачи его самые разные, они определяются при помощи флагов POSIX, перечисленных в табл. 5.10. Этому вызову передается дескриптор файла, код операции и при необходимости дополнительные аргументы. Так, эквивалентом вызова

```
dup2(fd, fd2);
```

будет следующий вызов fcntl:

```
fcntl(fd, F_DUPFD, fd2);
```

Некоторые из запросов просто устанавливают или считывают флаг, их код занимает всего несколько строк. Например, запрос F_SETFD устанавливает бит, означающий, что открытые файлы должны быть закрыты, когда пользователь вызовет exec. При помощи запроса F_GETFD можно определить текущее значение упомянутого бита. Запросы F_SETFL и F_GETFL позволяют устанавливать и считы-

вать флаги, индицирующие, доступен ли отдельный файл для работы в режиме без блокировки или для операций присоединения.

Кроме того, do_fcntl обеспечивает захват файлов. Запросы F_GETLK, F_SETLK и F_SETLKW транслируются в вызовы функции lock_op, которая обсуждалась ранее.

Следующий системный вызов, sync, копирует все модифицированные блоки и *i*-узлы на диск. Он выполняется функцией do_sync, которая просто ищет в таблицах все измененные *i*-узлы и блоки. Причем сначала необходимо обрабатывать *i*-узлы, так как функция rw_inode помещает результат в кэш блоков. После того как все модифицированные *i*-узлы перенесены в кэш, все измененные блоки из кэша записываются на диск.

Системные вызовы fork, exec, exit и set в действительности относятся к менеджеру памяти, но результаты их работы передаются и сюда. Когда процесс создает потомка, важно, чтобы и ядро, и менеджер памяти, и файловая система об этом узнали. Чтобы передать нужные сведения, менеджер памяти вызывает перечисленные функции файловой системы. Функции do_fork, do_exit и do_set заполняют таблицу процессов файловой системы соответствующей информацией, а do_exec ищет и закрывает все открытые файлы, отмеченные битом «закрыть при exec». Эти «системные вызовы» не вызываются пользовательскими процессами напрямую, а только менеджером памяти.

Последняя функция в файле не является системным вызовом, хотя и обрабатывается как один из них. Это do_revive. Она вызывается, когда закончила свою работу задача, у которой файловая система ранее запросила какие-то данные, например входные данные для пользовательского процесса. Тогда файловая система вновь запускает ранее приостановленный процесс и отправляет ему ответное сообщение.

5.7.7. Интерфейс устройств ввода/вывода

В MINIX ввод/вывод осуществляется путем передачи сообщений задачам в ядре. Интерфейс файловой системы для взаимодействия с задачами определен в файле device.c. Когда требуется произвести реальный ввод или вывод, для символьных специальных файлов функцию dev_io вызывает read_write, а для блочных ее вызывает rw_block. Функция dev_io строит стандартное сообщение и отправляет его нужной задаче. Вызов задачи происходит в строке:

```
(*dmap[major].dmap_rw)(task, &dev_mess);
```

Это выражение означает вызов функции, указатель на которую хранится в таблице dmess. Все соответствующие функции находятся здесь, в device.c. Когда dev_io ждет ответа от задачи, система приостанавливает работу, поскольку внутренней многозадачности в MINIX нет. Но обычно это ожидание длится недолго (например, порядка 50 мс).

Специальные действия могут потребоваться, когда открывается или закрывается специальный файл. Что именно необходимо — зависит от типа устройства. Поэтому в таблице dmap указано, какую функцию вызывать для закрытия или

открытия каждого из типов устройств в системе. Для дисковых устройств, будь то дисководы, жесткие диски или устройства в памяти, процедура единая — dev_opcl. В строке

```
mess_ptr->PROC_NR = fp - fproc;
```

вычисляется номер вызывающего процесса. Фактически работа выполняется функцией call_task, которой передается указатель на сообщение и номер процесса. Эту функцию мы рассмотрим далее. Дополнительно dev_opcl применяется и для закрытия устройств. Практически, с высоты этой функции единственная разница между открытием и закрытием состоит в том, что будет происходить после вызова call_task.

Среди прочих функций, вызываемых посредством dmap, следует упомянуть tty_open и tty_colse, обслуживающие последовательные линии, и ctty_open и ctty_close, работающие с консолью. Последняя из них, ctty_close, практически представляет собой функцию-заглушку, так как она без оглядки на что-либо возвращает статус «OK».

Выполнение системного вызова setsid требует некоторого участия файловой системы, что обеспечивается функцией do_setsid. Один из системных вызовов, ioctl, обслуживается по большей части в файле device.c, так как он тесно связан с интерфейсом задач. Когда вызывается ioctl, функция do_ioctl формирует сообщение и отправляет его соответствующей задаче.

Программы, написанные с учетом требований POSIX, должны для управления терминалом использовать функции, объявленные в файле include/termios.h. Стандартная библиотека C в MINIX преобразует вызовы этих функций в вызовы ioctl. Системный вызов ioctl применяется и для выполнения множества разнообразных действий с устройствами, не являющимися терминалами. Многие из этих операций описаны в главе 3.

Следующая функция в рассматриваемом нами файле объявлена как PRIVATE. Это небольшая вспомогательная процедура find_dev, извлекающая из полного номера устройства старший и младший номера.

Как уже было сказано ранее, чтение с большинства устройств и запись на них проходят через «руки» функции call_task. Она направляет сообщение нужной задаче в ядре при помощи sendrec. Попытка перенаправить сообщение может провалиться, если задача пытается перезапустить процесс в ответ на предыдущий запрос. Это, скорее всего, будет не тот процесс, запрос которого передается. Если call_task получает неинтерпретируемое ей сообщение, она выводит его на консоль. При нормальной работе MINIX подобных сообщений на консоли появляться не должно, но при разработке нового драйвера они вполне могут возникнуть.

