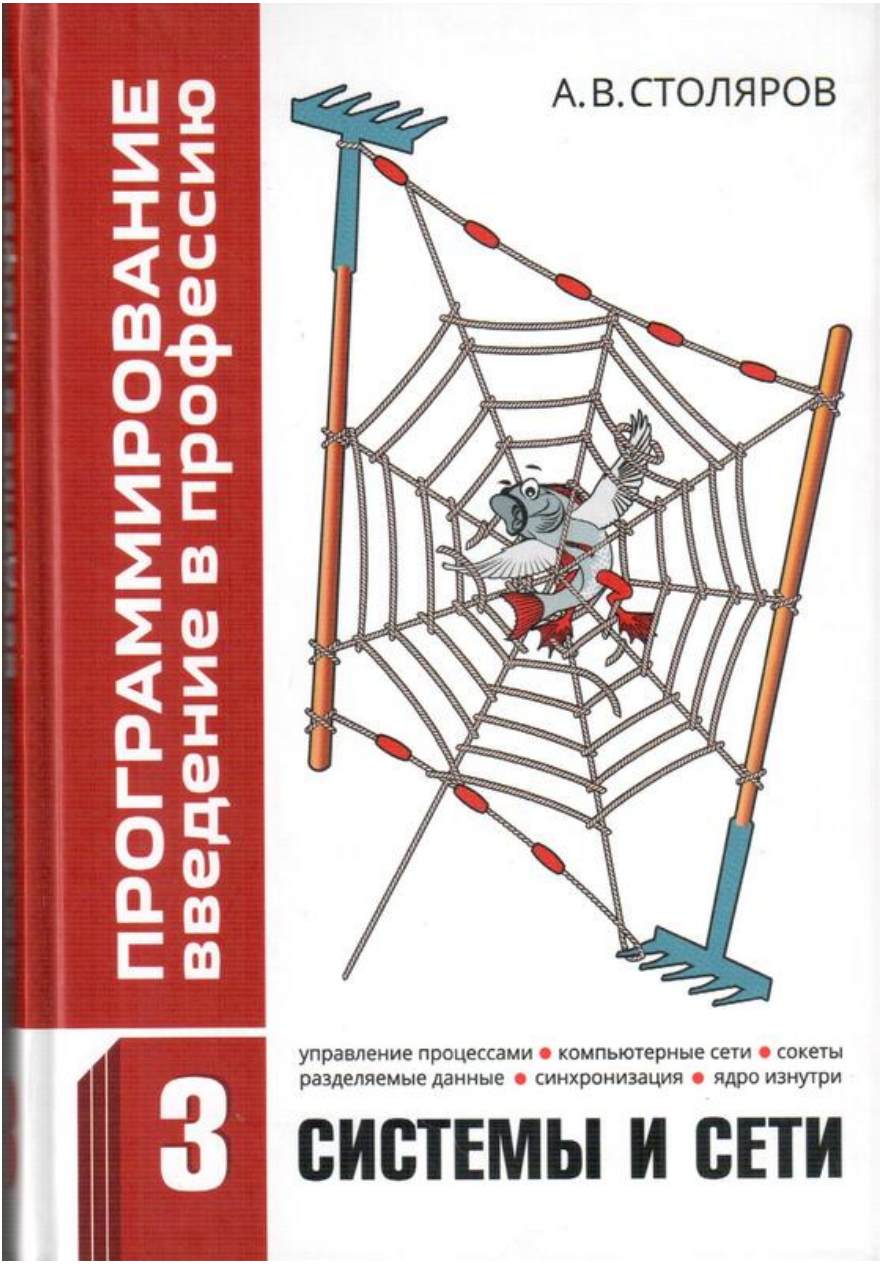

ПУБЛИЧНАЯ ЛИЦЕНЗИЯ

Учебное пособие Андрея Викторовича Столярова «Программирование: введение в профессию. III: Системы и сети», опубликованное в издательстве МАКС Пресс в 2017 году, называемое далее «Произведением», защищено действующим российским и международным авторско-правовым законодательством. Все права на Произведение, предусмотренные законом, как имущественные, так и неимущественные, принадлежат его автору.

Настоящая Лицензия устанавливает способы использования электронной версии Произведения, право на которые предоставлено автором и правообладателем неограниченному кругу лиц, при условии безоговорочного принятия этими лицами всех условий данной Лицензии. Любое использование Произведения, не соответствующее условиям данной Лицензии, а равно и использование Произведения лицами, не согласными с условиями Лицензии, возможно только при наличии письменного разрешения автора и правообладателя, а при отсутствии такого разрешения является противозаконным и преследуется в рамках гражданского, административного и уголовного права.

Автор и правообладатель настоящим **разрешает** следующие виды использования данного файла, являющегося электронным представлением Произведения, без уведомления правообладателя и без выплаты авторского вознаграждения:

1. Воспроизведение Произведения (полностью или частично) на бумаге путём распечатки с помощью принтера в одном экземпляре для удовлетворения личных бытовых или учебных потребностей, без права передачи воспроизведённого экземпляра другим лицам;

2. Копирование и распространение данного файла в электронном виде, в том числе путём записи на физические носители и путём передачи по компьютерным сетям, с соблюдением следующих условий: (1) **все воспроизведённые и передаваемые любым лицам экземпляры файла являются точными копиями оригинального файла** в формате PDF, при копировании не производится никаких изъятий, сокращений, дополнений, искажений и любых других изменений, включая изменение формата представления файла; (2) **распространение и передача копий другим лицам производится исключительно *бесплатно*, то есть при передаче не взимается никакое вознаграждение ни в какой форме**, в том числе в форме просмотра рекламы, в форме платы за носитель или за сам акт копирования и передачи, даже если такая плата оказывается значительно меньше фактической стоимости или себестоимости носителя, акта копирования и т. п.

Любые другие способы распространения данного файла при отсутствии письменного разрешения автора запрещены. В частности, **запрещается**: внесение каких-либо изменений в данный файл, создание и распространение искаженных экземпляров, в том числе экземпляров, содержащих какую-либо часть произведения; распространение данного файла в Сети Интернет через веб-сайты, оказывающие платные услуги, через сайты коммерческих компаний и индивидуальных предпринимателей, а также **через сайты, содержащие рекламу любого рода**; продажа и обмен физических носителей, содержащих данный файл, даже если вознаграждение значительно меньше себестоимости носителя; включение данного файла в состав каких-либо информационных и иных продуктов; распространение данного файла в составе какой-либо платной услуги или в дополнение к такой услуге. С другой стороны, **разрешается** дарение (бесплатная передача) носителей, содержащих данный файл, бесплатная запись данного файла на носители, принадлежащие другим пользователям, распространение данного файла через бесплатные децентрализованные файлообменные P2P-сети и т. п. Ссылки на экземпляр файла, расположенный на официальном сайте автора, разрешены без ограничений.

А. В. Столяров *запрещает* Российскому авторскому обществу и любым другим организациям производить любого рода лицензирование любых его произведений и осуществлять в интересах автора какую бы то ни было иную связанную с авторскими правами деятельность без его письменного разрешения.

А. В. СТОЛЯРОВ

ПРОГРАММИРОВАНИЕ
ВВЕДЕНИЕ В ПРОФЕССИЮ

III: СИСТЕМЫ И СЕТИ

Москва — 2017

УДК 519.683+004.2+004.45
ББК 32.97
С81

Столяров А. В.
С81 Программирование: введение в профессию. III: Системы и сети. – М.: МАКС Пресс, 2017. – 400 с.
ISBN 978-5-317-05606-3

Вашему вниманию предлагается третий том учебника «Программирование: введение в профессию», все части которого объединены использованием unix-систем в качестве единой учебной операционной среды. Учебник ориентирован в основном на самостоятельное изучение программирования.

Третий том учебника посвящён операционной системе как явлению, услугам, которые она предоставляет пользовательским программам, и некоторым принципам её собственного устройства; рассматриваются системные вызовы файлового ввода-вывода, управление процессами и межпроцессное взаимодействие, подсистема сокетов. Отдельная часть целиком посвящена программированию с разделяемыми данными, проблемам синхронизации и взаимоисключения. Рассмотрены различные модели виртуальной памяти, принципы взаимодействия драйверов с внешними устройствами и другие аспекты функционирования операционной системы.

Предполагается, что читатель владеет языком Си.

Для школьников, студентов, преподавателей и всех, кто интересуется программированием.

УДК 519.683+004.2+004.45
ББК 32.97

ISBN 978-5-317-05606-3

Оглавление

Содержание

Предисловие к третьему тому

Третий том серии «Программирование: введение в профессию», который вы держите в руках, целиком посвящён возможностям операционной системы и тому, как она всего этого добивается. Том открывает часть V, в которой мы познакомимся с основными «видимыми» объектами операционной системы и тем, как с ними взаимодействовать через системные вызовы; в эту часть вошёл материал по файловому вводу-выводу, управлению процессами, межпроцессному взаимодействию и по управлению драйвером терминала.

Обсуждение услуг ядра системы продолжается в части VI, посвящённой компьютерным сетям. Любая передача данных через сеть, разумеется, тоже становится возможной только благодаря операционной системе. Как показывает опыт автора этих строк, простота интерфейса сокетов и лёгкость, с которой Unix позволяет создавать программы, взаимодействующие друг с другом через сеть, буквально восхищает многих учеников и резко повышает «градус энтузиазма». Материал по сокетам предваряется небольшим «ликбезом» по сетям в целом, рассматривается стек протоколов TCP/IP, даются примеры протоколов прикладного уровня.

В VII части рассказывается о том, какие проблемы могут возникнуть, когда одновременно несколько «действующих лиц» (выполняющихся программ или экземпляров одной и той же программы) будут обращаться к одной и той же порции данных, будь то область оперативной памяти или дисковый файл. Именно такая ситуация возникает, если использовать так называемые ***треды*** — независимые потоки параллельного исполнения в рамках одного экземпляра запущенной программы. Надо признать, что эта часть книги написана скорее не с целью научить читателя использованию тредов (хотя вся нужная для этого информация там есть), а с целью убедить его, что треды использовать не следует; но даже если читатель вслед за автором примет решение никогда и ни для чего не применять многопоточное программирование, материал части останется полезным. Во-первых, такое решение должно было принято осознанно, с возможностью привести аргументы в его пользу. Во-вторых, работа с разделяемыми данными встречается отнюдь не только в многопоточных программых: те же многопользовательские базы данных тому пример, и рано или поздно любой профессиональный программист с задачей такого рода столкнётся. Кроме того, в ядре операционной системы работа с разделяемыми данными, увы, неизбежна, так что некоторые аспекты его внутреннего устройства было

бы трудно объяснить без предварительного рассказа о разделяемых данных, критических секциях и взаимоисключениях.

Том заканчивается частью VIII, где делается попытка объяснить, как операционная система устроена изнутри. Здесь мы познакомимся с моделями виртуальной памяти, поговорим о планировании времени центрального процессора и о том, как на самом деле устроен ввод-вывод.

Первоначально планировалось, что весь этот материал войдёт во второй том, но итоговый объём частей, посвящённых ассемблерному программированию и языку Си, эти планы благополучно разрушил. Впрочем, и хорошо, что разрушил; возникновение «внепланового» третьего тома позволило посвятить отдельную часть работе с компьютерными сетями и более подробно осветить многие другие аспекты операционных систем как явления.

Как водится, несколько слов о работе с текстом. Предполагается, что читатель знаком с материалом двух первых томов, в особенности с языком Си. Как и в предшествующих томах, в тексте встречаются фрагменты, набранные уменьшенным шрифтом без засечек. При первом прочтении книги такие фрагменты можно безболезненно пропустить; некоторые из них могут содержать ссылки вперёд и предназначаться для читателей, уже кое-что знающих о предмете. Примеры того, **как не надо делать**, помечены вот таким знаком на полях:

Вводимые новые понятия выделены ***жирным курсивом***. Кроме того, в тексте используется *курсив* для смыслового выделения и **жирный шрифт** для выделения фактов и правил, которые желательно не забывать, иначе могут возникнуть проблемы с последующим материалом. Домашняя страница этой книги в Интернете расположена по адресу

`http://www.stolyarov.info/books/programming_intro`

Здесь вы можете найти архив примеров программ, приведённых в книге, а также электронную версию самой книги. Для примеров, включённых в архив, в тексте книги указаны имена файлов.

Книга «Программирование: введение в профессию» стала возможна благодаря людям, которые сочли мой проект достойным того, чтобы пожертвовать на него деньги. Ниже приведён полный (на 14 июля 2017 г.) список донэйторов, кроме тех, кто пожелал сохранить инкогнито; каждый участник включён в список под таким именем, которое указал сам:

Gremlin, Grigoriy Kraynov, Шер Арсений Владимирович,
Таранов Василий, Сергей Сетченков, Валерия Шакирзянова,
Катерина Галкина, Илья Лобанов, Сюзана Тевдорадзе,
Иванова Оксана, Куликова Юлия, Соколов Кирилл Владимирович,
jeckep, Кулёва Анна Сергеевна, Ермакова Марина Александровна,
Переведенцев Максим Олегович, Костарев Иван Сергеевич,
Донцов Евгений, Олег Французов, Степан Холопкин,
Попов Артём Сергеевич, Александр Быков, Белобородов И. Б.,
Ким Максим, artyrian, Игорь Эльман, Илюшкин Никита,
Кальсин Сергей Александрович, Евгений Земцов, Шрамов Георгий,
Владимир Лазарев, eupharina, Николай Королев,
Горошевский Алексей Валерьевич, Леменков Д. Д., Forester, say42,
Аня «canja» Ф., Сергей, big_fellow, Волканов Дмитрий Юрьевич,
Танечка, Татьяна 'Vikora' Алпатова, Беляев Андрей,
Лошкины (Александр и Дарья), Кирилл Алексеев, kopish32,
Екатерина Глазкова, Олег «burunduk3» Давыдов, Дмитрий Кронберг,
yobibyte, Михаил Аграновский, Александр Шепелёв, G.Nerc=Y.uR,
Василий Артемьев, Смирнов Денис, Pavel Korzhenko,
Руслан Степаненко, Терешко Григорий Юрьевич 15e65d3d, Lothlorien,
vasiliandets, Максим Филиппов, Глеб Семёнов, Павел, unDEFER,
kilolife, Арбичев, Рябинин Сергей Анатольевич, Nikolay Ksenev,
Кучин Вадим, Мария Вихрева, igneus, Александр Чернов,
Roman Kurynin, Власов Андрей, Дергачёв Борис Николаевич,
Алексей Алексеевич, Георгий Мошкин, Владимир Руцкий,
Федулов Роман Сергеевич, Шадрин Денис,
Панфёров Антон Александрович, os80, Зубков Иван,
Архипенко Константин Владимирович, Асирян Александр,
Дмитрий С. Гуськов, Тойгильдин Владислав, Masutacu, D.A.X.,
Каганов Владислав, Анастасия Назарова, Гена Иван Евгеньевич,
Линара Адылова, Александр, izin, Николай Подонин, Юлия Корухова,
Кузьменкова Евгения Анатольевна, Сергей «GDM» Иванов,
Андрей Шестимеров, vap, Грацианова Татьяна Юрьевна,
Меньшов Юрий Николаевич, nvasil, В. Красных,
Огрызков Станислав Анатольевич, Бузов Денис Николаевич, capgelka,
Волкович Максим Сергеевич, Nikolay Ksenev, Владимир Ермоленко,
Горячая Илона Владимировна, Полякова Ирина Николаевна,
Антон Хван, Иван К., Сальников Алексей,
Щеславский Алексей Владимирович, Золотарев Роман Евгеньевич,
Константин Глазков, Сергей Черевков, Андрей Литвинов, Шубин М. В.,
Сыщенко Алексей, Николай Курто, Ковригин Дмитрий Анатольевич,
Андрей Кабанец, Юрий Скурский, Дмитрий Беляев, Баранов Виталий,
Новиков Сергей Михайлович, maxon86, mishamm,
Спиридонов Сергей Вячеславович, Сергей Черевков, Кирилл Филатов,
Чаплыгин Андрей, Виктор Николаевич Остроухов, Николай Богданов.

Всем перечисленным людям, а также и тем, кто предпочёл остаться неизвестным, я хотел бы высказать свою глубочайшую признательность. Дело тут даже не в том, что благодаря всем вам мне удалось написать и издать лучшее произведение своей жизни, хотя и в этом, конечно, тоже; но сейчас для меня важнее другое: благодаря пожертвованиям, которые поступили и продолжают поступать, я теперь точно знаю, что всё делаю правильно.

Часть 5

Объекты и услуги операционной системы

Операционная система неоднократно упоминалась в предыдущих частях нашей книги. Мы уже знаем, что **операционная система — это программа**, которая загружается на компьютере первой (на самом деле не совсем первой, но от тех, кто загружается раньше, к моменту начала обычной работы не остаётся никаких следов) и что все остальные программы запускаются и работают *под управлением* операционной системы. Сам по себе термин «операционная система» употребляется в литературе и лексиконе специалистов в двух существенно различных значениях; в предыдущих томах нашей книги под «операционной системой» всегда понималась именно та *программа*, которая загружается первой и осуществляет работу с аппаратурой, обрабатывает прерывания и обслуживает системные вызовы. Эту программу также называют ***ядром операционной системы***.

В некоторых случаях под словосочетанием «операционная система» понимают большой набор программ, включающий, кроме ядра, разнообразные системные утилиты, программы для управления и настройки, иногда даже компиляторы и интерпретаторы языков программирования. Как видим, термин «операционная система» может использоваться в разных значениях, тогда как использование термина «ядро операционной системы» никакой неопределённости не оставляет. В дальнейшем мы будем использовать оба термина, считая их синонимами; для обозначения «большого набора программ» мы будем применять другие термины, такие как «дистрибутив», «окружение» и т. п., наиболее подходящие по контексту.

Программируя на языке ассемблера, мы выяснили, что написанные нами программы самостоятельно могут только преобразовывать информацию в отведённой им памяти, а любое взаимодействие с внешним

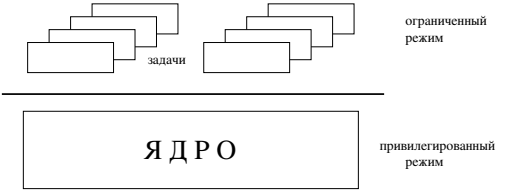

Рис. 5.1. Ядро и задачи

миром осуществляется исключительно через операционную систему, причём обращение пользовательской задачи к операционной системе за услугами называется ***системным вызовом***.

Нам также известно, что команды, которые умеет выполнять центральный процессор, делятся на *привилегированные* и *непривилегированные*, а сам процессор может работать в *привилегированном режиме* или в *ограниченном режиме*; в первом процессор готов выполнять любые команды, во втором — только непривилегированные. Операционная система стартует в привилегированном режиме, а когда ей приходится отдать управление пользовательской задаче, она меняет режим процессора на ограниченный.

Процессор может перейти обратно в привилегированный режим только при условии одновременной передачи управления на так называемые *обработчики* — фрагменты программного кода, адреса которых заданы заранее, во время работы в привилегированном режиме, и эти адреса нельзя изменить, находясь в режиме ограниченном; естественно, операционная система настраивает процессор так, чтобы в роли обработчиков использовались её собственные фрагменты. Как следствие, одновременно с переключением режима процессора на привилегированный управление всегда возвращается операционной системе, так что только её код может выполнять привилегированные команды, больше ни у кого такой возможности нет. Возврат управления системе происходит в трёх основных случаях: когда внимание операционной системы требуется внешнему устройству (***аппаратные/внешние прерывания*** или просто ***прерывания***); когда процессор не может исполнить очередную команду из-за её некорректности — несуществующего кода операции, деления на ноль, обращения за пределы отведённой задаче памяти и т. п. (***внутренние прерывания*** или ***исключения***); и при системном вызове (иногда говорят, что системные вызовы реализуются с помощью ***программных прерываний***).

Подчеркнём, что с момента загрузки **ядро является единственной программой, код которой выполняется в привилегированном режиме центрального процессора**. Все остальные программы, вне зависимости от уровня их полномочий, выполняются в ограниченном режиме в виде пользовательских задач (рис. 5.1). Вынужденная

обходиться только непривилегированными командами, пользовательская задача сама по себе не может сделать ничего такого, что повлияло бы на всю вычислительную систему[1] в целом, хотя она может попросить о таких действиях операционную систему.

Мы также успели обсудить, что́ на самом деле скрывается под термином «одновременное выполнение задач»; мы уже знаем, что ***мультизадачность*** бывает ***пакетной***, где задача перестаёт выполняться лишь дойдя до конца, либо когда ей требуется дождаться какого-нибудь события, чаще всего — окончания операции ввода; кроме того, существует ***режим разделения времени***, при котором задачи дополнительно могут сменять друг друга, если одна из них успела отработать определённое количество времени, а в системе есть другие задачи, готовые к выполнению. Пакетный режим чаще всего применяется на всевозможных суперкомпьютерах, где конкретная последовательность обработки задач не столь важна, гораздо важнее их общее количество, которое система успеет обработать за единицу времени, и нет смысла терять драгоценное процессорное время на лишние переключения между задачами. На машинах, ведущих непосредственный диалог с пользователями, а также на серверных машинах, обслуживающих множество клиентов одновременно, пакетный режим не подходит и приходится применять режим разделения времени, обеспечивающий каждой задаче, которая готова к выполнению, определённую долю процессорного времени. При этом создаётся впечатление, что каждая задача работает на своей собственной машине, возможно, более медленной. Кроме этих двух режимов, выделяют также промежуточный ***невытесняющий режим мультизадачности***, сейчас вышедший из употребления, и ***режим реального времени***, где задача планировщика не в том, чтобы дать всем поработать хоть сколько-нибудь, а в том, чтобы обеспечить предсказуемость «грязного» времени выполнения задачи, т. е. возможность заранее понять, успеет ли та или иная программа завершиться к заданному моменту времени.

Наконец, нам известно, что для полноценного обеспечения мультизадачного режима аппаратура компьютера должна поддерживать по меньшей мере четыре возможности: аппарат прерываний, защиту памяти, разделение машинных команд на обычные и привилегированные (с поддержкой соответствующих режимов процессора) и таймер — простейшее устройство, генерирующее запросы прерывания через равные промежутки времени.

Все эти сведения об операционных системах мы были вынуждены приводить в предыдущих частях нашей книги, поскольку без них не могли двигаться дальше; но задачи, стоявшие перед нами ранее, бы-

[1] Под ***вычислительной системой*** понимается, грубо говоря, компьютер со всеми программами, которые на нём используются.

ли достаточно сложны сами по себе, так что операционные системы пришлось обсуждать довольно поверхностно.

Подробное рассмотрение операционных систем логично разделяется на два аспекта: как выглядят услуги, которые операционная система предоставляет программам, выполняющимся под её управлением, и каково устройство самой системы. В этой и двух последующих частях нашей книги мы постараемся осветить первый аспект, то есть показать, как выглядит операционная система с точки зрения программиста, непосредственно использующего системные вызовы. Заключительная часть третьего тома, восьмая в общей нумерации, посвящена тому, как операционная система устроена и каким образом она функционирует.

5.1. Что делает операционная система

В этой главе мы попытаемся ответить на вопрос, *зачем* нужна операционная система. Мы выделим здесь два основных пункта: **управление пользовательскими задачами** и **взаимодействие с внешними устройствами** (управление аппаратурой).

5.1.1. Управление пользовательскими задачами

К управлению пользовательскими задачами относится прежде всего **запуск** задач и их **останов**. В самом деле, сам факт появления задачи в системе означает, что её кто-то запустил; как было замечено выше, операционная система, стартовав на компьютере, полностью «захватывает власть», так что запустить задачу может только она. С другой стороны, задачи имеют свойство завершаться, и действие, обратное запуску, то есть останов задачи — тоже, естественно, выполняет именно операционная система: с каждой задачей у неё связаны достаточно сложные структуры данных, и только она знает, как их нужно правильно модифицировать, чтобы исключить завершённую задачу из очереди на выполнение, высвободить занятую ею память и т. п.

Кроме того, к управлению задачами следует отнести **планирование времени центрального процессора**; коль скоро у нас в системе может одновременно существовать больше одной задачи, нужно как-то распределять между ними время центрального процессора (или центральных процессоров, если их больше одного, а в наше время это обычно так и есть), временно приостанавливать одни задачи и ставить на исполнение другие. В зависимости от используемого типа мультизадачности (см. т. 2, §3.6.1) задаче может быть предоставлена возможность работать, пока ей есть что делать (пакетная мультизадачность), либо

задачам могут выделяться ***кванты времени***[2], по истечении которых планировщик снимает текущую задачу с выполнения и даёт поработать другим задачам (конечно, если в системе есть другие задачи, готовые к выполнению).

В понятие управления пользовательскими задачами входит также **управление оперативной памятью** в широком смысле, включающем, например, распределение имеющейся памяти между задачами, управление настройками виртуальной памяти, защитой памяти и прочими возможностями аппаратуры, связанными с оперативной памятью. Временная откачка данных из оперативной памяти на диск с целью её высвобождения тоже относится к управлению памятью.

Отдельные запущенные в системе программы (так называемые «процессы»[3]) должны иметь возможность при необходимости взаимодействовать между собой; но поскольку повлиять друг на друга напрямую они не могут, средства для такого взаимодействия должна предоставлять им операционная система; иначе говоря, к управлению пользовательскими задачами относится также **организация межпроцессного взаимодействия**.

Наконец, к управлению пользовательскими задачами мы отнесём **разграничение полномочий**. Сразу отметим, что речь обычно идёт о полномочиях *пользователей*, но реально *полномочиями* в системе обладает не человек, а запущенная программа — всё тот же *процесс*, который может представлять собой пользовательскую задачу или её часть.

В системе, к которой имеет доступ много пользователей — например, если это серверный компьютер, обслуживающий группу людей или целое предприятие — необходимость регламентирования доступа пользователей к имеющимся ресурсам очевидна: даже если исключить злонамеренные действия (что, вообще говоря, уже само по себе далеко не всегда возможно), простые (не злонамеренные) ошибки пользователей в такой системе могут приводить к слишком серьёзным последствиям, если у каждого будет возможность сделать в системе что угодно. Но

[2]Отметим, что исходный англоязычный термин — *time slice*; слово *quantum* в этом контексте используется реже — обычно в случаях, когда квант времени имеет переменную длину и речь идёт о вычислении этой длины.

[3]Читатель может заметить, что ранее мы использовали термин «задача». Действительно, мы избегали употребления слова «процесс», поскольку это достаточно узкий технический термин, который мы ещё не обсуждали; слово «задача» мы полагали интуитивно понятным. В контексте обсуждения взаимодействия запущенных программ друг с другом мы, к сожалению, не можем себе позволить употребление слова «задача», поскольку как раз «задачи» между собой взаимодействуют гораздо реже, чем процессы, и сами термины «межпроцессное взаимодействие» или «взаимодействие процессов» являются устоявшимися и часто употребляемыми, тогда как термина «взаимодействие задач» вы в литературе, скорее всего, не встретите. К обсуждению термина «процесс» мы вернёмся позже, пока вы можете приблизительно считать, что процесс — это программа, которую запустили под управлением операционной системы.

многопользовательские компьютеры встречаются сравнительно редко, так что такой случай, *на первый взгляд*, можно было бы считать исключением из правил, а для машин, не предполагающих работы большого числа разных людей — например, для домашних компьютеров, ноутбуков, офисных рабочих мест — пользователю компьютера можно предоставить неограниченные полномочия в системе, ведь он в системе один.

Именно так решили поступить в конце прошлого века специалисты одной известной компании, выпускающей коммерческое программное обеспечение, в том числе операционные системы. Последствия этой ошибки не заставили себя долго ждать. Как мы уже отмечали, реально доступ к ресурсам и возможностям компьютера осуществляет не *пользователь*, а запускаемые им *программы*; фактически при запуске любой программы пользователь передаёт управление своим компьютером автору этой программы. Между тем, далеко не всякой программе можно полностью доверять. Отсутствие разграничения полномочий в некоторых популярных операционных системах стало причиной эпидемий вирусов и массового распространения всевозможных «троянских» программ, делающих на компьютере что-то такое, что напрямую противоречит интересам его владельца, но при этом отвечает интересам «хозяина» такой программы. Кроме того, при отсутствии разграничения полномочий зачастую ошибка в одной из программ, пусть и не злонамеренная, способна привести к уничтожению всей системы вместе с важными пользовательскими данными.

Отметим, что в системах семейства Unix вот уже без малого тридцать лет никто не видел живых и реально размножающихся вирусных программ; тому есть несколько причин.

Во-первых, в мире Unix никто не работает в системе с полномочиями системного администратора. Сами администраторы заводят себе учётные записи обычных пользователей и работают под ними, пока не потребуется сделать что-то такое, на что у обычного пользователя нет полномочий. В таких случаях администраторы запускают сеанс работы с неограниченными полномочиями, выполняют нужные действия, после чего немедленно выходят из системы. Программы, запускаемые в ходе повседневной работы, полномочий администратора не получают, возможности же, предоставляемые системой простому пользователю, настолько ограничены, что делают размножение вирусов практически неосуществимым.

Во-вторых, на проектирование систем семейства Unix в большинстве случаев почти не влияют маркетинговые соображения; поэтому, например, в пользовательских интерфейсах в среде Unix нет понятия «открыть файл произвольного типа», что в некоторых случаях оказалось бы связано с запуском файла как программы. Говоря обобщённо, в unix-системах от пользователя не стараются из маркетинговых соображений скрыть технические детали, в особенности такие, от незнания которых может пострадать безопасность. В частности, если по электронной почте отправить приложенный исполняемый файл, то в большинстве клиентских программ, работающих в системах семейства Unix, не найдётся

никаких пунктов меню, кнопок и т. п., осуществляющих его запуск; в других системах современные программы такую возможность предоставляют, хотя и выдают предупреждение, а ещё несколько лет назад «случайно» (по незнанию) запустить приехавшую не пойми откуда и непонятно что делающую программу пользователь мог, не встретив ни малейших препятствий.

Кроме того, и само техническое устройство систем семейства Unix определяется в первую очередь соображениями инженерного, а не маркетингового характера, так что выбор между безопасностью и неведомым «удобством для пользователя», как правило, делается в пользу безопасности; к сожалению, в последние годы возник целый ряд исключений из этого правила. Подчеркнём, что в действительности никто толком не знает, что́ на самом деле было бы удобно пользователю; то, что пропаганда крупных коммерческих компаний называет «удобством пользователя», на самом деле представляет собой удобство для продавца по впариванию соответствующих «продуктов» покупателю. Красивые картинки, которые могут понравиться неискушённому пользователю при просмотре рекламного буклета, на выставке или на витрине магазина, совершенно не обязаны быть сколько-нибудь удобными в повседневной работе.

5.1.2. Управление внешними устройствами

Управление внешними устройствами складывается из двух основных составляющих: **абстрагирования** и **координации**. Координация доступа к устройствам необходима в мультизадачных системах, чтобы избежать конфликтов между разными задачами, которым понадобилось одно и то же устройство. Если этот момент вызывает сомнения, достаточно представить себе ситуацию, когда одной задаче надо произвести операцию чтения с диска, причём нужные ей данные находятся в его начале, а другой задаче требуется операция записи, и нужное ей место расположено в конце диска. Первая задача потребует от контроллера переместить рабочую головку диска в его начало; поскольку это требует времени, как мы знаем, задача после этого должна «заснуть» (заблокироваться) в ожидании окончания запрошенного действия. Пока первая задача спит, вторая задача обратится к тому же контроллеру и потребует переместить рабочую головку в конец. Две задачи ещё имеют какие-то шансы между собой «договориться», но если таких задач с десяток, заниматься они будут в основном «перетягиванием контроллера», а не полезной работой, и в особо неудачных случаях могут вообще испортить друг другу результаты. Чтобы такого не происходило, всю работу с внешними устройствами операционная система берёт на себя. Задачи обращаются к ней через системные вызовы с просьбами о выполнении тех или иных действий с устройствами, а система решает, в какой последовательности эти просьбы исполнять; это мы и называем координацией.

С абстрагированием дела обстоят чуть иначе, здесь определяющим является не то, что в системе много задач, а то, что в мире много

разнообразных внешних устройств. Работа с разными устройствами часто ведётся по совершенно различным принципам, их контроллеры поддерживают разные команды и разные форматы данных. Если бы программистам, создающим прикладные программы, приходилось в каждой такой программе учитывать особенности всех устройств, с которыми их программам, возможно, придётся работать, то на такую поддержку уходило бы гораздо больше времени, чем собственно на реализацию прикладной логики, и всё равно при установке очередной программы на случайно выбранный компьютер оставалась бы высокой вероятность того, что эта программа окажется несовместима с аппаратурой, установленной в этом компьютере, просто потому что авторы программы не учли существования какого-то из устройств, или даже, возможно, устройство ещё не существовало в природе, когда программисты писали эту программу.

Операционная система избавляет программистов от необходимости всё время думать о многообразии существующих устройств, а пользователей — от опасений за совместимость программ с аппаратурой их компьютеров. Частные особенности каждого устройства операционная система учитывает сама, а с точки зрения программ, выполняющихся под её управлением (то есть на уровне системных вызовов), большинство устройств выглядят значительно проще, чем на самом деле, причём многие совершенно различные устройства выглядят так, как если бы они по своей конструкции были одинаковы. Иначе говоря, задачи вместо реальных физических устройств со всеми их особенностями видят некие *упрощённые абстракции*; поэтому мы ведём речь об *абстрагировании* как о функции операционной системы из области управления внешними устройствами.

Пожалуй, самой наглядной иллюстрацией на эту тему будет абстрактное понятие *диска*. Читатели этой книги, возможно, видели пятидюймовые дискеты и наверняка встречались с дискетами трёхдюймовыми; любители самостоятельной сборки компьютеров наверняка знают, как выглядит встроенный в компьютер жёсткий диск, а некоторые, возможно, из любопытства разбирали неисправные жёсткие диски[4] и знают, что находящаяся внутри «начинка» выглядит совсем не похожей на дискеты, несмотря на то, что принцип работы у них одинаковый — хранить информацию в виде намагниченных участков поверхности, расположенных концентрическими дорожками на вращающемся дискообразном носителе. Оптические диски (CD и DVD) читатель наверняка не только видел, но и активно использовал; принцип хранения информации на них иной, здесь под воздействием лазерного луча меняются *оптиче-*

[4]Если вас заинтересовало, что́ в процессе такой разборки можно увидеть, ни в коем случае не разбирайте рабочий жёсткий диск: после нарушения герметичности корпуса он, как правило, приходит в негодность, а информация на нём теряется. Найдите для экспериментов испорченное устройство, благо это не так уж трудно.

ские свойства поверхности диска, а затем другой лазер, используя эти свойства, восстанавливает исходную информацию. Дорожка присутствует и здесь, но представляет собой не семейство концентрических окружностей, как на магнитных дисках, а одну непрерывную спираль, раскручивающуюся из центра диска к краю. Наконец, для полноты картины прибавим сюда ещё флеш-брелки, «карточки памяти»[5], используемые в портативных устройствах вроде фотоаппаратов и смартфонов, и недавно появившиеся «твердотельные накопители» (англ. *solid-state drive*, SSD), построенные на основе всё той же flash-памяти. По своему физическому устройству они вообще не похожи на диски, в них нет ничего круглого и ничего вращающегося.

На первый взгляд все перечисленные носители информации не имеют между собой ничего общего. Тем не менее, всё это многообразие устройств операционная система представляет для пользовательских задач в виде одной и той же абстракции — *диска*, то есть некоторого объекта, про который известно, что он состоит из пронумерованных *секторов* одинакового размера (обычно 512 байт каждый), хранящих некую информацию. Диски отличаются друг от друга, во-первых, количеством таких секторов; во-вторых, тем, возможно ли в них осуществлять запись данных или же данные можно только читать; и, в-третьих, можно ли ожидать внезапного появления или исчезновения такого диска во время работы, то есть, грубо говоря, может ли пользователь (физически) неожиданно выдернуть носитель из компьютера. Большинство пользовательских задач, впрочем, на этот уровень не спускается, а работает на уровне *файлов*, у которых есть *имена* и которые можно, используя эти имена, *открывать* для чтения или записи информации, производить чтение и запись, а потом *закрывать*. При работе с файлами задача может никак не учитывать (и, как правило, не учитывает) никакие особенности дисков, на которых эти файлы расположены, особенно если говорить о физической конструкции внешних запоминающих устройств: чтение файла с оптического диска и с флеш-брелка с точки зрения пользовательской задачи выглядит абсолютно одинаково.

5.1.3. Границы зоны ответственности ОС

Итак, задачи, решаемые операционными системами, покрываются следующим списком:

- **управление пользовательскими задачами:**
  - запуск и останов процессов;

[5] Слово «память» здесь вносит изрядную путаницу; помните, что формально памятью следует называть только оперативную и постоянную память компьютера, с которой процессор может работать через шину без применения контроллеров, а все flash-накопители представляют собой не память, а внешние запоминающие устройства.

- планирование времени центрального процессора;
- распределение оперативной памяти;
- организация взаимодействия между процессами;
- разграничение полномочий;

- **управление внешними устройствами:**
  - абстрагирование от особенностей устройств разных моделей;
  - координация доступа нескольких задач к одному устройству.

Ядро операционной системы занимает особое положение среди других программ. В частности, память, занятая ядром, как правило, не может быть откачана на диск, так что чем больше будет занимать памяти ядро, тем меньше *физической* памяти останется пользовательским задачам; с самими пользовательскими задачами такой проблемы не возникает, поскольку их можно в любой момент как по частям, так и целиком откачать на диск, освободив память для других задач. Кроме того, код ядра выполняется в привилегированном режиме, что накладывает весьма суровые дополнительные требования на его качество: ошибка в коде ядра может слишком дорого стоить, а обнаружить и исправить её гораздо сложнее, чем ошибку в обычной пользовательской программе.

Всё это ведёт нас к довольно простому принципу: ядро операционной системы не должно заниматься ничем таким, чем может без серьёзных потерь заниматься программа, запущенная как обычная пользовательская задача. В некоторых операционных системах ядро стараются минимизировать даже ценой потери производительности; например, драйверы[6] физических устройств могут быть перемещены из ядра в обычные программы, но это приведёт к появлению активного обмена информацией между таким драйвером и ядром, этот обмен пойдёт через системные вызовы, так что ощутимое количество времени будет потеряно из-за частого переключения контекстов между ядром и программой, реализующей драйвер. Но если вытеснение из ядра драйверов сомнительно в силу их «системной» сущности, то всё, что связано с *прикладными* задачами, то есть с непосредственным обслуживанием интересов конечного пользователя, попросту *не имеет права* находиться в ядре.

В частности, можно совершенно определённо сказать, что **операционная система не должна содержать никаких средств, направленных на организацию общения с пользователем**. Ни оконных,

[6] Драйвер — это набор подпрограмм, отвечающих за работу с конкретным физическим устройством; как правило, драйвер действует как часть ядра операционной системы. Подробное обсуждение драйверов нам предстоит позднее; если вы не уверены в своём понимании этого термина — ничего страшного, здесь глубокое понимание пока не требуется.

ни каких-либо других пользовательских интерфейсов операционная система поддерживать не должна, это прерогатива прикладных программ; именно прикладные программы должны «разговаривать» с пользователем и обращаться к операционной системе за услугами, которые нужны, чтобы исполнить желания пользователя; иначе говоря, всё, что непосредственно видит пользователь, должно быть результатом работы прикладных программ, а не системы. В идеальной ситуации *конечный пользователь вообще должен иметь возможность не видеть её*, примерно так же, как пассажир автобуса, скорее всего, догадывается о существовании двигателя, но может совершенно не представлять, как этот двигатель устроен и даже как он выглядит.

Когда речь идёт о таких «операционных системах», как Android или Mac OS X[7], в первом приближении кажется, что как раз построением оконного интерфейса эти системы и заняты, но это не совсем точно. Графические надстройки в обеих этих системах не являются частью ядра, то есть, формально говоря, ни Android, ни Mac OS X нельзя вообще называть операционными системами; в качестве операционной системы в первом случае выступает Linux, во втором — Darwin, ядро из семейства BSD. Программирование «под Android» или «под Mac OS» означает в основном использование высокоуровневых библиотек для построения графического интерфейса в соответствии с требованиями высокоуровневых надстроек, но это происходит в пользовательских задачах, а не в ядре.

5.1.4. Unix: семейство систем и образ мышления

Рассказывая историю ОС Unix (см. т. 1, §1.2), мы отметили, что сейчас этим словом обозначается не одна конкретная операционная система, а целое семейство систем, объединённых общими принципами построения. За десятилетия истории ОС Unix вокруг неё выросла целая философия; одно из наилучших и наиболее полных изложений этой философии можно найти в книге Эрика Реймонда «Искусство программирования для Unix» [4]. Мы здесь отметим только наиболее важные моменты, связанные с ответом на вопрос, что же такое Unix.

Пожалуй, главный принцип построения операционной среды ОС Unix состоит в том, что **каждая программа должна решать ровно одну задачу, и делать это хорошо**. Этот принцип в такой формулировке приписывают Дугу Макилрою (Doug McIlroy) вместе с двумя другими тезисами: программы должны работать совместно друг с другом;

[7] Официальное маркетинговое название этой системы, начиная с 2016 года — «macOS»; некоторое время до этого она называлась просто «OS X». Под названием «Mac OS X» она была представлена изначально, причём буква X соответствовала числу *десять* в римской записи, а всё название означало «десятая версия операционной системы для компьютеров Макинтош». Именно так мы и будем её называть, не обращая внимания на пляски маркетоидов.

программы должны использовать текстовые потоки как универсальное представление информации.

В качестве иллюстрации этих принципов в действии автор этих строк обычно приводит своим студентам следующий пример. В одном из зарубежных филиалов МГУ приходилось вести занятия по программированию в компьютерном классе, предназначенном для студентов нескольких разных факультетов, так что на всех машинах была установлена какая-то из версий Windows; для обеспечения занятий по программированию там же был установлен сервер под управлением ОС Linux, на который студенты заходили в режиме удалённого доступа. В какой-то момент потребовалось узнать, сколько из присутствующих студентов успешно вошли в систему.

За основу была взята программа `w` (от слова *who*), выдающая список открытых в системе сеансов работы, с указанием, какому пользователю принадлежит каждый из сеансов. Подвергнув её выдачу нескольким последовательным преобразованиям, удалось в итоге получить искомое число; каждое из преобразований выполнялось отдельной программой-фильтром[8], причём весь набор этих программ запускался на одновременное исполнение *конвейером*: данные, выведенные первой программой, поступали на ввод второй, выдача второй отдавалась на вход третьей и т. д.

Первые две строки в выдаче команды `w` — служебные (своего рода заголовок), они в дальнейшем анализе не нужны, поэтому их следует пропустить; это было сделано с помощью программы `tail` с параметрами `-n +3`, которая принимает на вход (читает из потока стандартного ввода) произвольный текст, первые несколько строк отбрасывает, а остальные печатает (в данном случае — начиная с третьей по счёту). Из полученной информации следовало выделить входные имена, поскольку всё остальное (адрес, с которого выполнен заход в систему, время работы, текущая запущенная программа и прочее) нас в данном случае не интересует. Входное имя в каждой строке выдачи `w` стоит в начале, от остальной строки оно отделено пробелом; для того, чтобы оставить в каждой строке одно только входное имя, можно воспользоваться командой `cut -d' ' -f 1` (параметр `-d` задаёт символ-разделитель, параметр `-f` — номер нужного поля). Остаётся колонка из входных имён.

Самого себя, естественно, присутствующего в системе, автор учитывать не собирался; для этого пришлось с помощью программы `grep` убрать из выдачи строки, соответствующие его собственному входному имени (`avst`): `grep -v ^avst$`. После этого в обрабатываемом тексте остались только входные имена студентов, поскольку больше в системе никто на тот момент не работал. Однако некоторые студенты уже успели создать несколько сеансов работы одновременно; это сделать очень просто, достаточно запустить несколько экземпляров программы `putty` или, если используется доступ по X-протоколу, просто открыть несколько эмуляторов терминала. Для исключения повторов полученные строки пришлось отсортировать и оставить только уникальные; это было сделано командой `sort -u`. Для получения финального результата осталось лишь посчитать оставшиеся строки с помощью `wc -l`. Полностью команда выглядела так:

```
w | tail -n +3 | cut -d' ' -f 1 | grep -v ^avst$ | sort -u | wc -l
```

[8]Напомним, что *фильтр* — это такая программа, которая читает некие данные из потока стандартного ввода и выводит результаты в поток стандартного вывода.

Текстовое представление данных (см. т. 1, §1.6.6) в некоторых случаях требует бóльшего расхода памяти и дискового пространства, нежели компактное двоичное, и заставляет программы проводить анализ текста, который сам по себе может оказаться относительно трудоёмким делом, но при этом использование текстовых данных в качестве универсального представления информации имеет совершенно очевидные преимущества. Человеку не требуются никакие специальные программы, чтобы прочитать и понять такое представление, чтобы изменить его, а при необходимости — и создать данные в таком представлении. При наличии нескольких десятков разнообразных программ, тем или иным способом преобразующих текст, таких как `tail`, `cut`, `grep`, `sort` и `wc`, задействованных в вышеприведённом примере, работа с текстовыми данными становится на порядки легче (по трудозатратам), нежели с любыми другими.

В мире Unix текстовым данным отдаётся явное предпочтение; исключения составляют разве что исполняемые файлы, основанные на машинном коде, а также цифровая информация, полученная измерением исходно аналоговых явлений — фотографические изображения, видео- и звуковую информацию теоретически тоже можно перевести в текст, но человек с этим непосредственно работать всё равно не сможет, а места итоговый файл займёт в несколько раз больше. Следует подчеркнуть, что сказанное не распространяется на искусственно созданные изображения (рисунки, диаграммы и т. п.), на музыку, синтезированную в виде программы для виртуального музыкального инструмента (в противоположность записанной путём оцифровки сигнала с микрофонов), а также на компьютерную анимацию — в этих случаях текстовое представление остаётся оправданным и предпочтительным. Кроме оцифрованной аналоговой информации и машинного кода можно назвать ещё два случая использования нетекстовых данных: данные, запакованные архиваторами для уменьшения хранимого объёма, и данные, зашифрованные разнообразными криптографическими средствами; но на этом всё. Если обрабатываемая информация не попадает ни в одну из четырёх перечисленных категорий, в мире Unix она, скорее всего, будет представлена в виде текста.

Отметим, что многие протоколы, используемые в сети Интернет, также основаны на текстовых потоках — именно так работают, в частности, электронная почта и Web; справедливости ради надо сказать, что бинарные протоколы в Интернете тоже часто встречаются, самый популярный пример — система DNS.

Ещё один принцип, которому следует большинство программ в ОС Unix, выражен фразой «**молчание — золото**». В случае, если работа проходит успешно, программы выдают лишь ту информацию, которая от них требуется, и никоим образом не больше. Читатель мог уже заметить, что ассемблер NASM и компилятор gcc в случае успеха завер-

шаются молча, не выдавая на терминал ни единой буквы; к сожалению, этого нельзя сказать о Free Pascal, создатели которого, по-видимому, не разделяют философию Unix.

Говоря об особенностях общения ОС Unix с пользователем, можно выделить ещё два ключевых момента. Во-первых, пользовательский интерфейс обычно строится в предположении, что пользователь знает, что делает. Большинство программ в ОС Unix не пытается делать вид, что они умнее пользователя или лучше знают, как надо работать; любой отданный пользователем приказ просто молча выполняется. Во-вторых, к любому функционалу, имеющемуся в системе, возможен доступ через команды командной строки; альтернативные интерфейсы, такие как графические оболочки или полноэкранные терминальные программы, могут существовать, но они не заменяют командную строку и не вытесняют её, они лишь предоставляют пользователю альтернативу. Во многих случаях альтернативные интерфейсы реализованы через обращение к утилитам командной строки.

Средства графического пользовательского интерфейса в ОС Unix проектируются по принципу «**метод, а не политика**». Это означает, что средства работы с графикой дают приложениям возможность создания графических интерфейсов без уточнения, как конкретно такие интерфейсы должны выглядеть и функционировать. В результате пользователь может, например, выбрать любой из нескольких десятков существующих оконных менеджеров, отвечающих за оформление окошек (рамок, заголовков и прочего), полностью изменив внешний вид своего рабочего экрана; авторы этих оконных менеджеров не ограничены в построении внешнего вида никакими правилами, навязанными со стороны.