Устройство /dev/tty физически не существует, это просто имя, по которому пользователь в многопользовательской системе может обращаться к своему терминалу, не выясняя, какой именно терминал находится в его распоряжении. Когда необходимо отправить сообщение /dev/tty, вызывается функция call_ctty, она определяет младший и старший номера для терминала и подставляет их в сообщение, прежде чем передать его call_task.

Наконец, мы дошли до последней функции в файле, no_dev. Ее адрес записывается в те ячейки таблицы, для которых не существует устройства. Например, она вызывается при попытке обратиться к сети с машины без сетевой поддержки. Данная функция возвращает код ошибки ENODEV и предотвращает крах системы при попытке доступа к несуществующим устройствам.

5.7.8. Инструменты общего назначения

В файловой системе есть несколько используемых в разных местах многоцелевых утилит. Они собраны в файле utility.c.

Первая процедура носит название clock_time. Она, чтобы узнать текущее реальное время, отправляет сообщение задаче часов. Следующая функция, fetch_name, своим существованием обязана тому, что многие из системных вызовов одним из аргументов принимают имя файла. Если имя короткое, оно непосредственно включается в сообщение файловой системе, но имя длинное располагается в адресном пространстве пользователя, а в сообщении передается указатель на него. Функция fetch_name проверяет, каким из способов передано имя, и получает его.

Две другие функции работают с основными классами ошибок. Функция no_sys обрабатывает ошибку, возникающую при вызове несуществующего системного вызова. Когда в системе происходит что-либо непоправимое, функция panic печатает сообщение и предлагает ядру прекратить работу.

Две оставшиеся функции помогают MINIX решать проблему различного порядка байтов на различных машинах (в частности, Intel и Motorola). Функции conv2 и conv4 применяются при записи на диск структуры данных, например *i*-узла, или при считывании ее с диска. Порядок байтов, с которым создавалась файловая система, фиксируется в ее суперблоке. Если он отличается от порядка, используемого процессором, данные необходимо конвертировать, переставляя байты. Преобразование происходит прозрачно для остальной системы.

Последний файл, содержащий две процедуры, носит имя putk.c.

Обе они предназначены для печати сообщений. Стандартные библиотечные процедуры применять нельзя, так как они отправляют сообщения файловой системе. Данные же процедуры общаются напрямую с задачей терминала. Мы уже видели практически идентичную пару функций, когда рассматривали подобный файл в менеджере памяти.

Резюме

Со стороны файловая система представляется как коллекция файлов, каталогов и действий над ними. Можно считывать или записывать файлы, создавать и уничтожать каталоги и перемещать файлы из одного каталога в другой. В большинстве современных файловых систем поддерживается иерархическая структура каталогов, когда в один каталог может быть вложен второй и т. д., до бесконечности.

Если же смотреть изнутри, открывается совершенно другая картина. Разработчики файловой системы должны заботиться о том, как выделяется место, и отслеживать, какой блок какому файлу соответствует. Мы увидели, что в разных файловых системах структура каталога различается. Надежность и производительность файловой системы тоже имеет существенное значение.

Исключительно важны как для пользователей, так и для разработчиков системы вопросы безопасности и защиты. Мы обсудили известные уязвимости старых систем и общие проблемы, вероятные у многих других. Размышляя о защите, мы также рассмотрели аутентификацию, с паролем и без, списки управления доступом и другой вид пропускных систем — мандатные, а также матричную модель защиты.

Наконец, мы подробно изучили файловую систему MINIX. Ее код велик, но не очень сложен. Файловая система принимает запросы от пользовательских процессов, находит в таблице системных вызовов адрес процедуры и вызывает эту процедуру, чтобы обслужить запрос. Благодаря модульной структуре и тому, что файловая система вынесена из ядра, ее можно легко выделить из MINIX, превратив в самостоятельный сетевой файловый сервер, внеся лишь незначительные поправки.

При обращении к файлу MINIX буферизует блоки в кэше и пытается делать упреждающее чтение при последовательном режиме работы с файлом. Если кэш достаточно велик, то при операциях, когда многократно происходят обращения к одному и тому же набору программ (например, в процессе компиляции), нужный файл уже с высокой вероятностью окажется в памяти.

Вопросы

1. Создайте пять различных путей к файлу /etc/passwd. Подсказка: используйте элементы каталога «.» и «..».
2. В системах, поддерживающих последовательный доступ, всегда имеется операция для «перемотки» файлов. Нужна ли такая операция в системах, поддерживающих файлы произвольного доступа?
3. В некоторых операционных системах предоставляется системный вызов rename, позволяющий сменить имя файла. Есть ли разница между использованием этого системного вызова и копированием файла с новым именем с последующим удалением старого файла?
4. Рассмотрите дерево каталогов на рис. 5.5. Если /usr/jim является рабочим каталогом, как будет выглядеть абсолютный путь для файла с относительным путем ../ast/x?
5. Как указывалось в тексте, работа с монолитными файлами приводит к фрагментации диска, поскольку теряется некоторая часть дискового пространства в последних блоках файлов, чья длина не кратна целому числу блоков. Является эта фрагментация внутренней или внешней? Приведите аналогию с вопросом, обсуждавшимся в предыдущей главе.