Ещё один принцип ОС Unix можно выразить фразой «**всё есть файл**»; имена файлов в файловой системе — это самый популярный способ именования не только файлов как таковых, но и периферийных устройств (а равно и так называемых ***виртуальных устройств***, которые выглядят с точки зрения пользователя как устройства, но физически не существуют, их работу эмулирует ядро операционной системы), каналов связи программ друг с другом. Более того, через привычный файловый интерфейс в современных системах можно «добраться» до настройки режимов работы многих подсистем ядра, узнать об имеющихся в системе процессах и т. п. (это позволяют делать так называемые *искусственные* файловые системы; к ним мы вернёмся в заключительной части тома). К сожалению, из этого принципа тоже есть исключения: например, файловому принципу именования объектов не соответствуют объекты, относящиеся к так называемому System V IPC; в ОС Linux отсутствуют файлы устройств, отображающие сетевые карты, и так далее; но разнообразные девиации, даже многочисленные, не отменяют общего принципа.

Философия Unix, естественно, распространяется не только на программы, выполняемые под управлением операционной системы, но и на неё саму, то есть на ядро и в особенности на набор его системных вызовов. Как мы увидим позже, большинство системных вызовов обходится очень небольшим количеством параметров: многие системные вызовы вообще не принимают параметров (`getpid`, `fork` и прочие), часто встречаются системные вызовы с одним (`chdir`, `alarm`, `wait`, `close`), двумя (`signal`, `kill`, `getcwd`, `dup2`) или тремя (`read`, `write`, `open`, `execve`) параметрами. Реже встречаются вызовы с четырьмя параметрами (`recv`, `send`, `wait4`) и пятью (`select`, `recvfrom`, `sendto`), очень редко — с шестью (`mmap`, `pselect`); системных вызовов с бóльшим числом параметров в ОС Unix нет. Сложные действия обычно выполняются последовательным обращением к нескольким разным системным вызовам.

Также следует отметить широкое использование в интерфейсе системных вызовов ОС Unix универсальной парадигмы «**всё есть поток байтов**»: именно в виде таких потоков представлены ввод информации из файла и вывод информации в файл, чтение с клавиатуры и выдача на терминал, передача данных другому процессу через канал, во многих случаях — обмен данными по компьютерной сети. Как мы увидим позже, все так называемые ***потоки ввода-вывода*** делятся на *позиционируемые* (обычные файлы, в которых текущая рабочая позиция может быть в любой момент изменена) и *непозиционируемые* — все остальные, где байты передаются один за другим и на последовательность их передачи или приёма никак нельзя повлиять. В эту несложную модель укладываются почти все случаи приёма и передачи данных, за немногочисленными исключениями, которые обусловлены техническими причинами (например, обмен дейтаграммами по компьютерной сети не представляется в виде потока, потому что имеет совершенно иную природу).

Общий подход к созданию программного обеспечения, принятый в ОС Unix, можно выразить одним лозунгом, который, как считается, изначально появился на американском военно-морском флоте: «***Keep it simple, Stupid!***», сокращённо *KISS*; русскоязычные программисты часто называют этот лозунг «принципом поцелуя», буквально переводя слово «*kiss*» на русский.

Наконец, в ОС Unix имеются определённые традиции, связанные с разработкой и распространением программ. Как правило, программы создаются ***переносимыми***; это означает, что программа, написанная под одну Unix-систему на одной аппаратной архитектуре, может быть (во всяком случае, если она написана грамотно) без изменения исходных текстов откомпилирована на любой другой Unix-системе и другой аппаратной платформе. При этом конвенции системных вызовов изменяются от системы к системе, а машинный код, естественно, привязан к конкретной аппаратной платформе, так что двоичные исполняемые

файлы свойством переносимости не обладают (и не могут им обладать). Как следствие, основным способом распространения программ в ОС Unix является передача их исходных текстов. Большинство программ, включая и ядра самих систем, распространяются *свободно*, то есть условия лицензии таких программ разрешают всем желающим копировать эти программы, передавать их другим лицам, изучать, модифицировать и распространять модифицированные версии, при этом не требуется кому-либо за это платить. Большинство дистрибутивов того же Linux изначально содержит тысячи полезных программ, с помощью которых можно решать практически любые повседневные задачи, и всё это может быть совершенно свободно скачано из Интернета.

Свободное распространение программ играет чрезвычайно важную роль в культуре Unix, при этом неискушённые люди часто путают понятия «бесплатная программа», «программа с открытым исходным кодом» (open source) и «свободная программа». Между тем это совершенно разные понятия. В мире Windows встречаются программы, которые можно назвать бесплатными (причём зачастую с большой натяжкой), но никак нельзя считать ни открытыми, ни тем более свободными. Наиболее распространённое определение ***свободного программного обеспечения*** (free software) предложено Ричардом Столлманом и Фондом свободного программного обеспечения; согласно этому определению, любому пользователю программы принадлежат четыре фундаментальные свободы:

0. свобода запускать программу как будет угодно пользователю, для любых целей;
1. свобода изучать, как программа работает, и изменять её сообразно со своими потребностями; для этого необходима доступность исходных текстов;
2. свобода распространять копии, чтобы можно было помочь своему соседу;
3. свобода распространять (передавать другим людям) модифицированные версии программы, давая тем самым возможность всему сообществу извлекать пользу из этих модификаций; для этого также необходима доступность исходных текстов.

Сторонники концепции свободного программного обеспечения отстаивают эти свободы в качестве неотъемлемого права каждого пользователя и составляют тексты своих лицензий таким образом, чтобы лишить пользователей этих свобод было как можно труднее. В частности, одна из самых популярных лицензий на программное обеспечение, GNU GPL, фиксируя все четыре перечисленные свободы, при этом запрещает распространение копий (как точных, так и модифицированных), а равно и производных работ под какими-либо иными лицензиями, кроме GNU GPL. Самим фактом использования любой программы, распространяемой под GNU GPL, пользователь соглашается с условиями этой лицензии. При этом производной работой считается любая программа, в которую вошёл какой-либо фрагмент кода, распространяемого под GNU GPL, так что если вы хотите заимствовать такой фрагмент для своей программы, то и свою программу вы сможете распространять только под GNU GPL — либо не распространять вовсе.

Вопреки расхожему мифу, GNU GPL не требует *бесплатности* распространения как таковой. Напротив, сам Столлман неоднократно разъяснял, что каждый

волен при желании брать деньги в обмен на программу, распространяемую под GNU GPL. Иной вопрос, что коль скоро кто-то купил у вас программу, распространяемую под GNU GPL, этот кто-то получает все права, предусмотренные GNU GPL, и вы, согласно условиям лицензии, не имеете права даже просить его воздержаться от использования полученных таким образом прав. Например, если кто-то купит у вас за миллион долларов программу, на которую распространяется лицензия GNU GPL, а затем выложит её в Интернет в режиме свободного доступа, вы никак не сможете (и не будете вправе!) на это повлиять; любые «дополнительные соглашения», ограничивающие свободу, данную лицензией GNU GPL, являются нарушением этой лицензии.

Отметим, что понятие программного обеспечения с открытым кодом отличается от понятия свободного программного обеспечения: для свободного программного обеспечения открытость исходного кода необходима, но обратное неверно. Во-первых, существуют другие лицензии на программное обеспечение, опубликованное в виде исходных кодов; такие лицензии могут не заботиться, как GNU GPL, о сохранении свобод для *всех* пользователей. Например, так называемая BSD license позволяет делать с программой что угодно, в том числе и распространять производные работы без их исходных текстов; получатель таких программ обладает всеми вышеперечисленными свободами, но не обязан заботиться о том, чтобы ими обладал кто-то ещё. С другой стороны, известны примеры программного обеспечения, которое хотя и распространяется в исходных текстах, но при этом не является свободным ни с какой точки зрения: лицензионный договор не позволяет распространять его дальше ни в виде точных копий, ни тем более в виде копий модифицированных.

5.1.5. Замечания о системе X Window

Мы уже отмечали, что ОС Unix как таковая не претерпела существенных архитектурных изменений даже при таком серьёзном шаге, как введение графических оболочек, поскольку графические оболочки, не будучи частью операционной системы, не смогли оказать на неё заметного влияния. Давайте попробуем разобраться, как это стало возможным.

Средства, используемые в большинстве Unix-систем для работы с графическими (оконными) приложениями, получили общее название ***X Window System***. В качестве основного определяющего свойства этих средств можно назвать то, что реализуются они целиком в виде обычных пользовательских программ; поддержка со стороны ядра операционной системы ограничивается специальными средствами для доступа к видеокарте, но эти средства никак не учитывают особенности X Window System и могут использоваться любыми другими графическими оболочками, если таковые появятся. Следует отметить, что в проприетарных системах на основе Unix действительно часто встречаются альтернативные графические оболочки; к таким системам относятся, например, упоминавшиеся выше Mac OS X и Android. Созданные для них графические средства не имеют никакого отношения к X Window,

за исключением разве что основного принципа: они тоже реализованы в виде пользовательских программ и не затрагивают ядро.

Возвращаясь к X Window System, отметим, что приложение, использующее оконный графический интерфейс, продолжает при этом быть обычным Unix-процессом. В частности, как и у любого процесса, у оконного приложения есть параметры командной строки, а также дескрипторы стандартного ввода, вывода и сообщений об ошибках (`stdin`, `stdout` и `stderr`). Чтобы отобразить окно, приложение должно установить соединение с системой X Window и отправить запрос в соответствии с определёнными соглашениями, известными как ***X-протокол***.

«По ту сторону» соединения находится программа, называемая X-сервером. Именно эта программа производит непосредственное отображение графических объектов на экране пользователя. По собственной инициативе она ничего не рисует; чтобы на экране что-то появилось, необходим запрос на отрисовку того или иного изображения. Таким образом, в основном отображающая программа выполняет действия в ответ на запросы других программ (клиентов), что оправдывает название «X-сервер». Услугой (сервисом), которую оказывает клиентам этот сервер, оказывается отрисовка изображений на экране. В некоторых случаях X-сервер сам проявляет инициативу при общении с клиентом. Это происходит при возникновении тех или иных событий, относящихся к области экрана, в которой отображено окно клиента; к таким событиям относятся нажатия клавиш на клавиатуре, движения и щелчки мыши, перемещения окон, требующие полной или частичной повторной отрисовки изображения в окне, и т. д.

Система спроектирована так, что соединение с X-сервером можно установить, используя как локальные средства взаимодействия внутри одного компьютера, так и инструменты работы через компьютерную сеть. С точки зрения пользователя это значит, что при работе с X Window можно запускать оконные приложения на удалённых компьютерах, при этом графические окна этих приложений видеть локально, то есть на своём экране.

Забегая вперёд, скажем, что соединение с X-сервером осуществляется через потоковый сокет (сокет типа `SOCK_STREAM`), причём обычно X-сервер заводит слушающие сокеты как в семействе `AF_UNIX` для локальных подключений, так и в семействах `AF_INET` и `AF_INET6`, что позволяет связываться с X-сервером по сети. После установления соединения такие сокеты выглядят совершенно одинаково — как двунаправленные каналы последовательной передачи данных. Во время работы X-клиенту и X-серверу безразлично, какую природу имеет установленное между ними соединение. Подробности отложим до следующей части нашей книги.

Важно отметить, что X-сервер, как и X-клиенты, представляет собой обыкновенный процесс, обычно, правда, имеющий определённые привилегии для доступа к соответствующему оборудованию (видеокарте). Поддержка графического интерфейса со стороны ядра ограничивается

предоставлением доступа к видеокарте, например, путем отображения видеопамяти на виртуальное адресное пространство X-сервера. Это позволяет X-серверу не быть частью операционной системы.

Интересно, что X-сервер не обязан для отображения графики использовать доступ к видеокарте; он, собственно говоря, обязан поддерживать X-протокол — и всё. Так, широко известен X-сервер `Xvfb`, который построенное изображение нигде не показывает, а просто записывает в файл. Кроме того, существуют X-сервера, работающие в системах семейства Windows и позволяющие пользователю рабочей станции под MS Windows запускать удалённо (на Unix-машине) оконные приложения и взаимодействовать с ними; наиболее широкое распространение среди них получили Xming и Cygwin/X.

Если запустить X-сервер без обычной прикладной обвески (это делается командой `X`), мы увидим пустой экран с курсором мыши, напоминающим очень жирную букву X, и фоном, выглядящим, как увеличенный рисунок грубой ткани — это стандартное фоновое изображение X-сервера, которое обычно прикладные программы тут же меняют на другое.

Если вы захотите провести описываемый эксперимент самостоятельно, убедитесь, что на машине в это время не запущены никакие другие X-сервера. Если они всё-таки запущены, то уберите их или прикажите запускаемому X-серверу работать вторым дисплеем машины (используйте команду «`X :1`» или «`X :1.0`»).

Всё, что мы можем сделать с запущенным таким вот образом сервером — это подвигать курсор с помощью мыши. Чтобы получить более интересные результаты, нужно запустить хотя бы одну прикладную программу. Для этого следует, нажав комбинацию `Ctrl-Alt-F1`[9], вернуться в текстовую консоль, с которой мы запустили X-сервер, нажатием `Ctrl-Z` и командой `bg` убрать работающую программу `X` в фоновый режим. Теперь мы можем запустить какую-нибудь прикладную программу, например `xterm`. Поскольку мы не воспользовались обычной «обвеской» для запуска X-сервера, нам придётся самостоятельно указать программе, с каким X-сервером пытаться установить соединение. С учётом этого команда (при использовании Bourne Shell) будет выглядеть так:

```
DISPLAY=:0.0 xterm
```

Если вы запустили свой экспериментальный экземпляр X-сервера одновременно с другим работающим X-сервером, вместо «`:0.0`» укажите соответствующий идентификатор дисплея, например «`:1.0`».

Вернёмся теперь к нашему X-серверу; в зависимости от обстоятельств нам для этого понадобится комбинация клавиш `Alt-F7`, `Alt-F8`

[9]Предполагается, что эксперимент проводится на машине под управлением ОС Linux или FreeBSD, а запуск произведён с первой виртуальной консоли.

и т. п. Если всё прошло успешно, мы увидим в левом верхнем углу экрана окно программы `xterm` и, переместив в него курсор мыши, сможем убедиться, что командный интерпретатор в нём работает. Однако целью нашего эксперимента было не это. Главный факт, который сейчас можно констатировать — это полное отсутствие у окна каких-либо элементов оформления. Нет ни рамки, ни заголовка, ни привычных кнопок в уголках окна, служащих для свёртывания в иконку, развёртывания на весь экран и закрытия — ничего! В такой ситуации мы не можем, например, переместить имеющееся окошко или изменить его размер.

Итак, наш нехитрый эксперимент дал нам возможность узнать, что в системе X Window за обрамление окон не отвечают ни X-сервер, ни клиентское приложение. Так сделано не случайно. Возложив ответственность за декор окон на X-сервер, мы навязали бы один и тот же внешний вид (и возможности управления) всем пользователям данного сервера. Несмотря на то, что одна известная компания именно так и поступает с пользователями своих операционных систем, такой подход трудно назвать правильным. Если же заставить каждого X-клиента отвечать за стандартные элементы его собственного окна, это резко увеличит сложность оконных приложений, причём изрядная часть их функциональности будет в программах дублироваться. Кроме того, это снизит возможности пользователя по выбору удобного ему декора окон. В системе X Window за стандартные элементы оконного интерфейса отвечают специальные программы, называемые ***оконными менеджерами***.

Продолжая наш эксперимент, запустим какой-нибудь простой оконный менеджер, например `twm`, обычно входящий в поставку X Window. Это можно сделать не покидая X-сервер, ведь у нас уже запущена программа `xterm`, и в её окне работает интерпретатор командной строки. Итак, переместите курсор мыши в область окна и дайте команду `twm`. Если программы с таким именем в вашей системе не нашлось, вы можете её поставить средствами вашего пакетного менеджера, а можете воспользоваться каким-то ещё из «лёгких» оконных менеджеров — `fvwm2`, `icewm` и т. д.; для нужд нашего эксперимента подходит любой «оконник» из тех, что не пытаются притворяться «рабочими столами» (Desktop Environment).

После запуска оконного менеджера вокруг всех имеющихся окон появятся элементы оформления. Теперь окна можно двигать, закрывать, менять их размер и т. д. Для получения более интересного эффекта можно сначала запустить одно-два небольших оконных приложения, например, `xeyes`, и только после этого запускать `twm`. Теперь попробуйте подвигать окна по экрану, после чего убейте процесс `twm`; элементы декора со всех окон тут же исчезнут, но сами окна никуда не денутся. После этого можно снова запустить `twm` или другой оконный менеджер.

Оконный менеджер в X Window является обычным процессом, с точки зрения самой системы ничем не отличающимся от обычного приложения. С X-сервером оконный менеджер общается с помощью того же X-протокола, что и остальные клиенты. Популярные оболочки `KDE` и `Gnome` представляют собой по сути не что иное, как оконные менеджеры, снабжённые, правда, развитой дополнительной функциональностью.

Как уже говорилось, оконное приложение может выполняться на той же машине, где запущен X-сервер, а может и на совсем другой, связываясь с X-сервером по сети. Пользуясь этим свойством X-протокола, можно создавать специализированные компьютеры, единственным назначением которых будет поддержка X-сервера. Такие компьютеры называются ***сетевыми X-терминалами***. При работе с X-терминалом все пользовательские программы выполняются где-то в другом месте, как правило, на специально предназначенной для этого мощной машине. Такую машину обычно называют ***сервером приложений***. Отметим, что сервер приложений может вообще не иметь собственных устройств отображения графической информации, что не мешает ему выполнять графические программы.

При работе с X-терминалом пользователю нужен доступ к его домашнему каталогу и другим ресурсам, которые находятся, естественно, на удалённой машине (на самом сервере приложений, на файловом сервере и т. д.), ведь X-терминал никаких задач, кроме отображения графики (то есть выполнения программы X-сервера), не решает. Как следствие, требуется обеспечить возможность аутентификации пользователя на удалённой машине, создание сеанса работы, включающего, например, программу оконного менеджера, которая уже управляет X-сервером и позволяет запускать программы пользователя. Для проведения такой удалённой аутентификации система X Window предусматривает специальные средства. На сервере приложений запускается процесс-демон, называемый обычно `xdm` (X Display Manager). Запущенный на терминале пользователя X-сервер обращается к программе `xdm` с использованием протокола XDMCP (X Display Manager Control Protocol). Функционирование `xdm` несколько напоминает традиционную схему `getty`: на графический экран пользователя выдается приглашение к вводу входного имени и пароля, после чего (уже с правами аутентифицировавшегося пользователя) запускается головной процесс нового сеанса. В традиционной схеме работы текстовых терминалов таким главным процессом выступает интерпретатор командной строки, а для сеансов работы с X-терминалом в качестве главного процесса запускается обычно либо оконный менеджер, либо (чаще) некий командный файл, который производит подготовительные действия, а затем уже запускает оконный менеджер.

Со схемой взаимодействия X-терминала, сервера приложений, программы `xdm` и пользовательских программ (X-клиентов) связана, к

сожалению, изрядная терминологическая путаница. X-терминал — это *клиентская* рабочая станция, в конечном счёте обращающаяся к *серверной* (обслуживающей) машине — серверу приложений. Более того, на сервере приложений запускается программа `xdm`, представляющая собой *сервер* протокола XDMCP. И в то же время программа, выполняющаяся на X-терминале (клиентской машине!), называется ***X-сервером***, а пользовательские программы, выполняющиеся на сервере приложений (!), называются ***X-клиентами***.

Дело в том, что с точки зрения X Window System сервером, предоставляющим *услугу по отображению графических объектов*, является как раз X-терминал, а обращающиеся за такой услугой программы (пользовательские приложения) оказываются, соответственно, *клиентами*. Попросту говоря, используемая терминология зависит от уровня, на котором мы рассматриваем участников взаимодействия. На уровне X Window System X-терминал является сервером, на уровне пользовательских услуг — безусловно, клиентом.

Достоинством схемы с использованием нескольких мощных серверов приложений и множества X-терминалов является крайняя дешевизна администрирования и обслуживания такой сети. X-терминалы обычно не имеют дисковых подсистем; некоторые из них способны обходиться без вентиляторов за счёт использования сравнительно медленных процессоров. Как следствие, в них попросту нечему ломаться. Никакой настройки большинство X-терминалов не требует, все необходимые параметры они получают при подключении к сети. В обслуживании и администрировании нуждаются только серверные машины. В организации, имеющей несколько сот рабочих мест, в качестве серверов приложений приходится использовать мощные и дорогостоящие компьютеры, однако вложения в них быстро окупаются за счёт экономии расходов на текущий ремонт и прочее обслуживание пользовательских рабочих станций. Такие расходы при использовании X-терминала могут быть в десятки раз ниже, чем при использовании обычных персональных компьютеров.

До недавнего времени функциональность X-терминалов ограничивалась только передачей в одну сторону событий от клавиатуры и мыши, а в другую — графической информации. Для обычной работы с компьютером в офисе этого было достаточно, но в современных условиях тяжело говорить о полноценной работе без воспроизведения звука. Этот вопрос при работе с X-терминалами также легко решается с помощью специальных программ, таких как JACK и PulseAudio; большинство современных приложений для Unix-систем, предполагающих работу со звуком, поддерживают взаимодействие с одной из этих программ или с обеими. С точки зрения пользователя это означает возможность услышать средствами локальной машины (в том числе X-терминала)

звук, воспроизводимый программой, запущенной на удалённой машине (сервере приложений).

5.2. Интерфейс ядра — системные вызовы

5.2.1. Вызовы и их обёртки

Как мы уже знаем, ядро операционной системы предоставляет свои услуги пользовательским программам через интерфейс системных вызовов; программируя на языке ассемблера, мы видели этот интерфейс в действии, инициируя вручную так называемые программные прерывания, характерные для процессоров семейства i386, имеющих 32-битную архитектуру. Другие процессоры не используют концепцию программных прерываний, и даже на 64-битных «наследниках» архитектуры i386 команда `int` больше не используется для обращения к ядру операционной системы, поскольку появились более «правильные» механизмы, но общая идея остаётся прежней: пользовательская программа для выполнения системного вызова должна произвести некие (зависящие от процессора) действия, в результате которых ядро операционной системы получит управление и сможет узнать, чего от него хочет пользовательская программа.

Обсуждая язык Си, мы убедились, что стандартная библиотека предоставляет программисту обёртки системных вызовов в виде обычных (по внешнему виду) функций языка Си, которые называются так же, как и сами системные вызовы. Кроме того, мы узнали, что в принципе, применив фрагменты на языке ассемблера, мы можем обойтись без библиотечных «обёрток» и взаимодействовать с ядром напрямую, но так обычно всё же не делают. Интерфейсы системных вызовов для ядер различных систем друг от друга заметно отличаются — например, мы видели, что для ядер Linux и FreeBSD некоторые системные вызовы имеют различные номера, плюс к тому один и тот же системный вызов (например, `open`) может в разных системах требовать различных значений констант. Интересно, что при переходе к 64-битной архитектуре ядро Linux сменило номера всех системных вызовов; зачем это было сделано, остаётся не вполне понятным.

Так или иначе, библиотека языка Си сглаживает различия между интерфейсами различных ядер, делая программы переносимыми. Дело здесь не только и не столько в проблемах нумерации и константах; часто бывает так, что некая функция, будучи системным вызовом в одной системе, оказывается в другой системе простой библиотечной функцией, эмулирующей соответствующую функциональность через другие системные вызовы.

Обёртки системных вызовов в системах семейства Unix используют общий механизм обработки ошибок. Как правило, если системный

вызов в принципе может завершиться ошибочно, его обёртка возвращает значение `-1`, при этом в глобальную переменную `errno` заносится код ошибки. Этот код можно использовать напрямую, сравнивая его с макроконстантами, определёнными той же библиотекой, такими как `ENOENT`, `EINVAL`, `EPERM`, `EACCESS` и т. п.; можно также воспользоваться библиотечными функциями, такими как `strerror` (возвращает текстовое описание ошибки по её коду) или `perror` (выдаёт сообщение о последней ошибке в диагностический поток вывода). В документации на каждый системный вызов перечисляются все коды ошибок, которые может установить данный системный вызов.

Полезно знать, что с массовым распространением многопоточного программирования `errno` перестала быть, собственно говоря, переменной; теперь это, как правило, макрос, который вызывает некую функцию, эта функция возвращает указатель на целочисленную переменную, макрос разыменовывает полученное значение, превращая его в переменную в том смысле, что результирующее выражение по-прежнему можно использовать как справа, так и слева от присваивания. Все эти пляски и приседания сделаны затем, чтобы у каждого независимого потока управления был свой собственный экземпляр `errno`. В целом со словом `errno` вы можете работать так же, как и с любой другой целочисленной переменной — обращаться к ней, присваивать ей значение, брать её адрес; единственное, чего точно не следует делать — это пытаться назвать таким именем свою собственную переменную или другой объект, поскольку при этом прелесть макропроцессора языка Си проявится во всей красе, ваша программа, разумеется, не пройдёт компиляцию, причём выданная компилятором диагностика будет такова, что разобраться в ней окажется не слишком просто даже опытным программистам.

5.2.2. О разграничении полномочий

Как мы знаем, возможности пользовательской задачи (самой по себе) ничтожны: всё, что она может без обращения к системе — это преобразовывать данные в отведённой ей памяти. Возможности ядра операционной системы, напротив, безграничны — или, точнее, ограничены только возможностями аппаратуры компьютера. Обслуживая системные вызовы, ядро задействует часть своего всемогущества, чтобы исполнить просьбы, поступившие от пользовательских задач; без этого выполнение пользовательских задач не имело бы смысла, ведь все результаты работы так и оставались бы в отведённой им памяти, где их никто не видит.

С другой стороны, как мы уже отмечали, одна из важнейших функций операционной системы — это *разграничение полномочий*. Если бы ядро было готово исполнить *любую* просьбу пользовательской задачи, то само ядро как отдельная сущность было бы не нужно, оно не смогло бы достигнуть большей части поставленных перед ним целей. Можно придумать множество потенциальных просьб со стороны пользовательских задач, которые ядро не станет выполнять, сколько его

ни проси. Так, никакая пользовательская задача не может получить доступ к памяти ядра, перепрограммировать защиту памяти, запретить прерывания или перенастроить их обработку; для таких просьб ядро попросту не предусматривает системных вызовов. Но даже если то или иное действие по инициативе пользовательской задачи может быть выполнено, это не означает, что оно может быть выполнено по просьбе *любой* пользовательской задачи. Технически именно в этом и состоит разграничение полномочий в системе: прежде чем исполнить тот или иной запрос пользовательской задачи (т. е. системный вызов), ядро операционной системы проверяет, *имеет ли право* эта задача требовать выполнения такого запроса.

Можно привести примеры запросов к ядру, на которые имеет право вообще любая задача. Так, задача всегда может потребовать её немедленно завершить: знакомый нам по предыдущему тому системный вызов `_exit` всегда отрабатывает успешно. Также любая задача имеет право узнать текущее время (значение системных часов), получить целый ряд сведений о самой себе, закрыть любой из своих открытых потоков ввода-вывода и т. п. Во всех этих случаях ядро исполняет просьбу задачи без каких-либо проверок допустимости такой просьбы.

Во многих случаях ядро соглашается исполнить просьбу задачи лишь при условии, что задача работает от имени конкретного пользователя или же от имени одного из пользователей, входящих в ту или иную *группу*. К примеру, задача может обратиться к ядру с просьбой о прекращении выполнения *другой* задачи. Такая просьба будет удовлетворена только в случае, если у обеих задач один и тот же владелец; иначе говоря, пользователь имеет право снять свою собственную задачу, но не чужую. Кроме того, именно принадлежность задачи конкретному пользователю лежит в основе разрешения или запрещения тех или иных операций с файлами в файловой системе: как мы знаем из вводной части первого тома, у файла тоже есть владелец, а также *права доступа*, устанавливаемые отдельно для владельца, группы пользователей и всех остальных пользователей. Когда пользовательская задача пытается открыть файл с помощью системного вызова `open`, ядро проверяет, кто является владельцем задачи и достаточно ли у него полномочий в отношении данного файла, чтобы открыть этот файл в том режиме, в котором просит задача; если полномочий не хватает, система отказывается открывать файл, вызов `open` возвращает ошибку. Аналогичные проверки выполняются и в других случаях; например, изменить права доступа к файлу может только его владелец.

Среди всех пользователей системы выделяется один особый пользователь, называемый системным администратором или ***суперпользователем*** (англ. *superuser*); в системах семейства Unix этот пользователь

по традиции называется `root`[10]. На задачи, выполняемые от имени суперпользователя, ограничения по владельцу не действуют. В частности, такие задачи могут открывать любые файлы в системе как на чтение, так и на запись, не обращая внимания на права доступа; также они могут потребовать от ядра приостановить или вообще уничтожить любую другую задачу, кому бы она ни принадлежала, и т. д. Кроме того, существует множество действий, которые ядро готово исполнить только для суперпользователя, т. е. если соответствующий системный вызов выполнила задача, имеющая полномочия суперпользователя. Простейший пример такого действия — изменение текущего времени в системе, но, конечно, этим дело не ограничивается; позже мы столкнёмся с целым рядом системных вызовов, которые доступны только пользователю `root` (его задачам).

Наконец, в системе могут быть установлены разнообразные количественные ограничения, также во многих случаях связанные с пользователями, хотя и не всегда. Например, можно ограничить количество оперативной (на самом деле виртуальной) памяти, которую имеет право затребовать отдельно взятая задача, или количество одновременно открытых файлов (потоков ввода-вывода); можно также ограничить количество одновременно работающих задач для отдельно взятого пользователя и так далее. Бóльшая часть таких ограничений тоже реализована на уровне системных вызовов: ядро отказывается исполнять просьбу, поступившую от пользовательской задачи, если исполнение этой просьбы привело бы к превышению какого-либо из установленных количественных ограничений.

В системах семейства Unix реализация разграничения полномочий основана на концепции ***идентификатора пользователя*** и ***идентификатора группы пользователей*** (англ. *user identifier* (`uid`), *group idendifier* (`gid`)). Эти идентификаторы представляют собой неотрицательные целые числа; в некоторых системах разрядность параметров `uid` и `gid` составляет 16 бит, в большинстве современных систем — 32 бита, хотя обычно в системе не требуется заводить так много (свыше 65 тысяч) пользователей.

Следует сразу же оговориться, что под словом «пользователь» далеко не всегда подразумевается живой человек; для того, что в системах семейства Unix традиционно именуют «пользователем» (англ. *user*), в современных условиях точнее подходит другой термин — английское слово *account*, которое на русский язык в этом контексте лучше всего переводить словосочетанием *учётная запись*. В частности, никто не мешает пользователю-человеку завести на компьютере несколько учётных записей для разных видов работы; с точки зрения системы это

[10] Основное значение этого английского слова — *корень* или *основа*; как это часто бывает, слово *root* в английском языке имеет много значений и переносных смыслов; точно перевести, чтó в данном случае имели в виду создатели Unix, затруднительно.

ничем не будет отличаться от ситуации, когда с компьютером работают разные люди. Более того, многие учётные записи (как правило, даже большинство) вообще не имеют ничего общего с людьми и заводятся для изоляции в системе тех или иных программ, то есть чтобы при запуске некоторой программы она получала только строго определённые возможности; часто такие программы запускаются автоматически, без участия пользователя-человека. Учётные записи, не соответствующие реальным пользователям системы, по-английски иногда называют *dummy users* или *dummy accounts*. Несмотря на всё сказанное, мы будем по-прежнему употреблять термин «пользователь», но при этом важно помнить, что пользователь-человек и «пользователь» с точки зрения ядра системы — это совершенно разные сущности; по большому счёту, с точки зрения ядра «пользователь» — это просто номер, тот самый `uid`.

Идентификатор пользователя, равный нулю, имеет в ОС Unix особый смысл: это идентификатор суперпользователя — вышеупомянутого пользователя `root`. Как мы уже говорили, суперпользователь обладает в системе гораздо более широкими возможностями, нежели обычный пользователь; ядро отличает задачи, работающие от имени суперпользователя, как раз по нулевому значению их пользовательского идентификатора, а про имена пользователей ядро вообще ничего не знает, ему это не нужно.

Пользователи в системах семейства Unix могут объединяться в ***группы***, причём каждый пользователь входит минимум в одну группу, называемую *основной* для данного пользователя. В некоторых дистрибутивах Linux при создании в системе новой пользовательской учётной записи одновременно создаётся также и группа, а создаваемый пользователь становится в этой группе единственным членом; в других дистрибутивах по умолчанию создаётся одна группа для всех пользователей, обычно она так и называется *users*. Следует отметить, что группа пользователей — это самостоятельный объект, она может существовать, даже если в неё не входит ни один пользователь. С другой стороны, отдельно взятый пользователь может входить в несколько групп, но только одна из них будет для него основной; остальные группы, в которые входит пользователь, называются *дополнительными* (англ. *supplementary*).

Исходно группы были придуманы, чтобы пользователи, имеющие доступ к многопользовательской вычислительной машине, могли организовать совместную работу — например, делать те или иные файлы и директории доступными членам своей группы, но оставлять их недоступными для всех остальных. В современных условиях, когда у большинства компьютеров всего один живой пользователь, группы чаще применяются для более гибкой настройки полномочий, ассоциированных с разными учётными пользовательскими записями в системе. Если, к примеру, нужно дать возможность печатать на принтере не всем пользователям в системе, а только некоторым из них (в том числе,

возможно, лишить этой возможности часть автоматически запускаемых программ, для которых созданы отдельные учётные записи), то можно создать специальную группу, дать этой группе доступ к файлам соответствующих устройств или каналов, имеющих отношение к печати на принтере, и включить в эту группу тех и только тех пользователей системы, которым печать на принтере разрешена.

Чтобы получить более наглядное представление о пользователях и группах, вы можете посмотреть, как обстоят с ними дела в вашей собственной системе. В системах семейства Unix для описания пользователей и групп используются файлы `/etc/passwd` и `/etc/group`. Это обычные текстовые файлы, доступные на чтение всем пользователям системы, так что просмотреть их (например, с помощью команды `less`) вы можете в своём обычном сеансе работы, администраторские полномочия для этого не нужны. Каждая строка в файле `/etc/passwd` соответствует учётной записи пользователя, строка в `/etc/group` — группе пользователей. Отдельные поля в строках обоих файлов разделены двоеточиями. Файл `passwd` может выглядеть примерно так:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
```

...

```
avst:x:1000:1000:Andrey V. Stolyarov:/home/avst:/bin/bash
```

Традиционно этот файл начинается с описания пользователя `root`, за ним следуют системные учётные записи, не имеющие отношения к живым людям — упоминавшиеся выше *dummy accounts*. Свою собственную учётную запись вы, скорее всего, обнаружите в самом конце файла. Поля в каждой строке-записи означают (слева направо) входное имя пользователя (*login name*), пароль, идентификатор пользователя (`uid`), идентификатор его основной группы (`gid`), «настоящее имя» (*real name*), путь к домашней директории пользователя и, наконец, командный интерпретатор, который нужно запускать для данного пользователя при его входе в систему. Так, последняя строчка в примере выше, которая соответствует учётной записи автора книги на его личном компьютере, указывает, что входное имя для этой учётной записи — `avst`, идентификаторы (`uid` и `git`) равны `1000`, домашняя директория — `/home/avst`, командный интерпретатор — `/bin/bash`, а в качестве «настоящего имени» указано «`Andrey V. Stolyarov`» (конечно, здесь можно написать вообще что угодно, ведь система не умеет проверять документы).

Отметим, что поле, предназначенное для пароля, практически во всех системах содержит букву «`x`»; это означает, что никакого пароля

там на самом деле нет. Примерно до середины 1990-х годов пароли действительно хранили в файле `/etc/passwd` в виде хеша, т. е. в зашифрованной форме, не допускающей расшифровки; правильность введённого пароля в такой ситуации проверяется путём его зашифровки и сравнения уже зашифрованного варианта с хешем, имеющимся в системе. Практика показала, что такой подход не удовлетворяет требованиям безопасности. Дело в том, что файл `/etc/passwd` обязательно должен быть доступен на чтение всем пользователям системы — в противном случае, например, команда `ls` не сможет показать, кто является владельцем того или иного файла, и т. д. Имея хеши паролей всех пользователей системы, злоумышленник вроде бы не может их расшифровать, но зато он может *подобрать* пароли с помощью некоего оптимизированного перебора. Так, пароли длиной до шести символов включительно подбираются полным перебором; более длинные пароли можно попытаться подобрать, используя словарные слова и их комбинации с цифрами; известны и более изощрённые методы подбора. Поэтому в современных системах пароли в `/etc/passwd` не включаются ни в каком виде. В большинстве случаев пароль хранится (в виде всё того же хеша) в отдельном файле `/etc/shadow`, который доступен на чтение только суперпользователю; в некоторых системах нет и его, а пароли хранятся в специальной директории `/etc/tcb`, где для каждой учётной записи имеется отдельная поддиректория, а в ней — файл, содержащий пароль. Существуют и другие подходы.

Структура файла `/etc/group` несколько проще. Выглядит он примерно так:

```
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:avst,olga
```

...

```
avst:x:1000:
```

Полей здесь, как видим, всего четыре: имя группы, пароль (причём никто толком не знает, зачем он здесь нужен; автору никогда не встречались системы, где хоть как-то мог использоваться пароль группы), идентификатор (`gid`) и список пользователей, входящих в группу; пользователи в списке перечисляются через запятую. Отметим, что в свою *основную* группу — ту, что указана в четвёртом поле в `/etc/passwd` — пользователь входит в любом случае, этот факт не требует отражения в `/etc/group`. Любопытно, что обычно в системе присутствует группа `root` с идентификатором `0`, но, в отличие от такого же идентификатора *пользователя*, нулевой идентификатор группы никакого специального смысла не имеет.

Важно понимать, что **ядро системы ничего не знает о файлах `/etc/passwd` и `/etc/group`**. Информацию из этих файлов используют программы, работающие под управлением системы, но не сама система; с точки зрения ядра учётная запись пользователя представляет собой только `uid` (т. е. целое число) и больше ничего, группа пользователей — `gid` и больше ничего. В частности, когда пользователь входит в систему, диалог с ним ведёт одна из программ, выполняющихся с полномочиями системного администратора; эта программа запрашивает у пользователя входное имя и пароль, проверяет их, используя информацию в файлах `/etc/passwd` и `/etc/shadow`, затем присваивает себе (как задаче) соответствующие идентификаторы пользователя и основной группы, а также идентификаторы дополнительных групп (уже на основе информации из `/etc/groups`); от прав администратора программа при этом отказывается. Установив себе такие полномочия в системе, которые, согласно информации из конфигурационных файлов, положены входящему в систему пользователю, программа вместо себя запускает пользовательский сеанс работы; при входе в систему через текстовую консоль это командный интерпретатор, при входе в графическом режиме — специальная программа (лидер сеанса), например, оконный менеджер. Роль ядра в этой процедуре сводится к обработке системных вызовов, изменяющих полномочия задачи по её просьбе (естественно, такие вызовы доступны только задаче, исходно выполнявшейся с правами администратора). Позже (см. §5.4.10) мы подробно рассмотрим соответствующие системные вызовы.

5.3. Ввод-вывод и файловые системы

С вводом-выводом мы уже давно и прочно знакомы, ведь без него не обходится ни одна программа. Напомним, что к вводу-выводу относится как работа с терминалом (пресловутые «ввод с клавиатуры» и «вывод на экран»), так и чтение/запись дисковых файлов, причём мы уже успели понять, что и на стандартных потоках ввода-вывода у нас вместо клавиатуры и экрана могут оказаться текстовые файлы, а наша программа этого не заметит.

Осваивая начальные навыки программирования на примере Паскаля, мы могли себе позволить не задумываться, как в действительности устроен ввод-вывод. Добравшись до языка ассемблера, мы были вынуждены обратить внимание, что программы, работающие под управлением мультизадачных операционных систем, все свои операции ввода-вывода могут проделать только через обращения к операционной системе, называемые системными вызовами. Рассматривая ввод-вывод с помощью стандартной библиотеки языка Си, мы отметили, что, хотя высокоуровневые механизмы этой библиотеки позволяют при необходимости

производить «блочный» ввод и вывод произвольных порций данных, удобнее для этого пользоваться непосредственно системными вызовами.

Несмотря на весь наш опыт работы с файлами и потоками, многие возможности и особенности так и остались «за кадром». В этой главе мы попытаемся наверстать упущенное.

5.3.1. Знакомьтесь: файловая система

Под ***файлом*** в терминологии, связанной с компьютерами, обычно понимается некий набор хранимых на внешнем запоминающем устройстве данных, имеющий имя[11]. Подсистема операционной системы, отвечающая за хранение информации на внешних запоминающих устройствах в виде именованных файлов, называется ***файловой системой***.

Термин «*файловая система*» часто используется в совершенно ином значении, а именно для обозначения структур данных, создаваемых на внешнем запоминающем устройстве (то есть, попросту говоря, в секторах диска) с целью организации хранения на этом устройстве данных в виде именованных файлов. Такая структура данных обычно включает некий заголовок, хранящий наиболее общую информацию о файловой системе в целом (так называемый ***суперблок***), а также тем или иным способом организованное хранилище информации, какие сектора диска свободны, какие заняты, как называются файлы, хранящиеся в системе, какой набор секторов занимает каждый из файлов. Кроме того, для каждого файла файловая система может хранить разнообразную дополнительную информацию вроде типа файла, прав доступа, даты последней модификации и тому подобного.

Обычно из контекста можно однозначно определить, какая из двух «файловых систем» имеется в виду, так что путаницы такая перегрузка термина не создаёт. Тем не менее, стоит помнить о том, что под термином «файловая система» может пониматься как определённая структура данных на диске, так и набор подпрограмм в ядре операционной системы, и очевидно, что это совершенно разные вещи.

Ранние файловые системы позволяли давать файлам имена только ограниченной длины, причём эти имена находились в одном общем пространстве имён; естественно, от имён требовалась уникальность. Конечно, такой подход годится лишь до тех пор, пока файлов на одном устройстве сравнительно немного. С ростом объёма дисковых накопителей потребовалась более гибкая схема именования. Чтобы получить искомую гибкость, ввели понятие ***директории***, или ***каталога***[12].

[11] Ясно, что эта фраза не может претендовать на роль строгого определения; она дана лишь для того, чтобы зафиксировать основные нужные нам свойства файлов.

[12] В англоязычной литературе используется термин *directory* (читается «дайре́ктри»); слово «каталог» представляет собой более-менее точный перевод этого термина на русский, однако ещё в девяностые годы прошлого века слово «директория» вошло в действующий словарный состав русского языка, перестав быть

В последние годы маркетинговая политика отдельных компаний была направлена на замену термина *directory* словом *folder* (соответствующий русский перевод — «папка»). Для профессиональных программистов такая лексика, как уже отмечалось в предыдущих томах, неприемлема.

Каталог представляет собой особый тип файла, хранящий имена файлов, среди которых, возможно, тоже есть каталоги. При создании на диске файловой системы формируется один каталог, называемый ***корневым каталогом***. При необходимости можно создать дополнительные каталоги, а их имена записать в корневой. Такие каталоги иногда называют *каталогами первого уровня (вложенности)*. В них, в свою очередь, тоже можно создавать каталоги (*второго уровня*) и т. д.

Мы знаем, что при работе с операционной системой обычно некий каталог считается ***текущим***[13], в связи с чем операционная система позволяет использовать имена файлов различного вида. ***Абсолютное*** имя файла однозначно идентифицирует файл в системе и никак не зависит от текущего каталога; в Unix такое имя всегда начинается с символа «/», которым обозначается корневой каталог. Например, в корневом каталоге обычно присутствует каталог первого уровня `home`, в котором по традиции размещаются личные каталоги пользователей системы. Если в системе есть пользователь с именем `vasya`, то именно так обычно и называется его личный каталог; абсолютным именем такого каталога будет строка `/home/vasya`. Если мы в нём создадим каталог третьего уровня `project`, в котором, в свою очередь, разместим файл `program.c`, то абсолютное имя этого файла будет выглядеть так: `/home/vasya/project/program.c`.

Если наш текущий каталог — `/home/vasya/project`, мы можем обратиться к этому же файлу по ***краткому*** имени файла, которое не содержит имён каталогов; в данном случае это имя `program.c`.

Наконец, находясь в каком-то другом каталоге, мы можем воспользоваться ***относительным*** именем файла, которое содержит имена каталогов, но при этом не начинается с символа «/»; такое имя будет *отсчитываться* от текущего каталога. В относительных именах файлов часто используется символ «`..`» (две точки), который обозначает ***родительский каталог***, то есть каталог на уровень выше данного. Например, если мы находимся в каталоге `/home/vasya`, то к тому же самому файлу мы можем обратиться, указав в качестве его имени строку `project/program.c`; если наш текущий каталог — `/home/anya`, то нам потребуется путь `../vasya/project/program.c`, если же мы забрались в каталог `/home/anya/work/progs`, то путь станет длиннее: `../../../vasya/project/program.c`. Символ родительского каталога можно использовать и сам по себе, без сочета-

жаргонным. Отметим, что слово «каталог» тоже имеет иностранное происхождение, просто было заимствовано раньше.

[13]Как мы увидим позже, текущий каталог — это свойство, которым обладает в системе каждый процесс независимо от других процессов.

ния с другими именами каталогов; например, находясь в каталоге `/home/vasya/project/build`, мы могли бы «дотянуться» до всё того же файла через имя `../project.c`.

Система допускает также довольно бессмысленные на первый взгляд комбинации вроде