6. Простая операционная система поддерживает только один каталог, но позволяет хранить в нем произвольное количество файлов с именами любой длины. Можно ли на такой системе имитировать иерархическую файловую систему? Как?
7. Учет свободного дискового пространства может осуществляться с помощью списков или битовых массивов. Дисковые адреса состоят из D битов. При каком условии для диска из B блоков, F из которых свободны, список займет меньше места, чем битовый массив? Выразите ваш ответ в процентах от объема диска для D = 16.
8. Было предложено хранить первую часть каждого файла системы UNIX в том же дисковом блоке, что и его *i*-узел. Каковы преимущества такого подхода?
9. Производительность файловой системы зависит от процента блоков, которые удается в нем найти. Напишите формулу для среднего времени удовлетворения запроса блока при частоте успешных обращений, равной *h*, если обслуживание запроса с помощью кэша занимает 1 мс, а для считывания блока с диска требуется 40 мс. Нарисуйте график этой зависимости для значений *h* в интервале от 0 до 1,0.
10. У гибкого диска 40 цилиндров. Операция поиска занимает 6 мс на цилиндр. Если не пытаться разместить блоки файла впритирку, два логически последовательных блока (то есть следующих один за другим в файле) окажутся в среднем на расстоянии 13 цилиндров друг от друга. Однако если операционная система пытается объединять логически соседние блоки в кластеры, то среднее межблочное расстояние может быть уменьшено до 2 (например) цилиндров. Сколько понадобится времени в обоих случаях для считывания 100-блочного файла, если задержка вращения составляет 100 мс, а время переноса одного блока равно 25 мс?
11. Осмыслено ли и до какой степени периодическое уплотнение дискового пространства?
12. Как можно модифицировать операционную систему TENEX, чтобы избежать ошибки с паролями, описанной в тексте?
13. Закончив учебное заведение, вы получаете должность директора большого университетского компьютерного центра, где только что отправили свою древнюю операционную систему на заслуженный отдых и перешли на UNIX. Вы начинаете работать. Через пятнадцать минут ваша ассистентка вбегает в кабинет в панике: «Какие-то студенты обнаружили алгоритм, которым мы шифруем наши пароли, и выложили его в Интернете». Какой будет ваша реакция?
14. Схема защиты Морриса–Томпсона с *n*-разрядными случайными числами («солью») была разработана, чтобы затруднить взломщику отгадывание паролей при помощи подготовленного заранее словаря. Защищает ли такая схема от студентов, пытающихся угадать пароль суперпользователя? Предполагается, что файл паролей доступен для чтения.

15. У факультета технической кибернетики есть локальная сеть с большим количеством машин, работающих под управлением операционной системы UNIX. Пользователь на любой машине может ввести команду вида machine4 who, и эта команда будет выполнена на компьютере machine4, для чего пользователю не нужно регистрироваться на удаленном компьютере. Это свойство реализовано следующим образом. Ядро системы машины пользователя посылает команду и ее UID удаленной машине. Надежна ли такая схема, если ядрам системы можно доверять? Что, если одна из машин представляет собой персональный компьютер студента, на который не установлена защита?
16. При удалении файла его блоки, как правило, возвращаются в список свободных блоков, но их содержимое не стирается. Как вы полагаете, будет ли хорошей идеей, если операционная система будет очищать каждый блок перед тем, как его освободит? Рассмотрите в вашем ответе факторы безопасности и производительности, а также покажите, какой эффект окажет эта схема на каждый фактор.
17. В данной главе обсуждались различные механизмы защиты: списки полномочий, списки управления доступом и биты rwx UNIX. Какой из этих механизмов может быть применен для каждой из следующих проблем (рассматривая UNIX, представьте, что группы соответствуют таким категориям, как факультет, студенты, секретари и т. д.):
    - Кен хочет, чтобы его файлы могли читать все, кроме его коллеги по офису;
    - Мич и Стив хотят вместе пользоваться некоторыми секретными файлами;
    - Линда хочет сделать открытыми некоторые из своих файлов.
18. Рассмотрим следующий механизм защиты. Каждому объекту и каждому процессу присваивается номер. Процесс вправе обращаться только к тем объектам, номер которых больше его номера. На какие из схем, обсуждаемых в тексте, это похоже? Каковы принципиальные отличия между данной и другими?
19. Возможна ли атака с внедрением троянского коня в систему, защищенную списками полномочий?
20. Две студентки с факультета кибернетики, Кэролин и Элинор, обсуждают *i*-узлы. Кэролин утверждает, что память стала настолько дешевой, что в расчете на тенденцию при открытии файла проще и быстрее считать новую копию *i*-узла в таблицу *i*-узлов, чем искать этот *i*-узел по всей таблице. Элинор не согласна. Кто прав?
21. В чем разница между вирусом и червем? Как каждый из них размножается?
22. Символические ссылки представляют собой файлы, косвенно указывающие на другие файлы или каталоги. В отличие от обычных ссылок, реализованных в MINIX, у символической ссылки есть собственный *i*-узел

и собственный блок данных. Блок данных содержит путь к объекту, на который направлена ссылка, а отдельный *i*-узел позволяет ссылке принадлежать другому пользователю и иметь другие разрешения доступа. Такая ссылка не обязана находиться на том же устройстве, что и файл, на который она ссылается. Символические ссылки не являются частью стандарта POSIX 1990, но ожидается, что в будущем они будут добавлены в MINIX. Реализуйте их поддержку в MINIX.