```
work/progs/../../work/../../vasya/project/../project/program.c
```

или

```
project/../project/../project/../project/../project/program.c
```

Очевидно, что эти странные пути можно записать гораздо короче и понятнее: `../vasya/project/program.c` и `project/program.c`. Как ни странно, в некоторых случаях при написании программ способность операционной системы успешно обрабатывать подобные имена оказывается удобной, но описание таких случаев было бы слишком длинным.

Сложное имя файла, содержащее каталоги и символы «/», часто называют ***путём к файлу*** (англ. *file path*). Более того, можно заметить, что при работе с файлами везде, где подразумевается имя файла, можно указать сложный путь, абсолютный или относительный, так что зачастую от употребления термина «имя файла» вообще отказываются в пользу термина «путь к файлу» или даже «путь файла», хотя это, конечно, жаргонизм, проистекающий от слишком буквального перевода английского *file path*.

В отличие от некоторых других операционных систем, в ОС Unix имена файлов организованы в виде единого дерева каталогов. В имя файла ни в каком виде не входит имя устройства, на котором этот файл находится, то есть ничего похожего на привычные для пользователей Windows обозначения `A:`, `C:` и т. п. в ОС Unix нет. Когда в системе имеется несколько дисков, файловая система одного из них объявляется ***корневой***, а остальные ***монтируются*** в тот или иной каталог, называемый ***точкой монтирования*** (англ. *mount point*), при этом для указания полных путей к файлам на этом диске нужно к полному имени файла в рамках диска добавить спереди полный путь точки монтирования. К примеру, если у нас есть флеш-брелок, на нём создан каталог `work`, в этом каталоге — файл `prog.c`, а сам брелок смонтирован с использованием каталога `/mnt/flash` в качестве точки монтирования, то полный путь к нашему файлу будет выглядеть так: `/mnt/flash/work/prog.c`.

В ОС Unix ***каталоги*** хранят только имя файла и некоторый номер, позволяющий идентифицировать соответствующий файл. Вся прочая информация о файле, включая его тип, размер, расположение на диске, даты создания, модификации и последнего обращения, данные о владельце файла и о правах доступа к нему связываются не с именем

файла (как это делается в некоторых других операционных системах), а с вышеупомянутым номером. **Хранимая на внешнем запоминающем устройстве (диске) структура данных, содержащая всю информацию о файле, исключая его имя, называется *индексным дескриптором*** (англ. *index node*, или `i-node`). Индексные дескрипторы имеют номера, уникальные в рамках файловой системы данного диска; как раз эти номера и записываются в каталоги вместе с именами файлов. Итак, **каждая запись в каталоге состоит из двух полей: имя файла и номер индексного дескриптора**.

С некоторой натяжкой можно заявить, что индексный дескриптор — это и есть файл как таковой, но это будет верно лишь для некоторых специфических типов файлов, а в общем случае файл состоит из индексного дескриптора и некоторого количества *блоков данных*, расположенных где-то на диске. Информация об этих блоках тоже хранится в индексном дескрипторе. Можно также считать, что индексный дескриптор — это некая служебная часть файла, а сам файл представляет собой набор блоков на диске; но это тоже не всегда верно, ведь количество блоков, занимаемых файлом, может быть равно нулю, и тогда такой файл действительно состоит из одного только индексного дескриптора. Никто не мешает нам завести *пустой* файл (файл размера ноль); ни одного блока данных этот файл занимать не будет. Существуют такие типы файлов, которые никогда не занимают ни одного блока. В любом случае, можно сказать совершенно определённо, что **имя файла в ОС Unix не является ни в каком виде частью или принадлежностью самого файла** — имя файла не указывается ни в его индексном дескрипторе, ни тем более в блоках данных, относящихся к файлу; единственное место, где мы встречаем имена файлов — это каталоги.

Отметим, что имя файла в ОС Unix может быть достаточно длинным (обычно ограничение составляет 255 символов) и содержать, вообще говоря, любые символы, кроме нулевого и символа-разделителя. Так, имя файла из пятнадцати точек является в Unix вполне допустимым. Тем не менее, настоятельно не рекомендуется использование в именах файлов таких символов, как пробел, звёздочка, восклицательный и вопросительный знаки, хотя это и возможно. Также рекомендуется воздержаться от использования в именах файлов спецсимволов, таких как перевод строки, табуляция, звонок, backspace и пр., и символов с кодом, превышающим 127, таких как русские буквы. Наконец, имя файла не стоит начинать с символа «-» (минус). Несоблюдение этих рекомендаций приводит к возникновению проблем в работе. Эти проблемы всегда могут быть преодолены, однако преодолимость трудностей не следует считать поводом для их создания.

В ОС Unix допускается, чтобы два или более имён файлов, расположенных как в разных каталогах, так и в одном, ссылались на один и тот же номер индексного дескриптора. Конечно, файл всегда создаёт-

ся под каким-то определённым именем, но позже он может получить дополнительные имена — записи в каталогах, ссылающиеся на тот же самый индексный дескриптор. Такие имена называются ***жёсткими ссылками*** (англ. *hardlinks*). Отличить жёсткую ссылку от оригинального имени файла невозможно: эти имена совершенно равноправны. Очевидно, что жёсткая ссылка может быть установлена только в пределах одного диска; в самом деле, нумерация индексных дескрипторов у каждого диска своя, так что сослаться на индексный дескриптор другого диска не представляется возможным, для этого в структуре данных файла-каталога просто не предусмотрено полей. В индексном дескрипторе содержится, кроме всего прочего, *счётчик количества ссылок* на данный дескриптор. При создании файла этот счётчик устанавливается в единицу, при создании новой жёсткой ссылки — увеличивается на единицу, при удалении ссылки — уменьшается.

Интересно, что в ОС Unix не предусмотрено функции *удаления файла*; вместо этого имеется системный вызов, удаляющий ссылку, который называется `unlink`. При выполнении этого вызова имя удаляется из каталога, а счётчик ссылок в соответствующем индексном дескрипторе уменьшается. **Сам файл удаляется, только если удалённая ссылка была последней (счётчик обратился в нуль) и при этом файл не был ни одним из процессов открыт на запись или чтение.** Если счётчик обратился в нуль, но файл кем-то открыт, удалён он будет только после закрытия. Название `unlink` распространилось и на другие системы — функция, удаляющая файл, обычно называется именно так, что часто вызывает удивление у программистов, плохо знакомых с ОС Unix. Подробно этот и другие системные вызовы для работы с файлами мы рассмотрим в §5.3.4.

Для создания жёсткой ссылки средствами командной строки можно воспользоваться командой `ln`. Она похожа на уже знакомую нам команду `cp`, но осуществляет не копирование файла, а создание для него нового имени.

Каталоги в файловой системе ОС Unix представляют собой не более чем *файлы*, пусть и специального типа. Информацию, содержащуюся в каталоге (имена и номера индексных дескрипторов), нужно где-то хранить, так что файлу типа каталог, как и обычному файлу, должны принадлежать дисковые блоки, у него должен быть размер и т. п. На низком уровне каталог отличается от обычного файла только значением признака типа в индексном дескрипторе; в остальном хранение на диске каталога организовано точно так же, как и хранение обычного файла. Обычными файлами и каталогами многообразие типов файлов в ОС Unix не исчерпывается, постепенно мы познакомимся со всеми остальными типами файлов. Забегая вперёд, скажем, что всего их существует семь, но обычный файл и каталог встречаются чаще других.

Особая роль каталогов накладывает определённые ограничения на то, что с ними можно делать. Во-первых, каталог невозможно удалить, как обычный файл, для этого применяется специальный вызов `rmdir`, который сработает успешно лишь в случае, если удаляемый каталог пуст. Во-вторых, **система запрещает создание жёстких ссылок на каталоги**. Дело в том, что создание таких жёстких ссылок может привести к возникновению ориентированных циклов в дереве каталогов: например, такой цикл получился бы после выполнение команд

```
mkdir a; cd a; mkdir b; cd b; ln ../../a ./c
```

если бы, конечно, система позволила их выполнить. В этой ситуации попытка рекурсивно пройти директорию `a`, скажем, с целью подсчёта количества файлов в ней, привела бы к зацикливанию. Кроме того, оказалось бы, что директорию `a` невозможно удалить, ведь она всегда что-то содержит (косвенно она содержит сама себя). По этой причине жёсткие ссылки на директории запрещены на уровне ядра операционной системы, и никакие права доступа не позволяют этот запрет обойти.

Символическая ссылка (англ. *symbolic link*) представляет собой файл специального типа, содержащий **имя** другого файла. Операция открытия символической ссылки на чтение или запись приводит на самом деле к открытию файла, на который она ссылается, а не её самой. В отличие от жёсткой ссылки, символическая ссылка легко отличима от основного имени файла, поскольку представляет собой самостоятельный файл, пусть и специального типа; это позволяет не накладывать ограничений на установление символических ссылок на каталоги. Будучи файлом, символическая ссылка имеет свой собственный номер индексного дескриптора. Создание и удаление символической ссылки никак не затрагивает ни файл, на который она ссылается, ни его индексный дескриптор. Более того, файл, на который указывает ссылка, может вообще не существовать в момент ее создания, а также может быть удалён позднее, что никак не повлияет на ссылку. Символическая ссылка, содержащая имя несуществующего файла, называется ***висячей*** (англ. *dangling*, более точный буквальный перевод — «болтающаяся»). Висячие ссылки считаются вполне корректным явлением и в некоторых случаях целенаправленно используются. Символические ссылки не ограничены рамками одного диска, поскольку реализуются через хранение имён, а дерево имён, как мы знаем, в ОС Unix общее для всех имеющихся в системе дисковых устройств.

Для создания символической ссылки средствами командной строки следует использовать уже знакомую нам команду `ln`, указав ей флаг `-s`:

```
ln -s /path/to/old/name new_name
```

С символическими ссылками связан один любопытный момент. В индексном дескрипторе довольно ощутимое пространство выделено под

информацию о номерах блоков, занимаемых данным файлом; конкретный размер этого пространства различается от системы к системе, но он вряд ли где-то будет меньше 52 байт[14], а на современных дисках с их огромными размерами может быть гораздо больше. В то же время на символическую ссылку никогда не нужно больше одного блока, имена просто не бывают такими большими. Даже один блок расходовать несколько жаль, ведь из всего блока (например, 4-килобайтного) будет использоваться в большинстве случаев несколько десятков байт. Поэтому символические ссылки реализованы довольно экзотическим способом. Если длина имени, хранимого в ссылке, такова, что её можно уместить в индексный дескриптор, заняв пространство, обычно используемое для информации о номерах блоков, то именно там это имя и располагается, не занимая лишнего блока, то есть файл такой ссылки состоит из одного лишь индексного дескриптора. Если же имя в дескриптор не помещается, система выделяет под такую ссылку отдельный блок.

Позже мы познакомимся с другими типами файлов, которые вообще никогда не занимают дискового пространства за пределами своего индексного узла. Символическая ссылка в этом плане интересна тем, что она может как занимать, так и не занимать отдельный блок.

5.3.2. Права доступа к файлам

Права доступа к файлу (англ. *access permissions*) определяют, кто из пользователей (точнее, процессов, выполняющихся от имени пользователей) какие операции может с данным файлом произвести. Мы уже обсуждали права доступа в первом томе (см. §1.4.11). Для каждого файла в его индексном дескрипторе хранятся идентификатор пользователя-владельца (`uid`, *user id*) и идентификатор группы пользователей (`gid`, *group id*). Сами по себе права доступа к файлу, также хранящиеся в индексном дескрипторе, представляются в виде 12-битного слова (см. рис. 5.2). Младшие девять бит этого слова объединены в три группы по три бита; каждая группа задаёт права доступа для владельца файла, для группы владельца и для всех остальных пользователей. Три бита в каждой группе отвечают за право чтения файла, право записи в файл и право исполнения файла.

Поскольку права доступа состоят из групп битов по три в каждой, логично применять для их записи восьмеричную систему счисления. В большинстве случаев для записи прав доступа хватает трёх восьмеричных цифр, старшая обозначает права для владельца, следующая — для группы, младшая — для всех остальных. Значения отдельных двоичных разрядов в таком числе соответствуют показанным на рисунке: `4` — права на чтение, `2` — права на запись, `1` — права на исполнение.

[14] Десять номеров обычных блоков и три номера *косвенных* блоков, каждый номер занимает четыре байта; косвенные блоки мы обсудим в последней части тома.

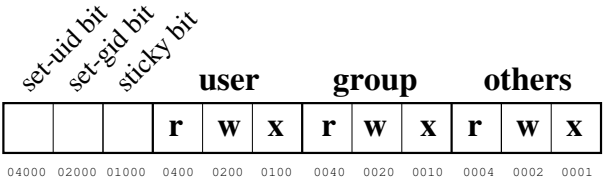

Рис. 5.2. Права доступа к файлу

Например, число 644_8 означает права на чтение и запись для владельца файла, права на чтение для группы и всех остальных; число 751_8 означает, что для владельца установлены права на чтение, запись и исполнение, для группы — на чтение и исполнение, для всех остальных — только на исполнение.

В случае директории биты исполнения указывают, кто (владелец, член группы, обычный пользователь) имеет право доступа к файлам из директории — например, может открыть файл, используя имя из этой директории (при условии, что ему также хватит прав на сам файл). Так, если у нас есть право на чтение директории, но нет права на её «исполнение», то мы сможем узнать имена файлов, содержащиеся в директории, но ни одним из них не сможем воспользоваться; напротив, если у нас есть права на «исполнение», но нет прав на чтение, то прочитать имена файлов в этой директории мы не можем, а воспользоваться сможем лишь такими файлами, имена которых нам стали известны каким-то другим путём, например, их нам просто сказали (при условии, что у нас хватит прав также и на сами файлы). Чаще всего права на чтение и «исполнение» на директорию устанавливаются одинаковыми; реже можно встретить директории, к которым у тех или иных категорий пользователей есть права на «исполнение», но нет прав на чтение; этим вариантом можно воспользоваться, например, чтобы из определённой директории наши коллеги могли забрать (прочитать и скопировать) те файлы, имена которых мы им скажем. Обратная ситуация (чтение разрешено, исполнение запрещено) на практике бессмысленна и не используется.

Оставшиеся три старших разряда прав доступа называются `SetUid bit` (04000), `SetGid bit` (02000) и `Sticky bit` (01000) и имеют достаточно специфическое назначение, различное для исполняемых файлов и директорий; для прочих файлов эти биты обычно не устанавливаются, поскольку ни на что не влияют. Для исполняемых файлов установленный бит `SetUid` означает, что при запуске такого файла программа, записанная в нём, будет выполняться не как обычно, с правами того, кто её запустил, а *с правами владельца файла*; аналогичным образом программа, для исполняемого файла которой установлен бит `SetGid`,

при выполнении получает «групповые полномочия» в соответствии с пользовательской группой, которой принадлежит файл.

Довольно интересна история `Sticky bit`. Само слово *sticky* переводится как «липкий»; в очень старых версиях ОС Unix этим битом помечали часто используемые в системе программы, чтобы сообщить ядру, что после окончания выполнения такой программы её сегмент кода не следует удалять из оперативной памяти, поскольку, скорее всего, кто-нибудь в ближайшее время запустит её снова. Машинный код такой программы, будучи единожды загружен в оперативную память, как бы «прилипал» к ней. В современных системах такая оптимизация лишена смысла, поскольку ядро буферизует весь файловый ввод-вывод, так что если некоторый исполняемый файл будет использоваться достаточно часто, его содержимое в любом случае окажется в буферах ядра; поэтому `Sticky bit`, даже если его установить, будет системой проигнорирован. В наше время он используется только для директорий и имеет совершенно иной смысл, который мы опишем чуть позже.

Ещё одно важное замечание связано с тем, что в системах Unix исполняемые файлы бывают двух видов: обычные, содержащие машинный код в определяемом системой формате, и *скриптовые* — текстовые, содержащие программу на некотором интерпретируемом языке программирования. Со скриптами на языке Bourne Shell мы уже встречались в первом томе (см. §1.4.13) и знаем, что такой файл должен начинаться со строки

```
#!/bin/sh
```

которая указывает системе, что для исполнения этого файла нужно запустить программу `/bin/sh` (командный интерпретатор) и передать ему имя файла-скрипта через аргумент командной строки. В роли интерпретатора может выступать не только `/bin/sh`, подойдёт любой интерпретатор, «знающий», что его будут использовать в таком качестве. Например, часто можно встретить оформленные таким образом программы, написанные на языке Perl, реже — на таких языках, как Scheme или Prolog. Достаточно «навесить» на скриптовой файл бит исполнения, и его можно будет запускать в командной строке точно так же, как и обычную программу в машинном коде, созданную с помощью ассемблера, компилятора Паскаля, Си и других компилируемых языков. Так вот, **биты SetUid и SetGid для скриптовых исполняемых файлов не работают**; это ограничение введено искусственно, из соображений безопасности.

Чтобы понять, в чём заключается проблема с безопасностью suid-скриптов, представьте себе, что в вашей системе есть скрипт (неважно, на каком языке), который должен выполняться с правами его владельца — например, пользователя `root` (системного администратора). При этом некто получил (легитимно или нет) доступ к вашей системе как обычный пользователь, имеющий право исполнять этот скрипт, и хочет расширить свои полномочия.

При запуске скрипта на исполнение система вынуждена сначала обратиться к файлу скрипта, открыть его на чтение, чтобы получить из него информацию

об имени используемого интерпретатора; затем она закрывает файл скрипта и запускает командный интерпретатор, который, в свою очередь, откроет файл скрипта на чтение.

Пользователь, имеющий право на исполнение вашего скрипта, может атаковать вашу систему, написав свой собственный скрипт на том же языке, делающий что-то совершенно другое — например, открывающий какой-нибудь канал для связи и исполняющий произвольные команды, присланные через этот канал. Дальше атакующий создаёт символическую ссылку на ваш скрипт, а затем запускает (естественно, не вручную, а с помощью им же написанной программы) ваш скрипт через эту ссылку и тут же меняет ссылку так, чтобы она указывала на его собственный скрипт. После достаточного числа попыток он может поймать такой момент, когда система уже успела прочитать имя командного интерпретатора и запустить этот интерпретатор, но сам он ещё не успел открыть на чтение файл скрипта; в этом случае с правами администратора будет выполняться не ваш скрипт, а тот, который написал атакующий, и таким образом он получит возможность выполнять произвольные действия с администраторскими правами, которых вы ему не давали.

Именно поэтому система игнорирует биты SetUid и SetGid при исполнении скриптов.

Для директорий старшие биты прав доступа имеют иное значение, нежели для исполняемых файлов, причём применять их имеет смысл только для директорий, публично доступных на запись. Чаще всего используется `Sticky bit`: если он установлен, то в такой директории пользователи имеют право удалять *только свои собственные файлы*, несмотря на наличие прав на запись в директорию. Обычно для создания «общественной» директории применяются права `01777` — всем разрешается всё, но при этом взведённый `Sticky bit` не позволяет удалять чужие файлы. Именно такие права устанавливаются на директорию `/tmp`, предназначенную для временных файлов и доступную всем пользователям системы.

`SetGid bit`, установленный на директорию, приводит к тому, что для файлов, создаваемых в такой директории, в качестве *группы пользователей* принудительно устанавливается та же группа, что и для самой директории, вне зависимости от того, в какие группы входит создатель файла. Такие права применяются, например, для директории `/var/spool/mail`, содержащей почтовые ящики пользователей системы; это позволяет почтовой подсистеме, для которой создана специальная группа (обычно называемая просто `mail`), осуществлять доступ ко всем почтовым ящикам, кто бы их ни создал. Если в такой директории создать поддиректорию, она унаследует не только группу пользователей своей родительской директории, но и её `SetGid bit`, так что внутри поддиректорий новые файлы тоже будут наследовать ту же самую группу.

`SetUid bit` для директорий в большинстве систем (в том числе в Linux) игнорируется, но, например, в системе FreeBSD имеет эффект,

аналогичный вышеописанному эффекту `SetGid bit`, то есть все файлы в такой директории создаются принадлежащими её владельцу вне зависимости от того, кто из пользователей создаёт этот файл.

Некоторые системы поддерживают специальную комбинацию битов: если на файле установлен `SetGid bit`, но при этом сброшен в ноль бит исполнения для группы, считается, что для такого файла должны применяться так называемые *обязательные захваты* (англ. *mandatory locking*). Вопросы, связанные с «захватом» доступа к файлам, мы будем рассматривать в части, посвящённой работе с разделяемыми данными; впрочем, обязательные захваты всё равно никто не использует.

Отметим ещё один довольно неочевидный момент. Для символических ссылок права обычно игнорируюся, то есть ни на что не влияют; при выполнении всех операций используются права доступа того файла, на который ссылка ссылается. Этот момент станет более понятен, если мы заметим, что *изменить* права доступа для символической ссылки мы не смогли бы, даже если бы очень хотели: вызов `chmod`, если его применить к символической ссылке, изменит права не для неё самой, а для файла, на который она ссылается.

Узнать права доступа к конкретному файлу можно знакомой нам командой `ls -l`, которая для их получения использует системный вызов `stat`, а изменить права — командой `chmod` (сокращение английских слов *change mode* — смена режима), системный вызов называется так же. Сменить владельца и группу для файла в командной строке можно командами `chown` (*change owner*) и `chgrp` (*change group*), а системный вызов для этих целей используется один — `chown`. Системные вызовы `stat`, `chmod` и `chown` мы рассмотрим в §5.3.4.

5.3.3. Чтение и запись содержимого файлов

Как отмечалось в §5.1.4, понятие *потока байтов* занимает одно из центральных мест в принципах (если угодно, в философии) систем семейства Unix; именно в виде таких потоков оформляется едва ли не любая[15] передача массивов информации.

Основные системные вызовы для работы с потоками ввода-вывода мы уже рассматривали (см. т. 2, §4.6.8). Напомним, что для открытия файла служит системный вызов `open`:

```
int open(const char *name, int mode);
int open(const char *name, int mode, int perms);
```

Параметр `name` задаёт имя файла, которое можно указать любым из способов, перечисленных в предыдущем параграфе, то есть в виде абсо-

[15] Существуют исключения и из этого правила; так, передача данных из одного процесса в другой через очереди сообщений System V и через области разделяемой памяти совсем не похожа на потоки байтов. В unix-системах эти способы применяются во много раз реже, чем традиционные потоки ввода-вывода.

лютного или относительного пути, а также в краткой форме, если файл расположен в текущем каталоге. Параметр `mode` задаёт режим, в котором мы намерены работать с файлом, в виде целого числа, отдельные биты которого обозначают те или иные особенности предстоящей работы; каждый такой бит представлен предопределённой целочисленной константой, а комбинации из нескольких битов мы получаем, применяя операцию побитового «или» («`|`»). Основные режимы работы — `O_RDONLY` (только чтение), `O_WRONLY` (только запись) и `O_RDWR` (чтение и запись), из этих трёх констант нужно указать одну и только одну. К ним можно добавить дополнительные флаги, такие как `O_APPEND` (режим добавления, когда перед каждой записью в файл текущая позиция принудительно устанавливается в его конец), `O_CREAT` (разрешение создания файла, если его нет), `O_TRUNC` (сброс старого содержимого, если файл уже существовал), `O_EXCL` (требование создать новый файл, а если он уже существует — выдать ошибку), `O_NONBLOCK` (открыть файл в ***неблокирующем режиме***, при котором системные вызовы чтения и записи никогда не ждут готовности, а возвращают управление сразу же — возможно, при этом выдают ошибку, если немедленной готовности не было). Существуют также другие флаги; некоторые из них мы рассмотрим позже.

Третий параметр (`perms`, права доступа) используется при создании нового файла и указывается только если во втором параметре присутствует `O_CREAT`. Файлы, которые мы создаём, обычно не нуждаются в правах на исполнение, так что максимальные разумные права — чтение и запись для всех категорий пользователей — представляются константой `0666` (напомним, что в языке Си лидирующий ноль означает восьмеричную константу). Вспомнив, что пользователь может, установив значение параметра `umask`, убрать «неугодные» ему права доступа для всех создаваемых файлов, мы приходим к выводу, что практически всегда в вызове `open` третьим параметром следует использовать число `0666`, чуть реже — число `0600` (для файлов, содержимое которых заведомо не следует никому показывать, например, для файлов, содержащих пароли, секретные ключи, номера кредитных карт и т. п.).

Вызов `open` возвращает значение `-1` в случае ошибки, а в случае успеха — небольшое целое неотрицательное число, называемое ***дескриптором файла*** или ***файловым дескриптором***[16]. Файловый дескриптор на самом деле располагается в ядре операционной системы и представляет собой достаточно сложную структуру данных, зависящую от

[16] Дескрипторы открытых файлов ни в коем случае не следует путать с *индексными дескрипторами*, это совершенно разные и никак между собой не связанные сущности. Отметим, что в английском языке слово *descriptor* применяется только для обозначения дескрипторов открытых файлов, термин же *индексный дескриптор* представляет собой пример неудачного (но, к сожалению, прижившегося) перевода: оригинальный англоязычный термин *index node* вообще не содержит слова *descriptor* и буквально может быть переведен как *индексный узел*.

конкретной реализации ядра и содержащую всю информацию, нужную ядру для поддержки работы с данным потоком ввода-вывода. В пользовательском процессе нам доступен лишь ***номер дескриптора*** — целое неотрицательное число, используемое, чтобы различать между собой файловые дескрипторы, доступные одному и тому же процессу. Заметим, что номер дескриптора локален по отношению к процессу: скажем, дескриптор №5 может в одном процессе быть связан с одним файлом, в другом — с совсем другим, а в третьем и вовсе не соответствовать никакому потоку ввода-вывода.

Дескрипторы могут быть связаны не только с открытыми дисковыми файлами, но и с потоками ввода-вывода произвольной природы. Дескрипторы с номерами 0, 1 и 2 играют особую роль: программы обычно исходят из соглашения, что именно дескрипторы с этими номерами являются стандартными потоками ввода, вывода и сообщений об ошибках. Как правило, на момент запуска программы эти дескрипторы уже открыты. Сказанное никоим образом не означает, что мы не можем использовать их для своих целей или связать с другими потоками ввода-вывода, в частности, с файлами. Подробнее к этому вопросу мы ещё вернемся.

Чтение из потока ввода и **запись** в поток вывода производится системными вызовами `read` и `write`:

```
int read(int fd, void *mem, int len);
int write(int fd, const void *data, int len);
```

Первый параметр задаёт файловый дескриптор; второй параметр указывает на область памяти, в которой следует разместить прочитанные данные (для `read`) либо из которой нужно взять данные для передачи в поток (для `write`); последний параметр указывает размер этой области памяти, то есть задаёт количество данных, которые нужно прочитать или записать.

Вызов `read` пытается прочитать из заданного потока заданное количество данных. Если в указанном потоке отсутствуют данные, готовые к прочтению, вызов заблокирует вызвавший процесс до тех пор, пока данные не появятся, и только после их прочтения вернёт управление. Если данные присутствуют, но их меньше, чем требует вызывающий, вызов сохранит их в области памяти по адресу `mem` и вернёт управление; подчеркнём, что **вызов `read` немедленно возвращает управление, если может отдать вызвавшей программе хотя бы один байт данных**, то есть он не станет ждать, пока данных накопится столько, сколько просили. Если бы это было устроено иначе, мы не могли бы писать интерактивные программы.

Вызов `write` пытается записать в поток заданное количество информации, расположенной по заданному адресу. При работе с некоторыми

«особенными» потоками вывода вызов `write` может заблокировать вызвавший процесс до тех пор, пока в потоке (точнее, в буфере внутри ядра) не появится свободное место. Например, так может случиться, если поток предполагает отправку данных через сеть на другой компьютер, но скорость, с которой вы пытаетесь передавать данные в поток, превышает скорость работы сети. То же самое произойдёт, если ваш поток вывода представляет собой канал связи с другим процессом и этот другой процесс, занятый своими делами, не читает данные из канала. При записи в обычные дисковые файлы блокировки почти никогда не случается, диски для этого достаточно быстры.

В случае ошибки `read` и `write` возвращают `-1`. В случае успешного выполнения чтения или записи возвращается положительное число, означающее количество переданных байтов информации. Естественно, это число не может быть больше `len`. **При наступлении ситуации «конец файла» `read` возвращает ноль**. Для обычного файла это означает, что мы дочитали его до конца и больше там читать нечего; при чтении с клавиатуры ситуация конца файла означает, что пользователь сымитировал её нажатием Ctrl-D; для потоков иной природы будет иной и природа ситуации «конец файла» — например, на сетевом сокете она наступает, если наш партнёр разорвал соединение.

Напомним, что анализ значения, возвращаемого `read` как функцией, строго обязателен. В программе не должно содержаться операторов вроде