23. Может статься, что предельный объем файла в MINIX, равный 64 Мбайт, недостаточен для ваших потребностей. Расширьте файловую систему, добавив в нее «косвенные» блоки третьего уровня косвенности. Для этого задействуйте неиспользуемые поля *i*-узла.
24. Покажите, как установка параметра ROBUST делает систему более или менее устойчивой в случае краха. Тщательно изучите, что происходит, когда измененный блок вытесняется из кэша. Учтите, что модификация блока может быть дополнена модификацией *i*-узла и битовой карты.
25. Размер таблицы filp сейчас задается константой NR_FILPS, определенной в fs/const.h. Чтобы приспособить сетевую систему для работы большего числа пользователей, может потребоваться увеличить константу NR_PROCS в include/minix/config.h. Как, в зависимости от NR_PROCS, нужно изменить NR_FILPS?
26. Разработайте механизм, позволяющий добавить поддержку «чужеязычной» файловой системы, чтобы, например, можно было монтировать файловую систему MS-DOS в каталог MINIX.
27. Предположим, произошел технологический прорыв и появилась энергонезависимая память, сохраняющая свое содержимое при исчезновении питания, по цене и производительности не отличающаяся от традиционных ОЗУ. Как это скажется на файловых системах?

Глава 6
Библиография

В предыдущих пяти главах был рассмотрен широкий спектр вопросов. Цель последней главы заключается в том, чтобы помочь заинтересованным читателям продолжить изучение операционных систем. Первый раздел представляет собой список книг, рекомендованных для дальнейшего чтения. Второй раздел является алфавитным перечнем всех изданий, на которые мы ссылались в данной книге.

Помимо этих списков литературы, в поисках новых статей по вопросам принципов операционных систем стоит заглянуть в материалы симпозиумов Ассоциации по вычислительной технике (ACM, Association for Computing Machinery), проводимых раз в год, а также в отчеты ежегодной международной конференции по распределенным вычислительным системам от IEEE. Представляют интерес и материалы симпозиума USENIX по проектированию и реализации операционных систем. Кроме того, интересные статьи об операционных системах печатаются в журналах *ACM Transactions on Computer Systems* и *Operating Systems Review*.

6.1. Литература, рекомендуемая для дальнейшего чтения

6.1.1. Введение и общие труды

1. Brooks, «The Mythical Man Month: Essays on Software Engineering».

   Фред Брукс был одним из разработчиков операционной системы OS/360. Он на собственном опыте научился отличать то, что работает, от того, что не работает.

2. Comer, D.: «Operating System Design. The Xinu Approach, Upper Saddle River», NJ, Prentice Hall, 1984.

   Книга об операционной системе Xinu, работавшей на LSI-11. Содержит детальный обзор исходных кодов и полный листинг на языке C.

3. Corbató, «On Building Systems That Will Fail».

   В своей лекции по поводу получения премии Тьюринга основатель системы разделения времени высказывает примерно те же соображения, что и Брукс в упомянутой выше книге. Он приходит к выводу, что все сложные системы в конце концов ждет крах, и чтобы иметь хоть какой-то шанс на успех, необходимо избегать любой сложности и придерживаться простоты и элегантности проекта.

4. Deitel, «Operating Systems», 2nd Ed.

Базовая книга об операционных системах. В дополнение к стандартным материалам, содержит обзоры операционных систем UNIX, MS-DOS, MVS, VM, OS/2 и ОС для Macintosh.

5. Finkel, «An operating Systems Vade Mecum».

   Еще одна общая книга об операционных системах. Она ориентирована на практические задачи и неплохо написана. Содержимое охватывает многие рассматриваемые здесь вопросы, предоставляя тем самым хорошую возможность взглянуть на них под новым углом.

6. IEEE, «Information Technology — Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]».

   Это не беллетристика, а стандарт. Некоторые разделы вполне удобочитаемы, особенно дополнение B, «Rationale and Notes» (обоснования и примечания), благодаря которому часто становится понятно, почему то или иное сделано именно так, а не иначе. Одно из преимуществ ссылки на стандарт заключается в том, что в этом документе по определению нет ошибок. Если какой-либо типографской опечатке в макросе удается преодолеть процесс редактирования, то после этого она уже не ошибка, а официальный факт.

7. Lampson, «Hints for Computer System Design».

   Батлер Лэмпсон, один из ведущих в мире разработчиков передовых операционных систем, за годы собрал множество подсказок, предложений и руководящих указаний и поместил их в эту увлекательную и информативную статью. Как и книга Брукса, эта статья является обязательным чтением для каждого честолюбивого разработчика операционных систем.

8. Lewine, «POSIX Programmer's Guide».

   Эта книга описывает требования POSIX более подробно, чем документы стандарта, а также включает обсуждения с примерами преобразования старых программ в стандарт POSIX и описывает методы разработки новых программ для среды POSIX.

9. Silberschatz and Galvin, «Operating System Concepts», 4th Ed.

   Еще один учебник по операционным системам. Тематика: процессы, управление накопителями, файлы и распределенные системы. Рассматриваются два конкретных случая: UNIX и Mach. На обложке полно динозавров. Какое отношение они имеют к операционным системам 1990-го года, непонятно.

10. Stallings, «Operating Systems», 4th Ed.

    Опять-таки учебник, и опять-таки по операционным системам. В нем затрагиваются все традиционные темы, а также включено небольшое количество материала по распределенным средам.

11. Stevens, «Advanced Programming in the UNIX Environment».

    Руководство, как написать программы на языке C с помощью интерфейса системных вызовов UNIX и стандартной библиотеки C. Примеры основа-

ны на версиях системы UNIX System V Release 4 и 4.4BSD. Подробное описание этих реализаций строится относительно стандарта POSIX.

12. Switzer, «Operating Systems, A Practical Approach».

    В основе книги лежит подход, аналогичный этому тексту. Теоретические понятия иллюстрируются примерами псевдокода и исходными кодами модельной операционной системы TUNIX. В отличие от MINIX TUNIX не рассчитана на работу с реальными машинами и выполняется на виртуальной. Она не столь реалистична, как MINIX, в плане драйверов устройств, но превосходит ее в некоторых других областях, в частности в организации виртуальной памяти.