```
read(fd, buf, sizeof(buf));
```

Правильный вызов `read` выглядит, например, так:

```
count = read(fd, buf, sizeof(buf));
```

Для вызова `write` всё не так строго, поскольку, например, при записи в поток небольшого фрагмента данных (нескольких десятков байт) можно практически достоверно предполагать, что этот фрагмент будет записан целиком либо произойдёт ошибка; больше того, в некоторых случаях мы вполне можем предполагать, что никакой ошибки не произойдёт, ведь не проверяли же мы в наших программах результат работы `printf`. Впрочем, вывод в стандартный поток вывода — это скорее исключение из правил, в остальных же случаях значение, возвращённое функцией `write`, лучше проверить. В большинстве случаев число записанных байтов в точности равняется значению `len`, однако полагаться на это опасно. Можно назвать ситуацию, при которой `write` заведомо вернёт число, мéньшее значения параметра `len`: когда наш поток связан с сетевым соединением, *объявлен неблокирующим* и мы попытаемся за один вызов отправить в сеть массив данных, не помещающийся в буфер ядра операционной системы (например, несколько мегабайт). Система

при этом примет у нас для последующей отправки в сеть столько информации, сколько поместится в буферной памяти.

Надо сказать, что если не перевести поток в неблокирующий режим, то вызов `write` в большинстве систем предпочтёт отправить в поток все данные, которые вы ему предоставили, и вернуть управление только после этого. Напомним, что поток будет неблокирующим, если при его открытии вызовом `open` воспользоваться флагом `O_NONBLOCK`; система также позволяет перевести в неблокирующий режим уже открытый поток с помощью системного вызова `fcntl`, и для потоков, создаваемых иными способами, нежели `open`, это может оказаться единственным вариантом. Вызов `fcntl` мы подробно рассмотрим чуть позже.

Вызов `write` в современных системах не возвращает ноль, если только параметр `len` не был равен нулю (но так делать не следует в любом случае). В прошлом существовали системы семейства Unix, в которых `write` мог вернуть ноль в неких специфических обстоятельствах; современные системы в таких случаях всегда возвращают `-1`.

После окончания работы с файлом его следует закрыть. Это особенно важно, поскольку дескрипторы — ресурс ограниченный: их общее количество в системе не может превышать некоторого числа, как и количество дескрипторов, открытых одним процессом. Закрытие файла производится вызовом `close`:

```
int close(int fd);
```

где `fd` — дескриптор, подлежащий закрытию. Вызов возвращает ноль в случае успеха, `-1` в случае ошибки, но даже если произошла ошибка, **дескриптор после вызова `close` никогда не остаётся открытым**. Считается, что игнорировать значение, возвращаемое вызовом `close`, не следует, особенно после записи данных в поток вывода, поскольку при использовании асинхронного режима вывода (а в большинстве ситуаций используется именно он) ошибка передачи данных может проявиться *после последнего вызова `write`*, и тогда закрытие потока останется для операционной системы последней возможностью нам о такой ошибке сообщить.

Текущая позиция в открытом файле может быть изменена с помощью системного вызова `lseek`:

```
int lseek(int fd, int offset, int whence);
```

Параметр `fd`, как обычно, задаёт номер файлового дескриптора. Параметр `offset` указывает, на сколько байт следует сместиться, параметр `whence` определяет, от какого места эти байты следует отсчитывать: от начала файла (`SEEK_SET`), от текущей позиции (`SEEK_CUR`) или от конца файла (`SEEK_END`). Вызов возвращает новое значение текущей

позиции, считая от начала; например, `lseek(fd, 0, SEEK_END)` вернёт длину файла, а текущей позицией станет его конец.

При смене позиции можно зайти за конец файла. Само по себе это не приводит к изменению размера файла, но если после этого произвести запись, размер файла увеличится; конечно, файл при этом должен быть открыт в режиме, допускающем запись. При этом возможно образование «дырки» (англ. *hole*) между последними данными перед старым концом файла и первыми данными, записанными в новой позиции. Ничего страшного в этом нет, «дырки» считаются штатной возможностью файловых систем. При чтении из таких мест файла мы получаем массивы нулевых байтов, так что если нам по каким-то причинам нужен файл, значительная часть которого забита нулями, можно не тратить физическое место на диске для хранения таких «нулевых областей».

В большинстве случаев (хотя и не во всех) операции записи выполняются системой в так называемом *асинхронном режиме вывода*. Это означает, что система сразу же принимает у нас информацию, переданную через вызов `write`, копирует её в свою внутреннюю память, возвращает нашему процессу управление (в нашей программе это выглядит как успешно завершившийся вызов `write`) а физическую операцию записи производит позже — например, когда окажется свободен нужный контроллер и для него не будет более срочных дел. Асинхронный режим позволяет процессам продолжать работу, не дожидаясь, пока будет физически завершена операция вывода, и это в большинстве случаев полезно, поскольку от результата записи (в отличие от результатов чтения) дальнейшая работа программы обычно не зависит.

С другой стороны, у асинхронного режима вывода имеется очевидный недостаток: если работа компьютера будет неожиданно прервана (например, пропадёт электропитание), некоторые из операций записи, завершившиеся успешно с точки зрения процессов, физически так и не будут исполнены. Если речь идёт о данных, только что сгенерированных нашим процессом, то в принципе ничего страшного не произойдёт — можно представить, что работа компьютера прервалась чуть раньше, так что процесс просто не успел создать эти данные. Совершенно иначе выглядит ситуация с данными, перемещаемыми из одного места в другое — с диска на диск и т. п. Например, почтовый сервер при выполнении локальной доставки электронного письма читает его содержимое из почтовой очереди, которая хранится в виде файлов на диске, записывает письмо в почтовый ящик пользователя, который также представляет собой файл, а затем, убедившись, что запись успешно прошла, удаляет письмо из очереди. Если компьютера выключить во время выполнения такой операции, высока вероятность, что письма не окажется ни в очереди, ни в почтовом ящике, то есть оно потеряется.

Для таких случаев система предусматривает возможность в явном виде дождаться завершения операции записи; делается это системным вызовом `fsync`:

```
int fsync(int fd);
```

Параметром служит номер файлового дескриптора. Вызов сообщает операционной системе, что все данные, связанные с файлом, должны быть как можно скорее физически записаны на диск; точнее говоря, должна быть как можно скорее выполнена физическая операция записи, поскольку речь не всегда идёт именно о дисковых файлах. Управление из этого вызова возвращается не раньше, чем система реально выполнит все нужные операции записи. Вызов возвращает `-1` в случае ошибки, `0` в случае успеха.

Во многих случаях вместо вызова `fsync` имеет смысл применять вызов `fdatasync`, имеющий точно такой же профиль и работающий аналогично, но гарантирующий запись на диск только той информации, которая необходима для последующего корректного прочтения данных; например, информация о времени последней модификации при этом на диск принудительно не записывается, поскольку для последующего чтения она не важна.

Кроме того, система поддерживает вызов `sync`, не принимающий параметров и не возвращающий значений. Этот вызов сообщает системе о необходимости немедленно физически записать всю информацию, которая пока что содержится только в памяти системы, но предназначена для записи на диск. Подчеркнём, что речь идёт о *всей* информации для *всех* дисковых файлов, находящихся в работе, для *всех пользователей системы*. Управление возвращается после того, как все операции физически произойдут. Вызов `sync` всегда заканчивается успешно, то есть система ни при каких условиях никому не отказывает в требовании выполнить общую синхронизацию. Впрочем, не следует думать, что, получив этот вызов, ядро бросит все свои дела и побежит писать данные на диск; запрос будет принят к сведению, соответствующие дисковые операции запланированы, дальше ядро их *когда-то* выполнит в своём обычном режиме. Если из соображений эффективности ядро сочтёт, что какие-то другие действия следует выполнить раньше, то оно выполнит их раньше. Иначе говоря, речь тут идёт не о том, что ядро будет сломя голову исполнять вашу волю, а о том, что ваш процесс будет ждать столько, сколько потребуется ядру на физическую запись всей информации, которая ещё не была записана в момент вызова `sync`. Это объясняет, почему вызов `sync` доступен всем, а не только администратору.

Ядро операционной системы поддерживает гибкую настройку поведения потоков, связанных с дескрипторами; доступ практически ко

всем возможностям такой настройки предоставляет системный вызов `fcntl`:

```
int fcntl(int fd, int cmd, ...);
```

Первым параметром вызов получает номер дескриптора, вторым — некий «код команды», задаваемый одной из предопределённых целочисленных констант, предназначенных специально для этого. Многоточие означает, что потенциально вызов может получить ещё сколько угодно параметров любых типов; конкретные параметры и их типы зависят от значения второго параметра (команды).

Для описания большинства команд `fcntl` у нас пока недостаточно знаний, но две мы можем рассмотреть прямо сейчас. При описании вызова `open` (см. стр. 50) мы перечислили несколько флагов, передаваемых в `open` вторым параметром. Эти флаги можно разделить на *флаги режима доступа* (`O_RDONLY`, `O_WRONLY`, `O_RDWR`), *флаги создания файла* (`O_CREAT`, `O_EXCL`, `O_TRUNC` и пока не рассматривавшийся нами `O_NOCTTY`), а также *флаги статуса* — все остальные, из которых мы пока обсуждали только `O_APPEND` и `O_NONBLOCK`. Флаги статуса отличаются от всех остальных тем, что их возможно и даже иногда осмысленно менять уже после начала работы с файлом.

Текущее значение флагов статуса для открытого потока можно узнать, выполнив вызов `fcntl` с командой `F_GETFL`. Дополнительные параметры в этом случае не предусмотрены, вызов выполняется от двух параметров:

```
flags = fcntl(fd, F_GETFL);
```

Установлен ли или нет конкретный флаг, можно понять с помощью операции побитового «и»:

```
if(flags & O_APPEND) { /* флаг O_APPEND установлен */
    /* ... */
```

Изменить текущий набор флагов статуса позволяет команда `F_SETFL`; в этом случае вызов `fcntl` получает третий параметр, имеющий тип `long` и представляющий собой побитовую дизъюнкцию нужных флагов. Например, следующая последовательность действий позволяет установить для файла неблокирующий режим, сохранив остальные флаги неизменными:

```
flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

Впрочем, можно просто выполнить `fcntl(fd, F_SETFL, O_NONBLOCK)`, если вы уверены, что никакие другие флаги статуса на вашем потоке ввода-вывода не используются.

Позже, по мере появления нужных знаний, мы рассмотрим другие команды вызова `fcntl`.

5.3.4. Управление объектами файловой системы

Кроме чтения информации из файлов и записи в файлы, с объектами файловой системы приходится производить целый ряд других действий. Рассмотрим некоторые из предназначенных для этого системных вызовов. Сразу же отметим, что все эти системные вызовы возвращают значение `-1` в случае ошибки, а в случае успеха возвращают какое-то другое значение (чаще всего `0`), но традиционно результат проверяют именно на равенство `-1`.

Жёсткие ссылки создаются с помощью системного вызова `link`:

```
int link(const char *oldpath, const char *newpath);
```

где `oldpath` — существующее имя файла, `newpath` — новое имя. Как обычно, имя считается абсолютным, если оно начинается с символа «/», в противном случае оно отсчитывается от текущей директории; в дальнейшем мы не будем возвращаться к этой оговорке, поскольку она без изменений применяется ко всем системным вызовам и библиотечным функциям, в которых упоминаются имена файлов, за одним исключением, которое мы, когда придёт время, оговорим явно. Отметим, что оба имени должны быть именами файлов, а не чем-то другим. Читатель может заметить, что команда командной строки `ln`, позволяющая создавать жёсткие ссылки, принимает в качестве второго аргумента как имя файла, так и имя директории, при этом ссылка создаётся в этой (новой) директории под тем же именем, что и в старой; но этот вариант команда `ln` реализует своими средствами, а системный вызов `link` такого не поддерживает.

Системный вызов, предназначенный для удаления файла или, точнее, для удаления *ссылки* на файл из каталога, называется `unlink`:

```
int unlink(const char *name);
```

Напомним, что файл реально удаляется лишь тогда, когда удаляется последняя жёсткая ссылка на него, и то только в том случае, если этот файл не был никем открыт, а иначе файл продолжает существовать в качестве дискового объекта, не имея при этом ни одного имени; счётчик ссылок в его индексном дескрипторе при этом равен нулю. Такой файл удаляется с диска после того, как его закроет последний из работавших с ним процессов. Может получиться так, что система окажется некорректно остановлена (например, при аварии электропитания) и файл с нулевым счётчиком ссылок так и останется в файловой системе; в этом случае он, как правило, удаляется при следующей загрузке системы. Отметим, что вызов `unlink` не позволяет удалять директории.

Переименование файла производится системным вызовом `rename`:

```
int rename(const char *oldpath, const char *newpath);
```

Параметры задают старое имя, то есть то имя, под которым файл уже существует в системе, и новое имя, которое этому файлу следует дать вместо старого. Имена могут относиться как к одной директории, так и к разным, то есть файл можно с помощью `rename` переместить из одной директории в другую, но только если эти директории находятся на одном диске; перемещать файл между разными файловыми системами этот вызов не умеет.

У вызова `rename` есть одно важное свойство, которое часто используется на практике: если файл с именем, заданным параметром `newpath`, существует, то вызов *заменяет* его файлом, заданным параметром `oldpath`, причём это происходит за одно атомарное (неделимое) действие. На директории это свойство не распространяется: если `oldpath` задаёт директорию, то `newpath` должен либо не существовать вовсе, либо быть пустой директорией (в этом случае замена всё же произойдёт). Кроме того, атомарность переименования не всегда может быть обеспечена для сетевых дисков (физически находящихся на других компьютерах).

Как мы знаем, для создания директории в командной строке применяется команда `mkdir`, а для её удаления — команда `rmdir`; точно так же называются и системные вызовы, с помощью которых можно попросить операционную систему создать или удалить директорию:

```
int mkdir(const char *name, int perms);
int rmdir(const char *name);
```

Параметр `name` для обоих вызовов задаёт имя директории (создаваемой или удаляемой); что касается параметра `perms`, то этот параметр задаёт ***права доступа*** к создаваемой директории, подобно одноимённому параметру для вызова `open`, рассмотренному в предыдущем параграфе. Как и при создании файлов, из значения `perms` удаляются биты, которые установлены в `umask`, что позволяет в большинстве случаев не беспокоиться о защите создаваемой директории: пользователь может позаботиться об этом самостоятельно, установив соответствующее значение параметра `umask`. С другой стороны, в данном случае мы создаём директорию, а не файл, а для директории очень важен бит `x` (права на исполнение), без которого использовать содержимое директории невозможно; поэтому в большинстве случаев мы указываем параметром `perms` число `0777`, а в достаточно редких случаях, когда создаваемая директория будет заведомо критична по безопасности — число `0700`, чтобы никто, кроме владельца, не имел к ней доступа вне зависимости от значения `umask`.

Символические ссылки создаются вызовом `symlink`:

```
int symlink(const char *oldpath, const char *newpath);
```

очень похожим на уже рассматривавшийся вызов `link`. Система требует, чтобы строка `oldpath` не была пустой, но больше никаких проверок этого параметра не делает. Символическая ссылка (то есть файл специального типа) создаётся под именем `newpath` и ссылается на `oldpath` вне зависимости от того, существует ли в действительности файл с таким именем; если его не существует, созданная ссылка оказывается «повисшей», но в этом, если подумать, нет ничего страшного.

Удаление символической ссылки, как и обычного файла, производится с помощью вызова `unlink`.

Для изменения прав доступа к файлам используется системный вызов, называющийся так же, как и соответствующая команда командной строки, `chmod`:

```
int chmod(const char *path, int perms);
```

Первым параметром вызова указывается имя файла, вторым — новые права доступа к нему в виде целого числа, биты которого соответствуют битам прав доступа. Чаще всего число задаётся явным образом в виде восьмеричной константы. Например, `chmod("file.txt", 0600);` установит для файла `file.txt` в текущей директории права на запись и чтение для его владельца, а все остальные права сбросит. Обычный пользователь может изменить права доступа только для файлов, принадлежащих ему, но на системного администратора это ограничение, как обычно, не распространяется. Вызов `chmod` возвращает `-1` в случае ошибки, `0` в случае успеха. Отметим, что параметр `umask` на работу вызова `chmod` никакого влияния не оказывает.

Для изменения владельца файла служат вызовы `chown` и `lchown`:

```
int chown(const char *path, int owner, int group);
int lchown(const char *path, int owner, int group);
```

Различие между ними в том, что если их применить к символической ссылке, то `chown` будет работать с файлом, на который эта ссылка ссылается, а `lchown` — с самой ссылкой, хотя в большинстве случаев это бессмысленно[17]. Параметры `owner` и `group` задают числовые идентификаторы пользователя и группы пользователей. Если в качестве любого из этих параметров указать значение `-1`, соответствующий идентификатор для файла сохраняется без изменений. Отметим, что изменить владельца для любого файла может только суперпользователь (т. е. для этого процесс, выполняющий вызов, должен иметь `euid`, равный `0`), а изменить группу может, кроме суперпользователя, также владелец файла, но только на такую группу, членом которой он является (то

[17] Единственный хорошо известный случай, когда для символической ссылки важен её владелец — при нахождении ссылки в директории, имеющей установленный Sticky bit, поскольку в такой директории пользователь может удалять только свои файлы.

есть указанный третьим парметром `gid` должен либо быть равен `euid`'у вызывающего процесса, либо находиться среди его «дополнительных групп»). Попытка выполнить неразрешённые изменения приведёт к тому, что вызов завершится ошибкой. Как обычно, в случае ошибки вызов возвращает значение `-1`; в случае успеха возвращается ноль.

Вызовы `stat`, `lstat` и `fstat` позволяют узнать подробную информацию о файле. Профили этих вызовов выглядят так:

```
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

Различие между `stat` и `lstat` такое же, как между `chown` и `lchowm`: если первым аргументом будет имя символической ссылки, то `stat` пройдёт по ссылке и выдаст информацию по файлу, а `lstat` выдаст информацию по самой ссылке. Вызов `fstat` позволяет узнать информацию об открытом файле, не зная его имя: первым параметром он получает открытый файловый дескриптор. Извлечённую информацию вызовы записывают в структуру, адрес которой передан параметром `buf`. Используемая для этого структура `struct stat`, описанная в системных заголовочных файлах, содержит следующие поля:

- `st_dev`: идентификатор устройства, на котором находится файл;
- `st_ino`: номер индексного дескриптора (*i-node*);
- `st_mode`: тип файла и права доступа к файлу;
- `st_nlink`: количество ссылок на файл;
- `st_uid`: идентификатор пользователя-владельца файла;
- `st_gid`: идентификатор группы пользователей;
- `st_rdev`: идентификатор устройства (для специальных файлов, соответствующих устройствам в системе);
- `st_size`: размер файла в байтах;
- `st_blksize`: размер блока ввода-вывода на данной файловой системе;
- `st_blocks`: количество блоков по 512 байт, занятых данным файлом;
- `st_atime`: дата и время последнего доступа к файлу;
- `st_mtime`: дата и время последней модификации содержимого файла;
- `st_ctime`: дата и время последнего изменения свойств файла (например, владельца, группы, прав доступа; запись информации в файл также изменяет этот параметр).

Отметим, что битовая строка, записываемая в поле `st_mode`, содержит как права доступа к файлу, так и его тип; для извлечения прав доступа как таковых следует произвести побитовую конъюнкцию значения из этого поля с маской `07777` (`0x0fff`), тем самым выделив из него 12 младших бит.

Существующий *обыкновенный* файл можно укоротить, уничтожив при этом всё его содержимое, начиная с определённой позиции. Это можно сделать с помощью системного вызова `truncate`, не открывая файл; если же файл уже открыт, можно воспользоваться вызовом `ftruncate`:

```
int truncate(const char *path, int length);
int ftruncate(int fd, int length);
```

Первым параметром для `truncate` служит имя файла, для `ftruncate` — открытый файловый дескриптор; вторым параметром обоих вызовов задаётся новая длина файла. Если файл до этого имел бóльшую длину, он укорачивается, а его содержимое, начиная с позиции `length`, теряется; если же файл был короче, чем указано в параметре, то он становится длиннее, а при чтении байтов, добавленных таким способом, мы получаем нули. Система, насколько возможно, старается в таких случаях использовать уже знакомые нам по вызову `lseek` «дырки».

Отметим, что операция изменения длины считается операцией записи в файл, так что при использовании `truncate` у нас должны быть права на запись в данный файл, а при использовании `ftruncate` файл должен быть открыт в режиме записи (`O_WRONLY` или `O_RDWR`).

5.3.5. Файлы устройств и классификация устройств

Одно из замечательных свойств ОС Unix — это обобщённая концепция файла как универсальной абстракции. Большинство внешних устройств представлено на пользовательском уровне как файлы специального типа. Это касается и жёстких дисков, и всевозможных последовательных и параллельных портов, и виртуальных терминалов, и т. п.

Так, часто требуется записать в файл образ компакт-диска (CD), например, для последующего создания его копии. В некоторых других операционных системах для проведения такой операции необходимо специальное программное обеспечение. В системах семейства Unix достаточно вставить диск в привод и дать команду

```
cat /dev/cdrom > image.iso
```

Точно так же, чтобы отправить файл на печать, достаточно команды

```
cat myfile.ps > /dev/lp0
```

Конечно, обычно так не делают, полагаясь на подсистему печати, однако нам в данном случае важнее сам факт такой возможности.

Такой подход позволяет работать с устройствами в основном с помощью тех же системных вызовов, что и для обычных файлов. Так,

чтобы записать информацию в определённый сектор жёсткого диска, в других операционных системах требуется обратиться к системному вызову, специально предназначенному для записи секторов физического диска. В ОС Unix достаточно открыть на чтение специальный файл, соответствующий нужному диску, с помощью вызова `lseek` позиционироваться на нужный сектор и выдать обычный `write`. Именно так происходит, например, форматирование диска, т. е. создание файловой системы.

В некоторых старых версиях Linux при вводе звукового сигнала с микрофона можно было открыть на чтение файл, соответствующий звуковому устройству, и произвести чтение. Прочитанную таким образом информацию можно было снова записать в звуковое устройство, в результате чего звук воспроизводился. К сожалению, нынешняя звуковая подсистема стала слишком сложной для таких наглядных действий.

Надо отметить, что некоторые периферийные устройства могут и не иметь файлового представления. Например, не во всех ОС семейства Unix существуют файлы, связанные с сетевыми интерфейсами; Linux таких файлов не поддерживает.

Устройства, имеющие представление в виде файла, делятся на два типа: ***символьные*** (или ***потоковые***, они же ***байт-ориентированные***) и ***блочные*** (***блок-ориентированные***). Основные операции над байт-ориентированными устройствами — запись и чтение одного символа (байта) или их последовательности. В противоположность этому блок-ориентированные устройства воспринимаются как хранилище данных, разделённых на блоки фиксированного размера; основными операциями, соответственно, становятся чтение и запись заданного блока. В качестве примеров байт-ориентированных устройств можно назвать терминал (клавиатура и устройство отображения), принтер[18], манипулятор «мышь», звуковую карту.

Существует целый ряд символьных псевдоустройств, не имеющих физического воплощения. Так, в устройство `/dev/null` можно записать любую информацию, которая попросту игнорируется; попытка читать из этого устройства вызывает ситуацию «конец файла». Устройство `/dev/zero` позволяет прочитать любой объём данных, причём все прочитанные байты будут равны нулю; записывать в него, как и в `/dev/null`, можно что угодно, вся эта информация игнорируется. Устройство `/dev/full` похоже на `/dev/zero` тем, что при чтении выдаёт бесконечный поток нулевых байтов, а вот запись вызывает такую же ошибку, как при отсутствии места на диске: вызов `write` возвращает `-1`, переменная `errno` принимает значение `ENOSPC`. Устройство `/dev/random` выдаёт читающему процессу последовательность псевдослучайных чисел, и т. п.

[18] Точнее было бы сказать «принтерный порт».

В виде блок-ориентированных устройств обычно представляются диски и другие подобные устройства. Действительно, дисковые устройства обычно позволяют чтение или запись только целыми секторами, что обусловлено особенностями их физической реализации.

Блок-ориентированные и байт-ориентированные устройства представляют собой ещё два специальных типа файлов наряду с директориями и символическими ссылками. Файлы устройств можно открывать на чтение и запись, а к полученным дескрипторам применять вызовы `read` и `write`. Кроме того, блок-ориентированные устройства поддерживают позиционирование с помощью вызова `lseek`. Следует отметить, что позиционироваться можно в любую существующую точку устройства, которая, вообще говоря, не обязана находиться точно на границе сектора. Прочитать или записать также можно произвольное количество данных; если читаемый или записываемый фрагмент начинается и/или заканчивается где-то, кроме границ блоков, система возьмёт на себя работу по отбрасыванию лишней информации при чтении или по предварительному прочтению из затронутых секторов информации, находящейся за пределами записываемого фрагмента, для последующей записи секторов целиком.

Байт-ориентированные устройства операцию позиционирования не поддерживают. Это, пожалуй, наиболее заметное отличие байт-ориентированных устройств от блок-ориентированных с точки зрения прикладного программиста.

Ясно, что управление устройствами не может ограничиваться только операциями чтения и записи. Для многих устройств определены также дополнительные операции, специфические для данного устройства, такие как открытие и закрытие лотка привода CD-ROM, установка скорости обмена последовательного порта, низкоуровневая разметка гибких дисков, управление громкостью воспроизведения звука звуковой картой и т. п. Все операции этой категории выполняются с помощью системного вызова `ioctl`:

```
int ioctl(int fd, int request, ...);
```

Параметр `fd` задаёт дескриптор открытого файла устройства, параметр `request` — код нужной операции. Вызов может получать дополнительные параметры, требуемые для выполнения данной операции. Например, следующий код

```
int fd = open("/dev/cdrom", O_RDONLY|O_NONBLOCK);
ioctl(fd, CDROMEJECT);
ioctl(fd, CDROMCLOSETRAY);
```

откроет, а затем закроет лоток привода CD-ROM. Параметр `O_NONBLOCK` задаётся, чтобы избежать поиска диска в устройстве и ошибки в случае,

если диск в устройство не вставлен. Если же вставить в то же устройство музыкальный диск (audio CD), код

```
struct cdrom_ti cti;
cti.cdti_trk0 = 2;
cti.cdti_ind0 = 0;
cti.cdti_trk1 = 2;
cti.cdti_ind1 = 0;
ioctl(fd, CDROMPLAYTRAKIND, &cti);
```

заставит ваше устройство воспроизвести вторую дорожку диска.

5.3.6. Работа с содержимым каталогов

Как мы уже отмечали, ***каталог*** представляет собой дисковый файл специального типа, в котором содержится информация об именах файлов и соответствующих номерах индексных дескрипторов.

Любые изменения в содержимом каталогов система производит только сама — при создании, удалении и переименовании файлов (точнее, жёстких ссылок на файлы). Любые попытки что-то записать в каталог в обход этих операций, очевидно, привели бы к нарушению целостности файловой системы, так что ядро их не допускает. С другой стороны, довольно часто требуется узнать, что содержится в каталоге — например, это нужно команде `ls`, но, конечно, не только ей. Поскольку чтение информации не может само по себе нарушить её целостность, а всевозможные проверки прав доступа и прочие действия при извлечении информации из каталога во многом похожи на аналогичные действия при чтении обычного файла, **система позволяет открывать каталоги на чтение с помощью вызова** `open` (но только на чтение). Впрочем, вызов `read` на полученном дескрипторе всё равно работать не будет; название системного вызова, специально предназначенного для чтения из директорий, зависит от конкретной системы; в ОС Linux и FreeBSD этот вызов называется `getdents`, причём работает он в обеих системах одинаково, но формальные профили — типы принимаемых параметров — различаются. Использовать его напрямую в любом случае не следует, поскольку это требует знания внутренней структуры каталогов и к тому же существенно ограничивает переносимость программы на другие системы.

Библиотека языка Си предусматривает для анализа содержимого каталогов (директорий) функции `opendir`, `readdir` и `closedir`:

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

Загадочное значение типа `DIR*` играет здесь приблизительно ту же роль, что и `FILE*` для высокоуровневой работы с файлами: это указатель на некую структуру, создаваемую библиотекой в своей памяти, причём содержимое этой структуры нас волновать не должно, нам она интересна лишь как «нечто, обеспечивающее работу с директорией».

Функция `readdir` читает очередную запись из открытого с помощью `opendir` каталога и возвращает указатель на некую структуру, имеющую тип `struct dirent`. Сама эта структура находится там же, где и созданное библиотекой *нечто*, именуемое `DIR`, и существует в одном экземпляре для каждого открытого на чтение каталога, то есть, скорее всего, читая из одного и того же потока типа `DIR*`, вы каждый раз будете получать один и тот же адрес; это нужно учитывать, ведь содержимое структуры `dirent`, возвращённое при предыдущем обращении к `readdir`, окажется перезаписано следующим обращением.

Когда записи в каталоге заканчиваются, `readdir` возвращает `NULL`, после чего поток следует закрыть с помощью `closedir`, чтобы высвободить ненужный теперь файловый дескриптор.

Довольно своеобразно обстоят дела с полями структуры `dirent`. Можно совершенно точно сказать только одно: в структуре присутствует поле `d_name`, представляющее собой массив элементов типа `char`, но каков размер этого массива — в общем случае неизвестно. Библиотека поддерживает константу `NAME_MAX`, равную максимальной длине имени файла, хранимого в директории; в большинстве случаев это число `255`, но никакой гарантии этого не даётся. Кроме того, обычно структура `dirent` содержит поле `d_ino`, имеющее *какой-то* целочисленный тип (например, в системе автора книги это оказался `unsigned long`; системные заголовочные файлы обозначают этот тип именем `ino_t`, вводимым с помощью директивы `typedef`). В поле `d_ino` записывается номер индексного дескриптора (*i-node*). Как правило, структура `dirent` имеет и другие поля, но они специфичны для конкретной системы.

Например, следующая программа принимает в качестве параметра командной строки имя каталога (если параметр не задан, используется текущий каталог) и выдаёт список имён файлов в этом каталоге, по одному имени в строке:

```
/* poor_ls.c */
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
int main(int argc, char **argv)
{
    DIR *dir;
    struct dirent *dent;
    const char *name = ".";
    if(argc > 1)
        name = argv[1];
```

```
    dir = opendir(name);
    if(!dir) {
        perror(name);
        return 1;
    }
    while((dent = readdir(dir)) != NULL) {
        printf("%s\n", dent->d_name);
    }
    closedir(dir);
    return 0;
}
```

5.3.7. Отображение файлов в память

В ОС Unix предусмотрена возможность отображения содержимого некоторого файла в виртуальное адресное пространство процесса. В результате такого отображения появляется возможность работы с данными в файле как с обычными переменными в оперативной памяти, то есть, например, с помощью присваиваний. Отображение осуществляется системным вызовом `mmap`:

```
void *mmap(void *start, int length, int protection,
           int flags, int fd, int offset);
```

Перед обращением к `mmap` файл должен быть открыт с помощью `open`; вызов принимает дескриптор файла, подлежащего отображению, через параметр `fd`. Параметры `offset` и `length` задают соответственно позицию начала отображаемого участка в файле и его длину. Здесь нужно заметить, что и длина, и позиция должны быть кратны некоторому предопределённому числу, называемому *размером страницы*[19]. Его можно узнать с помощью функции

```
int getpagesize();
```

Параметр `protection` вызова `mmap` задаёт режим доступа к получаемому участку виртуальной памяти. Для этого служат константы `PROT_READ`, `PROT_WRITE` и `PROT_EXEC`, которые можно объединять операцией побитового «или». Как ясно из названия, первые две константы соответствуют доступу на запись и чтение. Третья позволяет передавать управление в область отображения, то есть исполнять там код; это используется, например, при подгрузке динамических библиотек. Существует также константа `PROT_NONE`, соответствующая запрету доступа любого вида. Задаваемый параметром `protection` доступ должен быть совместим с

[19]Размер страницы для `mmap` в общем случае может не совпадать с размером страницы виртуальной памяти; так, на некоторых старых версиях Linux вызов `getpagesize` возвращал число 16, хотя таких виртуальных страниц нет ни на одной архитектуре.

режимом, в котором был открыт файл: так, если файл открыт в режиме «только чтение», то есть в вызове `open` был использован флажок `O_RDONLY`, то попытка отобразить файл в память с режимом, допускающим запись, вызовет ошибку.

В качестве параметра `flags` указывают либо `MAP_SHARED`, либо `MAP_PRIVATE` (в этом случае изменения, производимые в виртуальном адресном пространстве, никак на файле не отразятся). Кроме того, к любому из этих двух флагов можно добавить через операцию побитового «или» флажки дополнительных опций. Среди этих опций есть `MAP_ANONYMOUS`, позволяющая обойтись без файла, то есть создать просто область виртуальной памяти; в этом случае параметры `fd` и `offset` игнорируются.

Память, выделенная с помощью `mmap` с одновременным указанием `MAP_ANONYMOUS` и `MAP_SHARED`, отличается от обычной памяти тем, что при создании нового процесса она становится доступна из обоих процессов, то есть изменения, сделанные в такой памяти порождённым процессом, будут доступны родительскому и наоборот; это называется ***разделяемой памятью*** (англ. *shared memory*). Разделяемая память иногда используется для организации взаимодействия между процессами; надо сказать, что работа с разделяемой памятью порождает целый ряд неочевидных проблем, которым будет посвящена отдельная (седьмая) часть нашей книги.

Параметр `start` позволяет указать системе, в каком месте нашего адресного пространства нам хотелось бы видеть новую область памяти. Обычно пользовательские программы не используют эту возможность; в качестве параметра `start` можно передать `NULL`, тогда система сама выберет свободную область виртуального адресного пространства.

Вызов `mmap` возвращает указатель на созданную область виртуальной памяти. Обычно этот указатель преобразуют к другому типу, например к `char*`. В случае ошибки `mmap` возвращает значение `MAP_FAILED`, равное `-1`, преобразованной к типу `void*`.

Приведём пример:

```
int fd, pgs;
char *p;
int size = 4096;
pgs = getpagesize();
size = ((size-1) / pgs + 1) * pgs;
  /* минимальное целое число, большее либо равное
     исходному и при этом кратное размеру страницы */
fd = open("file.dat", O_RDWR);
if(fd == -1) {
    /* ... обработка ошибки ... */
}
p = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

```
if(p == MAP_FAILED) {
    /* ... обработка ошибки ... */
}
```

После выполнения этих действий выражение `p[25]` будет равно значению 26-го байта в файле `"file.dat"`, причём операция присваивания `p[25] = 'a'` занесёт в этот байт символ `'a'`.

Отменить отображение, созданное вызовом `mmap`, можно с помощью вызова `munmap`:

```
int munmap(void *start, int length);
```

Физическую запись в файл изменений, сделанных в области отображения, система может произвести не сразу. С помощью вызова `msync` можно заставить систему произвести запись немедленно; изучение этого вызова оставляем читателю для самостоятельной работы.

Отметим, что вызов `mmap`, изначально предназначавшийся для работы с содержимым дисковых файлов, в наше время воспринимается скорее как универсальный интерфейс для создания областей виртуальной памяти. Так, в системах семейства BSD именно этот вызов использует функция `malloc`, чтобы затребовать у системы очередную порцию памяти. Например, после выполнения

```
p = mmap(NULL, size, PROT_READ|PROT_WRITE,
         MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
```

указатель `p` будет содержать адрес новой области виртуальной памяти, полученной непосредственно от операционной системы.

5.4. Процессы

5.4.1. Процесс: что это такое

Как уже говорилось, пользовательские программы выполняются под управлением операционной системы в виде ***процессов***; иначе говоря, ***процесс*** — это некая сущность, создаваемая ядром операционной системы, чтобы выполнять пользовательскую программу.

Как это часто бывает в технических дисциплинах, то или иное понятие спонтанно формируется в среде специалистов, обретает устойчивые очертания, а когда кто-то спохватывается, что новый термин не имеет строгого определения, оказывается уже поздно: термин живёт своей жизнью и совершенно не желает укладываться в рамки любых определений. Именно так произошло и с понятием процесса: каждый программист понимает, что́ это такое, но никто не может дать такое определение, для которого нельзя было бы с ходу предъявить контрпример. Какое

бы определение процесса ни давалось, всегда можно придумать что-то, что процессом не является, но под определение подходит, или что-то, что не подходит, но при этом является процессом.

В самом первом приближении можно считать, что процесс — это *программа, которую запустили на выполнение*; выполняемая прямо сейчас программа, кроме собственно машинного кода, её составляющего, обладает также текущими значениями переменных, текущей позицией исполнения и прочими особенностями, образующими её ***состояние***. Кроме того, запущенная программа может владеть какими-то объектами, находящимися вне её — например, открытыми файлами; как уже говорилось, с процессом связаны определённые *полномочия*. Дойдя до этого момента, мы можем заметить, что наше «определение в первом приближении» уже опровергнуто: ядро операционной системы, вне всякого сомнения, представляет собой программу и даже выполняется, но *процессом не является*, с ядром не связаны ни полномочия (они у него безграничны, к тому же если спросить, кто следит за полномочиями процесса, правильным ответом будет как раз «ядро операционной системы», а за полномочиями самого ядра следить некому), ни состояние выполнения (в зависимости от подхода к реализации ядра в нём может существовать несколько контекстов выполнения одновременно). Впрочем, это даже не столь важно: просто любой программист, имеющий дело с процессами, скажет, что исполняющееся на машине ядро операционной системы — это не процесс.

Между прочим, то же самое можно сказать про специфические ситуации, когда операционной системы вообще нет; именно так, на «голом железе»[20], выполняются прошивки микроконтроллеров. В такой вычислительной системе программа вообще одна, ей не к кому обращаться, её некому контролировать. Несмотря на то, что программа, очевидно, запущена и выполняется, то есть обладает состоянием, называть это явление процессом никому в голову не приходит.

Исправить ситуацию позволяет небольшое уточнение: *процесс — это программа, которую запустили на выполнение под управлением операционной системы*. В таком виде определение можно принять в качестве рабочего, но и оно окажется опровергнуто, когда мы обнаружим, что в системах семейства Unix *можно в рамках одного процесса заменить одну выполняющуюся программу на другую*. Впрочем, это не страшно; следует, по-видимому, помнить, что строгого определения процесса нет (и не надо), но при этом рассмотрение процесса как «программы, которая выполняется под управлением ОС» позволяет нам приблизиться к пониманию действительной сущности процесса.

Позволим себе упомянуть ещё один подход к пониманию процесса. Для того, чтобы запустить в системе на выполнение некую программу, ядро должно, очевидно, создать в своей памяти определённую структу-

[20] Соответствующий английский термин — *on bare metal* или *on bare bones*.

ру данных, или, говоря более «научно», некий объект. В этом объекте (структуре данных) будут содержаться, например, сведения о памяти, которая выделена запущенной программе, о её полномочиях, об открытых ею файлах (потоках ввода-вывода) и многое другое. В этой же структуре данных в те периоды, когда программа снята с выполнения (но не навсегда), будет храниться содержимое регистров центрального процессора: при снятии программы с выполнения ядро скопирует содержимое регистров в наш объект, а при возобновлении выполнения — заполнит регистры информацией, ранее сохранённой в объекте. Программисты, пишущие части ядра операционной системы, часто говорят, что этот объект ядра и есть процесс; иначе говоря, процесс — это структура данных, создаваемая в ядре операционной системы для поддержки выполнения пользовательской программы. Конечно, это тоже никоим образом не определение, это скорее отражение определённой точки зрения.

5.4.2. Свойства процесса

Не имея возможности дать строгое определение процесса, мы восполним этот пробел, обсудив, какими процесс обладает *свойствами*. Прежде всего отметим, что каждый процесс в системе имеет уникальный идентификатор — целое число, называемое «`pid`» от слов *process identifier*. В ОС Unix с процессом связан также параметр `ppid`, равный идентификатору родительского процесса, т. е. процесса, породившего данный, если этот процесс ещё существует; если родительский процесс завершается раньше порождённого, `ppid` потомка становится равен 1. Идентификация процесса на этом не заканчивается: каждый процесс в ОС Unix относится к некой *группе* в рамках *сеанса*, которые обозначаются *идентификатором группы процессов* (`pgid`, *process group identifier*) и *идентификатором сеанса* (`sid`, *session identifier*). Сеансы и группы процессов мы подробно рассмотрим в §5.6.2.

При обсуждении задач, решаемых операционными системами, мы в §5.1.1 специально подчеркнули важность *разграничения полномочий*, упомянув при этом, что полномочиями в системе на самом деле обладает не пользователь, а процесс, запущенный от его имени. В системах семейства Unix *информация, определяющая полномочия процесса*, включает *идентификатор пользователя* (`uid`, *user identifier*), *идентификатор группы пользователей* (`gid`, *group identifier*), а также *эффективные* идентификаторы пользователя и группы (`euid` и `egid`, от слова *effective*). В большинстве случаев эффективные идентификаторы совпадают с обычными; примером случая, когда это не так, служат так называемые suid-программы, то есть программы, выполняемые с правами пользователя, владеющего исполняемым файлом данной программы, а не того пользователя, который программу запустил. К числу таких программ

относится, например, `passwd` — хорошо знакомая вам программа для смены пароля; впрочем, на некоторых системах `passwd` уже перестала быть suid'ной.

Отметим, что узнать свои значения `uid`, `gid`, `euid`, `egid`, `pid`, `ppid`, `sid` и `pgid` процесс может с помощью системных вызовов, которые так и называются `getpid`, `getuid`, `getgid` и т. д. Все эти функции вызываются без параметров и возвращают значение соответствующего свойства процесса; ошибочные ситуации для этих вызовов не предусмотрены. Параметры `pid` и `ppid` (идентификатор процесса и его предка) измененить нельзя. Манипуляция параметрами `sid` и `pgid` будет рассмотрена позже, при обсуждении сеансов и групп процессов. Параметры `uid`, `gid`, `euid`, `egid`, идентифицирующие полномочия процесса, в некоторых случаях могут меняться; об этом речь пойдёт при рассмотрении полномочий процессов (§5.4.10).

Продолжая рассмотрение свойств процессов, мы можем догадаться, что у каждого процесса имеется выделенная ему память, ведь иначе он не мог бы выполняться; как мы уже знаем из опыта работы на языке ассемблера, память процесса состоит из нескольких *сегментов* или *секций*. В секции кода хранится в виде машинного кода собственно программа, выполняющаяся в данном процессе. Изменять содержимое этой области памяти процесс не может, что позволяет при запуске нескольких экземпляров одной и той же программы держать в памяти только одну копию её кода. В секции данных располагаются глобальные переменные и небезызвестная *куча*, из которой выделяются области памяти для динамических переменных. Наконец, секция стека используется для организации хорошо знакомого нам аппаратного стека, хранящего локальные переменные подпрограмм и адреса возврата из них. В пространстве памяти, принадлежащем процессу, могут присутствовать и другие секции, точный список которых зависит от конкретной системы.

Очень важной характеристикой процесса является **состояние регистров центрального процессора**, включая счётчик команд (он же указатель инструкции), регистр флагов, указатель стека и все регистры общего назначения. Когда процесс по тем или иным причинам находится вне состояния выполнения (то есть он либо заблокирован, либо готов к выполнению, но процессор занят чем-то другим), содержимое регистров ЦП для этого процесса хранится в специальной структуре данных в ядре; когда процесс снова ставится на выполнение, данные из этой структуры копируются обратно в регистры.

За обеспечение иллюзии одновременного исполнения процессов отвечает подсистема ядра, которая называется ***планировщиком времени центрального процессора*** или, для краткости, просто планировщиком[21]. С точки зрения планировщика процесс обладает, во-первых,

[21] Отметим, что соответствующий англоязычный термин — *CPU scheduler*, а не «planner», что иногда встречается в английских текстах русских авторов.

логическим значением *готовности*, показывающим, нужно ли данный процесс вообще ставить на выполнение (иными словами, *готов* ли он к дальнейшему исполнению); во-вторых, процессу приписывается некий *приоритет*, используемый планировщиком для выбора процесса из нескольких готовых к выполнению, а в некоторых системах ещё и для определения величины кванта времени. Например, приоритет может состоять из двух чисел, называемых ***статическим приоритетом*** и ***динамическим приоритетом***. Сложение этих двух чисел даёт общий приоритет процесса. Статический приоритет процессу задаётся извне, тогда как значение динамического приоритета планировщик меняет в зависимости от поведения процесса; пока процесс ожидает выполнения в состоянии готовности, его динамический приоритет повышается, когда же процесс находится в состоянии выполнения на центральном процессоре, его динамический приоритет снижается. Каковы бы ни были значения статической составляющей приоритета двух процессов, если один из них выполняется, а другой ожидает в очереди, за счёт изменения динамического приоритета рано или поздно приоритет второго окажется выше, чем приоритет первого, что приведёт к их «рокировке».

Динамическая составляющая приоритета остаётся внутренним делом планировщика и извне не контролируется, а вот статическая, которая в ОС Unix обычно называется *nice value*, может быть изменена с помощью системных вызовов `nice` и `setpriority`, что позволяет управлять планировщиком. Например, процессам, выполняющим долгие расчёты, можно снизить приоритет до минимального, чтобы они не мешали другим процессам, а занимали процессор только тогда, когда он всё равно свободен; в то же время процессам, обслуживающим внешние запросы, критичные по времени отклика, можно назначить высокий приоритет и т. д. Подробности мы приведём в §5.4.8.

Как мы уже неоднократно обсуждали, при запуске программы на выполнение в операционных системах семейства Unix указываются ***аргументы командной строки***, первым из которых обычно является имя самой программы, по которому она была вызвана. Структура данных, содержащая аргументы командной строки, показана на рис. 5.3 слева. В каком конкретно месте адресного пространства процесса располагаются аргументы командной строки, зависит от конкретной системы; несомненно одно: командная строка процесса является его свойством, причём таким, о котором совершенно невозможно забыть: к примеру, командная строка каждого процесса показывается в выдаче команды `ps`, печатающей список процессов в системе[22].

Вместе с командной строкой заслуживает упоминания ***окружение*** процесса. С переменными окружения мы уже сталкивались в первом

[22]Напомним, что увидеть *все* процессы системы можно с помощью команды `ps ax` или `ps axu`.

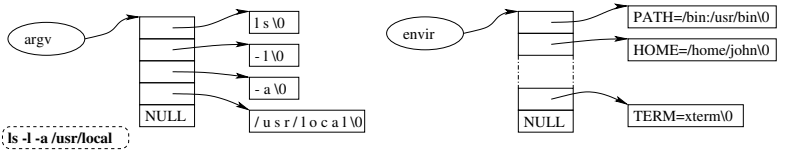

Рис. 5.3. Структуры данных командной строки и окружения

томе (§ 1.4.14); напомним, что каждая такая переменная имеет имя и значение, причём и то, и другое представляет собой строки; в имени не может быть символов пробела и знака равенства, в строке-значении символы можно использовать любые. Структура данных, содержащая имена и значения переменных окружения, очень похожа на структуру командной строки (рис. 5.3, справа). Обычно новые запускаемые программы наследуют окружение от тех, кто их запустил, что позволяет передавать определённые настройки системы всем имеющимся процессам с возможностью некоторые процессы запустить с изменёнными настройками. Забегая вперёд отметим, что наследование окружения обусловлено не свойством операционной системы, а библиотечными функциями, через которые обычно осуществляется запуск новых программ; с точки зрения ядра окружение для каждой запускаемой программы задаётся своё.