6.1.2. Процессы

1. Andrews and Schneider, «Concepts and Notations for Concurrent Programming».

   Учебный и обзорный материал по процессам и межпроцессному взаимодействию, в котором, среди прочего, описываются активное ожидание, семафоры, мониторы, передача сообщений и другие технологии. В этой статье также показывается, как эти понятия встроены в различные языки программирования.

2. Ben-Ari, «Principles of Concurrent Programming».

   Эта небольшая книга полностью посвящена проблемам межпроцессного взаимодействия. Среди прочих тем в отдельных главах обсуждаются взаимные исключения, семафоры, мониторы и вопросы кормления философов.

3. Dubois et al., «Synchronization, Coherence, and Event Ordering in Multiprocessors».

   Учебная статья по вопросам синхронизации в мультипроцессорных системах с общей памятью. Однако некоторые идеи в равной мере применимы также к однопроцессорным системам и системам с распределенной памятью.

4. Silberschatz and Galvin, «Operating System Concepts», 4th Ed.

   В главах с 4-й по 6-ю рассматриваются семафоры и взаимодействие между процессами, включая планирование, критические секции, семафоры, мониторы и классические проблемы взаимодействия между процессами.

6.1.3. Ввод/вывод

1. Chen et al., «RAID: High Performance Reliable Secondary Storage».

   Параллельное использование нескольких жестких дисков для ускорения ввода/вывода является современной тенденцией в профессиональных сис-

темах. Авторы обсуждают эту идею и изучают различные способы организации, сравнивая производительность, стоимость и надежность.

2. Coffman et al., «System Deadlocks».

   Эта книга представляет собой краткое введение во взаимоблокировки. В ней рассказывается о причинах их возникновения и способах их предотвращения или обнаружения.

3. Finkel, «An operating Systems Vade Mecum», 2nd Ed.

   В пятой главе рассматривается аппаратное обеспечение ввода/вывода, в частности терминалы и диски.

4. Geist and Daniel, «A Continuum of Disk Scheduling Algorithms».

   В данной книге обсуждается обобщенный алгоритм планирования передвижений блока головок диска. Приводятся результаты всесторонних экспериментов и моделирования.

5. Holt, «Some Deadlock Properties of Computer Systems».

   Обсуждение взаимоблокировок. Холт вводит модель направленных графов, хорошо подходящую для анализа некоторых тупиковых ситуаций.

6. IEEE Computer Magazine, Mar. 1994.

   Этот выпуск журнала *Computer* содержит восемь статей по передовым технологиям ввода/вывода, в которых обсуждаются такие темы, как моделирование, накопители с высокой производительностью, кэширование, ввод/вывод для «параллельных» компьютеров и мультимедиа.

7. Isloor and Marsland, «The Deadlock Problem: An Overview».

   Учебное пособие по взаимоблокировкам, в котором особое внимание уделяется системам баз данных. Описывается множество моделей и алгоритмов.

8. Stevens, «Heuristics for Disk Drive Positioning in 4.3BSD».

   Детальное изучение производительности жесткого диска в Berkeley UNIX. Как это часто бывает с компьютерными системами, реальность оказывается сложнее, чем предсказывает теория.

9. Wilkes et al., «The HP AutoRAID Hierarchical Storage System».

   RAID (Redundant Array of Inexpensive Disks) — это «массив» связанных воедино небольших дисков, вместе образующих единую высокопроизводительную систему. В этом тексте авторы раскрывают некоторые детали системы, которую они построили в HP Labs.

6.1.4. Управление памятью

1. Denning, «Virtual Memory».

   Классическая статья по многим вопросам виртуальной памяти. Деннинг был одним из пионеров в этой области. Он же является автором концепции рабочего набора.

2. Denning, «Working Sets Past and Present».

   Хороший обзор многочисленных алгоритмов управления памятью и страничной подкачкой. Включена всеобъемлющая библиография.

3. Knuth, «The Art of Computer Programming», vol. 1.

   В этой книге обсуждаются и сравниваются различные алгоритмы управления памятью, такие как «первый подходящий», «лучший подходящий» и т. д.

4. Silberschatz et al., «Applied Operating System Concepts».

   Главы 9 и 10 посвящены управлению памятью, включая свопинг, страничную подкачку и сегментацию. Упоминаются различные алгоритмы замещения страниц.

6.1.5. Файловые системы

1. Denning, «The United States vs. Craig Neidorf».

   Когда юный хакер обнаружил и опубликовал информацию о том, как работает телефонная система, ему было предъявлено обвинение в компьютерном мошенничестве. В статье описывается это судебное дело, затронувшее многие фундаментальные понятия, как, например, свобода слова. Следом за статьей несколько человек высказывают свои особые мнения, а Деннинг контраргументирует.

2. Hafner and Markoff, «Cyberpunk».

   Три захватывающие истории о молодых хакерах, вламывающихся в компьютеры по всему миру, рассказанные опубликовавшим историю с интернет-червем компьютерным обозревателем *New York Times* и его женой, журналисткой. *Computer*, Feb 2000.

3. Harbron, «File Systems».

   Книга по устройству файловых систем, приложениям и производительности. Описываются как структура, так и алгоритмы.

4. McKusick et al., «A Fast File System for UNIX».

   Файловая система для UNIX была полностью переделана для версии 4.2 BSD. В этой статье описывается устройство новой файловой системы с особым акцентом на ее производительности.

5. Silberschatz et al., «Applied Operating System Concepts».

   Глава 11 посвящена теме файловых систем. Среди прочих тем в ней рассматриваются операции с файлами, методы доступа, каталоги и вопросы реализации.

6. Stallings, «Operating Systems», 2nd Ed.

   В главе 14 содержится существенное количество материала о безопасном окружении, а также о хакерах, вирусах и других угрозах.

6.2. Алфавитный список литературы

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», ACM Trans. on Computer Systems, vol. 10, pp. 53–79, Feb. 1992.
2. Andrews, G. R., and Schneider, F. B., «Concepts and Notations for Concurrent Programming», Computing Surveys, vol. 15, pp. 3–43, Mar. 1983.
3. Bach M. J., «The Disign of the UNIX Operating System», Englewood Cliffs, NJ, Prentice Hall, 1987.
4. Bala, K., Kaashoek, M. F., Weihl, W., «Software Prefetching and Caching for Translation Lookaside Buffers», Proc. First Symp. on Operating System Design and Implementation, USENIX, pp. 243–254, 1994.
5. Bays, C., «A Comparison of Next-Fit, First-Fit, and Best-Fit», Commun. of the ACM, vol. 20, pp. 191–192, Mar. 1977.
6. Ben-Ari, M., «Principles of Concurrent Programming», Upper Saddle River, NJ, Prentice Hall International, 1982.
7. Brooks, F. P., Jr., «The Mythical Man-Month: Essays on Software Engineering», Anniversary edition, Reading, MA, Addison-Wesley, 1975.
8. Cadow, H. «OS/360 Job Control language», Englewood Cliffs, NJ: Prentisse Hall, 1970.
9. Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A., «RAID: High Performance Reliable Storage», Comp. Surv., vol. 26, pp. 145–185, Jun. 1994.
10. Cheriton, D. R., «An Experiment Using Registers for Fast Message-Based Interprocess Communication», Operating Systems Review, vol. 18, pp. 12–20, Oct. 1984.
11. Coffman, E. G., Elphick, M. J., and Shoshani, A., «System Deadlocks», Computing Surveys, vol. 3, pp. 67–78, Jun. 1971.
12. Comer, D., «Operating Systems Design. The Xinu Approach», Englewood Cliffs, N.J: Prentisse Hall, 1984.
13. Corbato, F. J., «On Building Systems That Will Fail», Commun. of the ACM, vol. 34, pp. 72–81, Jun. 1991.
14. Corbato, F. J., Merwin-Daggett, M., and Daley, R. C., «An Experimental Time-Sharing System», Proc. AFIPS Fall Joint Computer Conf., AFIPS, pp. 335–344, 1962.
15. Corbato, F. J., Saltzer, J. H., and Clingen, C. T.: «MULTICS — The First Seven Years», Proc. AFIPS Spring Joint Computer Conf., AFIPS, pp. 571–583, 1972.
16. Corbato, F. J., and Vyssotsky,V. A., «Introduction and Overview of the MULTICS System», Proc. AFIPS Fall Joint Computer Conf., AFIPS, pp. 185–196, 1965.

17. Courtois, P. J., Heymans, F., and Parnas, D. L., «Concurrent Control with Readers and Writers», Commun. of the ACM, vol. 10, pp. 667–668, Oct. 1971.

18. Daley, R. C., and Dennis, J. B., «Virtual Memory, Process, and Sharing in MULTICS», Commun. of the ACM, vol. 11, pp. 306–312, May 1968.

19. Dan, A., Sitaram, D., and Shahabuddin, P., «Scheduling Policies for an On-Demand Video Server with Batching», Proc. Second Int'l Conf. on Multimedia, ACM, pp. 15–23, 1994.

20. Deitel H. M. «Operating Systems», 2nd Ed., Reading, MA: Addison-Wesley, 1990.

21. Denning, D., «The United States vs. Craig Neidorf», Commun. of the ACM, vol. 34, pp. 22–43, March 1991.

22. Denning, P. J., «The Working Set Model for Program Behavior», Commun. of the ACM, vol. 11, pp. 323–333, 1968a.

23. Denning, P. J., «Thrashing: Its Causes and Prevention», Proc. AFIPS National Computer Conf., AFIPS, pp. 915–922, 1968b.

24. Denning, P. J., «Virtual Memory», Computing Surveys, vol. 2, pp. 153–189, Sep. 1970.

25. Denning, P. J., «Working Sets Past and Present», IEEE Trans. on Software Engineering, vol. SE-6, pp. 64–84, Jan. 1980.

26. Dennis, J. B., and VAN HORN, E. C., «Programming Semantics for Multiprogrammed Computations», Commun. of the ACM, vol. 9, pp. 143–155, Mar. 1966.

27. Dijkstra, E. W., «Co-operating Sequential Processes, in Programming Languages», Genuys, F. (Ed.), London, Academic Press, 1965.

28. Dijkstra, E. W., «The Structure of THE Multiprogramming System», Commun. of the ACM, vol. 11, pp. 341–346, May 1968.

29. Dubois, M., Scheurich, C., and Briggs, F. A., «Synchronization, Coherence, and Event Ordering in Multiprocessors», Computer, vol. 21, pp. 9–21, Feb. 1988.

30. Engler, D. R., Kaashoek, M. F., and O'toole, J. Jr., «Exokernel: An Operating System Architecture for Application-Level Resource Management», Proc. 15th Symp. on Operating Systems Principles, ACM, pp. 251–266, 1995.

31. Fabry, R. S.: «Capability-Based Addressing», Commun. of the ACM, vol. 17, pp. 403–412, July 1974.

32. Feeley, M. J., Morgan, W. E., Pighin, F. H., Karlin, A. R., Levy, H. M., and Thek-Kath, C. A., «Implementing Global Memory Management in a Workstation Cluster», Proc. 15th Symp. on Operating Systems Principles, ACM, pp. 201–212, 1995.

33. Finkel, R.A. «An Operating Systems Vade Mecum», 2nd Ed., Englewood Cliffs, NJ: Prentice Hall, 1988.

34. Fotheringham, J., «Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store», Commun. of the ACM, vol. 4, pp. 435–436, Oct. 1961.

35. Geist, R., and Daniel, S., «A Continuum of Disk Scheduling Algorithms», ACM Trans. on Computer Systems, vol. 5, pp. 77–92, Feb. 1987.

36. Golden, D., and Pechura, M., «The Structure of Microcomputer File Systems», Commun. of the ACM, vol. 29, pp. 222–230, Mar. 1986.

37. Graham, R., «Use of High-Level Languages for System Programming», Project MAC Report TM-13, M.I.T., Sep. 1970.

38. Hafner, K., and Markoff, J., «Cyberpunk», New York, Simon and Schuster, 1991.

39. Harbron, T. R., «File Systems», Upper Saddle River, NJ, Prentice Hall, 1988.

40. Hauser, C., Jacobi, C., Theimer, M., Welch, B., and Weiser, M., «Using Threads in Interactive Systems: A Case Study», Proc. 14th Symp. on Operating Systems Principles, ACM, pp. 94–105, 1993.

41. Havender, J. W., «Avoiding Deadlock in Multitasking Systems», IBM Systems Journal, vol. 7, pp. 74–84, 1968.

42. Hebbard, B., et al., «A Penetration Analysis of the Michigan Terminal System», Operating Systems Review, vol. 14, pp. 7–20, Jan. 1980.

43. Hoare, C. A. R., «Monitors, An Operating System Structuring Concept», Commun. of the ACM, vol. 17, pp. 549–557, Oct. 1974; Erratum in Commun. of the ACM, vol. 18, p. 95, Feb. 1975.

44. Holt, R. C., «Some Deadlock Properties of Computer Systems», Computing Surveys, vol. 4, pp. 179–196, Sep. 1972.

45. Huck, J., and Hays, J., «Architectural Support for Translation Table Management in Large Address Space Machines», Proc. 20th Int'l Symp. on Computer Architecture, ACM, pp. 39–50, 1993.

46. IEEE: Information Technology — Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language], New York, Institute of Electrical and Electronics Engineers, Inc., 1990[1].

47. Isloor, S. S., and Marsland, T. A., «The Deadlock Problem: An Overview», Computer, vol. 13, pp. 58–78, Sep. 1980.

48. Kernighan, B. W., and Pike, R., «The UNIX Programming Environment», Upper Saddle River, NJ, Prentice Hall, 1984.

49. Kleinrock, L., «Queueing Systems», Vol. 1, New York, John Wiley, 1975.

[1] Эта редакция позднее, в 1994 году, была дополнена расширением POSIX P1003.4a — Threads Extension for portable Operating Systems. В 2001 году вышла переработанная IEEE совместно с Open Group (торговая марка UNIX) редакция The Open Group Base Specifications Issue 6 (IEEE Std 1003.1-2001). Она описывает стандартный интерфейс операционной системы, включая оболочку и утилиты. Стандарт ориентирован как на прикладных разработчиков, так и на разработчиков операционных систем. Его электронный вариант представлен в Интернете, например, по адресу http://www.45.free.net/docs/POSIX/index.html. — *Примеч. ред.*

50. Knuth, D. E., «The Art of Computer Programming», vol. 1: «Fundamental Algorithms», 2nd Ed., Reading, MA, Addison-Wesley, 1973.
51. Lampson, B. W., «A Scheduling Philosophy for Multiprogramming Systems», Commun. of the ACM, vol. 11, pp. 347–360, May 1968.
52. Lampson, B. W., «A Note on the Confinement Problem», Commun. of the ACM, vol. 10, pp. 613–615, Oct. 1973.
53. Levin, R., Cohen, E. S., Corwin, W. M., Pollack, F. J., and Wulf, W. A., «Policy/Mechanism Separation in Hydra», Proc. Fifth Symp. on Operating Systems Principles, ACM, pp. 132–140, 1975.
54. Lewine, D., «POSIX Programmer's Guide», Sebastopol, CA, O'Reilly & Associates, 1991.
55. Li, K., and Hudak, P., «Memory Coherence in Shared Virtual Memory Systems», ACM Trans. on Computer Systems, vol. 7, pp. 321–359, Nov. 1989.
56. Linde, R. R., «Operating System Penetration», Proc. AFIPS National Computer Conf., AFIPS, pp. 361–368, 1975.
57. Lions, J., «Lions' Commentary on Unix 6th Edition, with Source Code», San Jose, CA, Peer-to-Peer Communications, 1996.
58. Liu, C. L., and Layland, J. W., «Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment», Journal of the ACM, vol. 20, pp. 46–61, Jan. 1973.
59. Marsh, B. D., Scott, M. L., Leblanc, T. J., and Markatos, E. P., «First-Class User-Level Threads», Proc. 13th Symp. on Operating Systems Principles, ACM, pp. 110–121, 1991.
60. McKusick, M. J., Joy, W. N., Leffler, S. J., and Fabry, R. S., «A Fast File System for UNIX», ACM Trans. on Computer Systems, vol. 2, pp. 181–197, Aug. 1984.
61. Morris, R., and Thompson, K., «Password Security: A Case History», Commun. of the ACM, vol. 22, pp. 594–597, Nov. 1979.
62. Mullender, S. J., and Tanenbaum, A. S., «Immediate Files», Software – Practice and Experience, vol. 14, pp. 365–368, Apr. 1984.
63. Organick, E. I., «The Multics System», Cambridge, MA, M.I.T. Press, 1972.
64. Peterson, G. L., «Myths about the Mutual Exclusion Problem», Information Processing Letters, vol. 12, pp. 115–116, Jun. 1981.
65. Rosenblum, M., and Ousterhout, J. K., «The Design and Implementation of a Log-Structured File System», Proc. 13th Symp. on Oper. Sys. Prin., ACM, pp. 1–15, 1991.
66. Saltzer, J. H., «Protection and Control of Information Sharing in MULTICS», Commun. of the ACM, vol. 17, pp. 388–402, Jul. 1974.
67. Saltzer, J. H., and Schroeder, M. D., «The Protection of Information in Computer Systems», Proc. IEEE, vol. 63, pp. 1278–1308, Sept. 1975.
68. Salus, P. H., «UNIX At 25», Byte, vol. 19, pp. 75–82, Oct. 1994.

69. Sandhu, R. S., «Lattice-Based Access Control Models», Computer, vol. 26, pp. 9–19, Nov. 1993.

70. Seawright, L. H., and Mackinnon, R. A., «VM/370 — A Study of Multiplicity and Usefulness», IBM Systems Journal, vol. 18, pp. 4–17, 1979.

71. Silberschatz, A., Galvin, P. B., and Gagne, G., «Applied Operating System Concepts», New York, Wiley, 2000.

72. Stallings, W., «Operating Systems», 4th Ed., Upper Saddle River, NJ, Prentice Hall, 2001.

73. Stevens, W. R., «Advanced Programming in the UNIX Environment», Reading, MA, Addison-Wesley, 1992.

74. Stevens, W. R., «Heuristics for Disc Drive Positioning in 4.3BSD», Computing Systems, vol. 2, pp. 251–274, Summer 1989.

75. Stoll, C., «The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage», New York, Doubleday, 1989.

76. Switzer, R.W. «Operating Systems, A Practical Approach», London: Prenitce Hall Int'1, 1993.

77. Tai, K.C., and Carver, R. H., «VP: A New Operation for Semaphores», Operating Systems Review, vol. 30, pp. 5–11, Jul. 1996.

78. Talluri, M., and Hill, M. D., «Surpassing The Tlb Performance of Superpages With Less Operating System Support», Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, ACM, pp. 171–182, 1994.

79. Talluri, M., Hill, M. D., and Khalidi, Y. A., «A New Page Table for 64-bit Address Spaces», Proc. 15th Symp. on Operating Systems Prin., ACM, pp. 184–200, 1995.

80. Tanenbaum, A. S., and Van Steen, M. R., «Distributed Systems», Upper Saddle River, NJ, Prentice Hall, 2002[1].

81. Tanenbaum, 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», Commun. of the ACM, vol. 33, pp. 46–63, Dec. 1990.

82. Teory, T. J.: «Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems», Proc. AFIPS Fall Joint Computer Conf., AFIPS, pp. 1–11, 1972.

83. Thompson, K., «Unix Implementation», Bell System Technical Journal, vol. 57, pp. 1931-1946, July-Aug. 1978.

[1] Русский перевод — Таненбаум Э. С., Ван Стеен М. Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003

84. Uhlig, R., Nagle, D., Stanley, T., Mudge, T., Secrest, S., and Brown, R., «Design Tradeoffs for Software-Managed TLBs», ACM Trans. on Computer Systems, vol. 12, pp. 175–205, Aug. 1994.

85. Vahalia, U., «UNIX Internals — The New Frontiers», Upper Saddle River, NJ, Prentice Hall, 1996[1].

86. Waldersprueger, C.A., and Weihi, W.E., «Lottery Scheduling: Flexible Proportional-Share Resource Management», Proc. First Symp. On Operating Systems Design and Implementation, USENIX, pp. 1–12, 1994.

87. Wilkes, J., Golding, R., Staelin, C, and Sullivan, T., «The HP AutoRAID Hierarchical Storage System», ACM Trans. on Computer Systems, vol. 14, pp. 108–136, Feb. 1996.

88. Wulf, W. A., Cohen, E. S., Corwin, W. M., Jones, A. K., Levin, R., Pierson, C., and Pollack, F. J., «HYDRA: The Kernel of a Multiprocessor Operating System», Commun. of the ACM, vol. 17, pp. 337–345, Jun. 1974.

89. Zekauskas, M. J., Sawdon, W. A., and Bershad, B. N., «Software Write Detection for a Distributed Shared Memory», Proc. First Symp. on Operating System Design and Implementation, USENIX, pp. 87–100, 1994.

[1] Русский перевод — Вахалия Ю. UNIX изнутри. СПб.: Питер, 2003

Алфавитный указатель

В

Г

Д

Ж

З

И

К

Л

М

Н

О

П

P

C

T

У

Ф

Ц

Ч

Ш

Э

Я

Эндрю С. Таненбаум, Альберт С. Вудхалл

Операционные системы: разработка и реализация (+CD)

Классика CS

Перевел с английского Д. Шинтяков

Главный редактор	*Е Строганова*
Заведующий редакцией	*А Кривцов*
Руководитель проекта	*В. Шрага*
Литературный редактор	*Е. Васильев*
Художник	*Л Адуевская*
Корректор	*В Листова*
Верстка	*Ю. Сергиенко*

Лицензия ИД № 05784 от 07 09 01.

Подписано к печати 01.08.05. Формат 70×100/16. Усл п. л. 46,44 Тираж 1 000 Заказ № 1576

ООО «Питер Принт», 194044, Саикт-Петербург, пр Б. Сампсониевский, 29а.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2, 95 3005 — литература учебная

Отпечатано с готовых диапозитивов в типографии «Береста»

196006, Санкт-Петербург, ул Коли Томчака, 28

Тел : 388-90-00