Окружение доступно в программах на Си через глобальную переменную

```
extern char **environ;
```

Поскольку окружение располагается в памяти процесса, системные вызовы для работы с окружением не нужны. Для манипуляции переменными окружения служат библиотечные функции `getenv`, `setenv` и `unsetenv`:

```
char *getenv(const char *name);
int setenv(const char *name, const char *value, int overwrite);
void unsetenv(const char *name);
```

Функция `getenv` возвращает (в виде указателя на строку) значение переменной окружения, имя которой задаётся аргументом `name`. Если такой переменной в окружении нет, возвращается значение `NULL`. Очень важно произвести проверку на `NULL` перед анализом возвращенной строки. Никто не может гарантировать наличие какой бы то ни было переменной в окружении процесса, даже если речь идет о стандартных переменных, включая `PATH`. Функция `setenv` устанавливает новое значение переменной, причём если такой переменной не было, значение устанавливается в любом случае, если же соответствующая переменная в окружении уже есть, новое значение устанавливается только

при ненулевом значении параметра `overwrite`; иначе говоря, параметр `overwrite` разрешает или запрещает изменять значение переменной окружения, если она есть. Функция `unsetenv` удаляет переменную с заданным именем из окружения.

К свойствам процесса в ОС Unix относятся также ***текущий каталог*** и ***корневой каталог***. С текущим каталогом всё просто: каждый процесс считается «находящимся» в одном из каталогов файловой системы; от текущего каталога, как мы видели ранее, отсчитываются «относительные пути» — имена файлов, начинающиеся с любого символа, кроме «/»; значение текущего каталога, таким образом, влияет на работу всех системных вызовов, в которые через параметры передаются имена файлов. Процесс может в любой момент сменить свой текущий каталог.

С корневым каталогом дела обстоят несколько сложнее. Unix позволяет ограничить файловую систему, видимую процессу и всем его потомкам, частью дерева каталогов, имеющей общий корень. Например, если установить процессу корневой каталог `/foo`, то под именем `/` процесс будет видеть каталог `/foo`, а под именем `/bar` — каталог `/foo/bar`. Каталоги за пределами `/foo` процессу и всем его потомкам вообще не будут видны ни под какими именами. Это используется для запуска отдельных программ в безопасном варианте — так, чтобы они не могли получить доступ ни к каким файлам кроме тех, которые предназначены специально для них.

Процесс может сменить текущий каталог с помощью вызова

```
int chdir(const char* path);
```

подав в качестве параметра полный путь нового каталога либо путь относительно текущего каталога. Строка `".."` означает каталог уровнем выше: например, `/usr/local/share/..` — это то же самое, что `/usr/local`.

Смена корневого каталога осуществляется вызовом

```
int chroot(const char* path);
```

После выполнения этого вызова каталоги за пределом нового корневого перестают быть видны или каким-либо образом доступны процессу и всем его потомкам. Операция смены корневого каталога необратима. Вызов `chroot` могут выполнять только процессы, имеющие права администратора системы (пользователя `root`), то есть имеющие нулевой `uid`. Отметим, что **как текущий, так и корневой каталог нельзя изменить для другого процесса; процесс может сменить их только для самого себя**.

Ещё одно важное свойство процесса связано с открытыми потоками ввода-вывода: это ***таблица файловых дескрипторов***. Каждый поток ввода-вывода на самом деле представляет собой довольно сложную

структуру данных в памяти ядра, причём конкретный вид этой структуры существенно зависит от природы потока — для потока, связанного с обычным дисковым файлом и для какого-нибудь сетевого сокета ядру приходится помнить совершенно разные сведения. Как мы уже неоднократно видели, с точки зрения процесса файловый дескриптор — это просто номер, причём это даже не номер потока ввода-вывода в масштабах всей системы, а номер открытого потока для данного процесса: так, дескрипторы `0`, `1` и `2` есть практически в каждом процессе, но связаны с совершенно разными потоками.

Таблица файловых дескрипторов процесса задаёт соответствие собственно файловых дескрипторов (то есть небольших целых чисел, с помощью которых различаются между собой открытые потоки ввода-вывода) объектам внутри ядра ОС, содержащим всё необходимое для функционирования этих потоков. Система предоставляет процессу широкий набор инструментов, влияющих на его таблицу дескрипторов. Два наиболее очевидных способа изменения содержимого этой таблицы — это открытие файла с вызовом `open` и закрытие его вызовом `close`. Позже мы узнаем, во-первых, другие способы создания потоков ввода-вывода, при которых новые дескрипторы появляются без использования `open`; во-вторых, в §5.4.9 мы рассмотрим системные вызовы, напрямую манипулирующие содержимым таблицы дескрипторов.

Обсуждая системный вызов `open`, мы упоминали параметр `umask`; он тоже является в ОС Unix свойством процесса. Изменить его можно с помощью системного вызова, который так и называется `umask`:

```
int umask(int mask);
```

Этот системный вызов всегда завершается успешно и возвращает предыдущее значение параметра `umask`.

Несколько позже, изучая так называемые ***сигналы***, мы увидим, что к свойствам процесса относится также ***диспозиция сигналов***, то есть набор указаний операционной системе, что следует делать, когда данному процессу приходит тот или иной сигнал. Также к свойствам процесса относятся *счётчики потреблённых ресурсов* (процессорного времени, памяти и т. п.) и некоторые другие свойства, рассмотрение которых мы оставим за рамками нашей книги.

5.4.3. Порождение процесса

Единственный способ порождения процесса в ОС Unix — это создание копии существующего процесса[23]. Для этого используется системный вызов `fork`:

[23] Некоторые системы семейства Unix имеют альтернативные возможности, такие как `clone` в ОС Linux, но эти возможности специфичны для каждой системы и их прямое использование не рекомендуется.

```
int fork(void);
```

В результате создается новый процесс, являющийся точной копией родительского, за исключением следующих различий:

- порождённый процесс имеет свой идентификатор (`pid`), естественно, отличающийся от идентификатора родителя;
- параметр `ppid` порождённого процесса равен `pid`'у родительского процесса;
- счётчики потреблённых ресурсов порождённого процесса сразу после `fork` равны нулю;
- выполнение обоих процессов (родительского и порождённого) продолжается с первой инструкции, следующей сразу за вызовом функции `fork` (чаще всего это присваивание возвращаемого ею значения какой-либо переменной), причём в родительском процессе `fork` возвращает `pid` порождённого процесса, а в порождённом — «возвращает»[24] число 0.

Например, программа

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("Hello\n");
    return 0;
}
```

напечатает восемь строк `"Hello"`. В самом деле, после первого вызова `fork` процессов станет два, каждый из них продолжит выполнение со следующей после вызова операции, в данном случае — со второго вызова `fork`, то есть каждый из двух процессов снова породит свою копию; имеющиеся четыре процесса продолжат выполнение с третьего `fork`, так что их станет восемь, и каждый из них выполнит `printf`.

В этом примере все процессы делали одно и то же. На практике обычно требуется, чтобы порождённый процесс делал не то же самое, что его родитель, а что-то другое. Для этого используется обычный `if`, проверяющий возвращённое вызовом `fork` значение, например, так:

```
p = fork();
if(p == 0) {
    /* действия порождённого процесса */
} else {
```

[24] Слово «возвращает» здесь взято в кавычки, поскольку на самом деле возвращать там нечего, ведь процесс только что появился; но всё выглядит именно так, как будто функция `fork` вернула значение.

```
    /* действия родительского процесса */
}
```

или так:

```
p = fork();
if(p == 0) {
    /* действия порождённого процесса */
    exit(0);
}
/* действия родительского процесса */
```

После обращения к `fork` оба процесса работают параллельно («одновременно»); например, если в обоих процессах программа предусматривает выполнение какого-то действия, то без специальных ухищрений нельзя будет предсказать, какой из процессов успеет своё действие сделать раньше. Рассмотрим фрагмент программы, который порождает процесс-потомок и выдаёт два сообщения, одно из порождённого процесса, второе из родительского:

```
p = fork();
if(p == 0) {
    printf("I'm the child\n");
} else {
    printf("I'm the parent\n");
}
```

Возможны две ситуации. В первой порождённый процесс не успевает начать исполнение прежде, чем родительский дойдёт до вызова функции `printf`. Например, системе могло не хватить памяти и порождённый процесс был создан в откачанном состоянии. В этом случае сначала будет напечатана фраза «I'm the parent», затем — фраза «I'm the child».

Вторая ситуация противоположна. К примеру, у родительского процесса может сразу после выполнения вызова `fork` истечь квант времени. При этом порождённый процесс, которому на сей раз хватило памяти, получит управление и успеет за свой квант времени выполнить печать. Тогда фразы появятся на экране в обратном порядке.

Ситуации, в которых результат зависит от конкретной последовательности независимых событий (обычно событий из работающих параллельно процессов) называются *ситуациями гонок*[25]. Ситуации гонок часто возникают в параллельном программировании, то есть при наличии более чем одного потока управления. К их возникновению следует относиться очень внимательно, поскольку в некоторых случаях неучтённые ситуации гонок могут приводить к ошибкам и даже проблемам в безопасности. К этому вопросу

[25] Соответствующий англоязычный термин — *race condition*. В русских переводах встречается также «ситуация состязаний».

мы вернёмся в седьмой части нашей книги, которая будет посвящена параллельному программированию.

Отметим, что после вызова `fork` оба процесса используют один и тот же сегмент кода; это возможно, т. к. сегмент кода не может быть модифицирован. Остальная память процесса, за исключением некоторых особых случаев, копируется[26]. Это означает, в частности, что в порождённом процессе присутствуют все переменные, существовавшие в родительском процессе, причём изначально они имеют те же значения, но изменения переменных в родительском процессе никак не отражаются на порождённом и наоборот. Копированию подвергаются открытые дескрипторы файлов, установленные обработчики сигналов и т. п.

К особым случаям, когда для части памяти процесса не создаётся копия при выполнении `fork`, относятся фрагменты виртуальной памяти, созданные вызовом `mmap` (см. §5.3.7) с использованием флага `MAP_SHARED`. Рассмотрим пример:

```
int fd, pgs;
int *ptr;
int size = 4096;
pgs = getpagesize();
size = ((size-1) / pgs + 1) * pgs;
ptr = mmap(NULL, size, PROT_READ|PROT_WRITE,
           MAP_SHARED|MAP_ANONYMOUS, 0, 0);
if(ptr == MAP_FAILED) {
    /* ... обработка ошибки ... */
}
if(fork() == 0) {
    /* ... порождённый процесс ... */
} else {
    /* ... родительский процесс ... */
}
```

В этом примере родительский и порождённый процессы имеют доступ к одному и тому же массиву целых чисел длиной 1024 элемента, если считать, что `int` занимает 4 байта. Массив доступен через указатель `ptr`, так что если один из процессов сделает присваивание `ptr[77] = 120`, то в **обоих** процессах выражение `ptr[77]` будет иметь значение 120.

Корректная работа с разделяемыми данными требует определённой квалификации. Если не соблюдать при этом целый ряд весьма неочевидных правил, ваша программа будет демонстрировать неожиданное поведение в самые неподходящие моменты. Работе с разделяемыми данными будет посвящена отдельная часть нашей книги; пока просто помните, что разделяемая память — это отнюдь не так просто, как может показаться на первый взгляд.

[26] В современных системах обычно процессы продолжают разделять страницы памяти до тех пор, пока один из них не попытается ту или иную страницу модифицировать: в этом случае создается копия страницы.

5.4.4. Замена выполняемой программы

Запустить на выполнение программу в ОС Unix можно путём *замены выполняемой программы* в рамках одного процесса. Концепция «замены программы в рамках процесса» может создать несколько удручающее впечатление, поскольку, как мы обсуждали в начале этой главы, процесс — это некая сущность, создаваемая операционной системой, чтобы выполнять программы, и при таком подходе к объяснению вполне естественным оказывается восприятие *программы* как «главной сущности», а процесса — как чего-то вторичного по отношению к программе. В принципе это так и есть: процессы созданы для программ, а не наоборот. Несмотря на это, как мы сейчас увидим, работающую программу можно заменить на другую программу, которая начнёт своё выполнение с её начала (например, с функции `main` для программ, написанных на Си), а *процесс* — то есть тот объект, который создала для выполнения программы операционная система — останется при этом тот же, который был. Больше того, в системах Unix это *единственный* способ запуска программ, другого просто нет.

Такой подход на первый взгляд может показаться (и действительно многим кажется) нелогичным, но у него есть несомненные достоинства. Дело в том, что при замене программы процесс не просто «формально» остаётся тем же самым — это действительно продолжает существовать тот же объект ядра операционной системы, что проявляется в сохранении целого ряда свойств процесса, таких как полномочия процесса, открытые потоки ввода-вывода и многое другое. Это позволяет произвести предварительную настройку, затрагивая лишь те свойства процесса, которые нужно поменять, и не отвлекаясь на постороннее.

Замена программы, выполняющейся в рамках процесса, на новую программу производится с помощью системного вызова `execve`:

```
int execve(const char *name, char* const *argv,
                             char* const *envir);
```

Параметр `name` задаёт исполняемый файл программы, которую нужно запустить на выполнение вместо текущей; файл можно задать как полным путём, так и относительно текущего каталога — как, собственно, и во всех системных вызовах, использующих имена файлов. Параметры `argv` и `envir` задают соответственно командную строку и окружение для запускаемой программы в виде адресов структур данных, показанных на рис. 5.3 (см. стр. 74). Для удобства программирования существует ещё несколько функций семейства `exec`, реализованных в библиотеке через вызов `execve`. Начнём с функции `execv`:

```
int execv(const char *name, char* const *argv);
```

От вызова `execve`, как можно заметить, эта функция отличается отсутствием параметра `envir`. Окружение для запускаемой программы в этом случае берётся такое же же, как было — попросту говоря, наследуется.

Следующая полезная функция имеет такой же прототип, как и `execv`:

```
int execvp(const char *name, char* const *argv);
```

Эта функция отличается от предыдущих подходом к обработке первого параметра: она умеет *отыскивать запускаемую программу по её имени в «системных» директориях*, то есть в директориях, перечисленных в переменной `PATH`. Поиск производится, если в строке, переданной параметром `name`, нет ни одного символа «/»; так, если значение переменной `PATH` содержит директорию `/bin`, то вызвать программу `/bin/ls` можно просто по имени «`ls`», не указывая полный путь. Если в первом параметре есть хотя бы один символ «/», функция `execvp` работает точно так же, как и `execv`. Между прочим, именно эта особенность `execvp` — причина того, что для запуска свежеоткомпилированных программ в ходе наших упражнений мы вынуждены пользоваться конструкциями вроде `./prog` вместо просто `prog`; префикс «`./`» обеспечивает наличие слэша в параметре `name`, отключая тем самым поиск по директориям из `PATH` и превращая имя программы в обычное относительное имя файла.

Часто встречаются случаи, когда уже на этапе написания исходной программы нам известно точное количество параметров командной строки для программы, которую мы собираемся запустить с помощью `exec`. Если это так, нам нет необходимости самим формировать массив указателей на элементы командной строки, как это требуется для функций семейства `exec`, рассмотренных выше. Вместо этого можно использовать функцию `execl` или `execlp`, которые сформируют этот массив за нас:

```
int execl(const char *name, const char *argv0, ...);
int execlp(const char *name, const char *argv0, ...);
```

Эти функции получают произвольное число аргументов, первый из которых задаёт исполняемый файл, остальные — аргументы командной строки. Чтобы функция «знала», где остановиться, после последнего слова командной строки нужно добавить ещё один параметр со значением `NULL`[27]. Следует обратить внимание, что командная строка включает нулевой элемент, под которым подразумевается имя самой программы; таким образом, аргумент `argv0` — это не первый аргумент командной строки, а нечто имеющее отношение к имени программы, в большинстве случаев значение `argv0` попросту совпадает с `name`. Различие между

[27] В результате придирчивого анализа текстов разнообразных стандартов многие программисты приходят к выводу, что использовать следует не `NULL`, а выражение `((char*)0)`. Судя по всему, в реально существующих системах никакой разницы нет.

execl и execlp в том, что первая требует указания явного пути к исполняемому файлу, тогда как вторая выполняет поиск по переменной PATH, подобно тому, как это делает execvp. Всё, что было сказано выше относительно первого аргумента execvp, точно так же применимо и к первому аргументу execlp.

Допустим, требуется выполнить команду ls -l -a /var. Это можно сделать, например, так:

```
char *cmdline[] = { "ls", "-l", "-a", "/var", NULL };
execvp("ls", cmdline);
```

либо так:

```
execlp("ls", "ls", "-l", "-a", "/var", NULL);
```

Повторим, что все функции семейства exec *заменяют* в памяти процесса выполнявшуюся (и вызвавшую exec) программу на другую, указанную в параметрах вызова. Поэтому в случае успеха эти функции управление уже не возвращают: в самом деле, программы, в которую можно было бы вернуть управление, уже нет, вместо неё работает новая программа. В случае ошибки возвращается значение -1, но проверять его не обязательно: сам факт возврата управления свидетельствует об ошибке. Чаще всего после любой из функций exec следующая строка программы представляет собой вызов знакомой нам функции perror. Поскольку в большинстве случаев exec выполняется в процессе, созданном специально для этого, а после ошибки этот процесс уже больше не нужен, после perror ставится вызов функции exit, завершающей процесс. **Если после вызова функции семейства** exec **в тексте программы мы видим что-то отличное от связки** perror/exit**, в большинстве случаев здесь что-то не так;** ситуации, когда после exec не ставятся вызовы этих двух функций, редки и достаточно необычны.

Строго говоря, следует учитывать, что функция exit перед завершением процесса может сделать что-то ещё; в подавляющем большинстве случаев это «что-то ещё» сводится к вытеснению буферов вывода в функциях из stdio.h. Если вы ухитритесь вызвать fork, когда некоторые из ваших высокоуровневых потоков вывода содержат неотправленные данные, эти данные окажутся скопированы вместе со всеми данными родительского процесса, а функция exit, вызванная из порождённого процесса, произведёт их вытеснение. Родительский процесс об этом, разумеется, ничего знать не будет и рано или поздно, скорее всего, вытеснит свой экземпляр скопированных данных. Результат может получиться довольно своеобразный.

Основываясь на этом соображении, некоторые авторы рекомендуют после exec использовать _exit, а не обычный exit, то есть обращаться сразу же к ядру системы с просьбой немедленно завершить ваш процесс без всякой предварительной подготовки. Рекомендация сама по себе правильная, только следует учесть, что диагностический поток, в который выводит сообщение

функция perror, тоже буферизуется, и если этот поток окажется перенаправлен куда-то (не будет связан с терминалом), то при завершении процесса с помощью _exit данные из его буфера так и не будут вытеснены — попросту говоря, сообщение, выданное perror'ом, так и не покинет пределы своего процесса и сгинет вместе с ним (напомним, что выдача перевода строки приводит к вытеснению буфера только если вывод идёт на терминал).

По-видимому, «совсем правильной» будет примерно такая последовательность вызовов:

```
execlp("ls", "ls", "-l", NULL);
perror("ls");
fflush(stderr);
_exit(1);
```

Подробное обсуждение различных вариантов завершения процесса вы найдёте в следующем параграфе.

Отметим, что **открытые файловые дескрипторы при выполнении exec остаются открытыми**, что позволяет перед запуском внешней программы произвести манипуляции с дескрипторами. Это свойство exec используется для реализации хорошо знакомых нам перенаправлений ввода-вывода.

Из этого правила есть одно исключение. На файловом дескрипторе может быть установлен флаг FD_CLOEXEC (*close-on-exec*), и в этом случае файл — точнее, один отдельно взятый дескриптор — будет закрыт при успешно отработавшей замене программы.

Наиболее общепринятым способом установки этого флага можно считать вызов fcntl (см. стр. 57). В отличие от флагов статуса, таких как уже известные нам O_APPEND или O_NONBLOCK, флаг *close-on-exec* относится не к потоку ввода-вывода, а к конкретному файловому дескриптору, связанному с этим потоком. Таких дескрипторов, как мы уже понимаем, может стать больше одного, например, после выполнения вызова fork; при обсуждении перенаправления ввода-вывода мы узнаем о других способах «размножения» дескрипторов, относящихся к одному потоку ввода-вывода. Так или иначе, флаги статуса, которые мы обсуждали раньше и которые можно изменить с помощью команды F_SETFL вызова fcntl, относятся к потоку ввода-вывода и действуют вне зависимости от того, через какой из, возможно, нескольких дескрипторов этого потока мы работаем; флаг *close-on-exec*, напротив, относится к конкретному дескриптору.

В связи с этим для работы с флагом *close-on-exec* вызов fcntl предусматривает отдельные команды — F_SETFD и F_GETFD. Изначально предполагалось, что эти команды будут работать с флагами, относящимися к дескриптору (в отличие от потока), но в итоге был введён всего один такой флаг — FD_CLOEXEC. Установить его можно, выполнив

```
fcntl(fd, F_SETFD, FD_CLOEXEC);
```

Узнать, установлен флаг или нет, можно с помощью fcntl(fd, F_GETFD) (такой вызов вернёт значение FD_CLOEXEC, если флаг установлен, и 0 — если нет), но это почти никогда не нужно.

С некоторых пор в наиболее новых версиях Unix-систем флаг *close-on-exec* можно установить в момент открытия файла, добавив во второй параметр вызова open константу O_CLOEXEC, но это поддерживается не везде и не факт, что будет хорошей идеей эту возможность использовать.

5.4.5. Завершение процесса

В системах семейства Unix предусмотрено два способа завершения процесса: самостоятельное и принудительное. В первом случае процесс сам обращается к ядру операционной системы с просьбой завершить его, и эта просьба немедленно удовлетворяется. Во втором случае кто-то — другой процесс либо сама операционная система — присылает процессу ***сигнал***, и процесс завершается, получив этот сигнал.

Программируя на языке ассемблера, мы узнали о существовании системного вызова `_exit` (см. т. 2, §3.6.5), который предусматривает один параметр — ***код завершения процесса***. Сам этот код мы упоминали ещё при программировании на Паскале, ведь оператор `halt` позволяет указать значение кода завершения. Изучая язык Си, мы завершали программы, возвращая управление из функции `main` оператором `return`, а для досрочного завершения программы использовали библиотечную функцию `exit` (см. т. 2, §4.2.2). Написав на Си программу без стандартной библиотеки, мы на собственном опыте убедились, что функцию `main` «кто-то» (на самом деле, невидимая подпрограмма `_start`) вызывает в составе выражения `exit(main());`, то есть возврат значения из «главного» экземпляра `main` и прямой вызов `exit` есть в сущности одно и то же. Как несложно догадаться, паскалевский `halt` делает то же самое; все эти способы завершения в конечном счёте приводят к системному вызову `_exit`[28], который, в принципе, из программы на Си можно вызвать и напрямую. Профиль его обёртки таков:

```
void _exit(int code);
```

Параметр `code` задает всё тот же *код завершения процесса*. Напомним, что значение 0 означает успешное завершение, значения 1, 2, 3 и т. д. — что произошла та или иная ошибка или неудача. Обычно используются значения, не превышающие 10, хотя это не обязательно; следует только учитывать, что сам по себе код завершения процесса — это беззнаковое восьмибитное целое, т. е. в терминах Си — значение типа `unsigned char`; поэтому не следует пытаться использовать отрицательные значения или значения, превосходящие 255, это всё равно не получится. Прямое обращение к системе с требованием завершить текущий процесс, как можно догадаться, немедленно завершает процесс, никаких иных вариантов тут не предусмотрено. В этом плане вызов библиотечной

[28] Или его аналогу; современные реализации стандартной библиотеки Си в ОС Linux обычно используют вызов `exit_group`.

функции `exit`, а равно и возврат значения из функции `main` (который тоже приводит к вызову именно библиотечной функции `exit`, а не к прямому обращению к ОС) существенно отличаются: конечно, функция `exit` тоже рано или поздно обратится к операционной системе, причём с помощью того же `_exit`, но перед этим она может «привести дела в порядок»: в частности, при использовании библиотечного («высокоуровневого») интерфейса ввода-вывода, т. е. функций, вводимых заголовочником `stdio.h`, функция `exit`, прежде чем завершить программу, вытеснит всю информацию из буферов вывода[29], тогда как `_exit`, будучи системным вызовом, этого не сделает. Эту разницу легко обнаружить на примере следующей программы (обратите внимание, что перевод строки в `printf` отсутствует, это сделано специально, иначе эффекта не возникнет):

```
int main()
{
    printf("Hello, world");
    _exit(0);
}
```

Если эту программу запустить, она *ничего не напечатает*. Дело в том, что `printf` поместит строку в промежуточный буфер вывода и на этом успокоится; поскольку в строке нет символа перевода строки, вытеснения буфера не произойдёт. Далее вызов `_exit` завершит программу немедленно, не оставив библиотеке шанса вытеснить буфер. Если заменить `_exit` на обычный `exit` (то есть вызвать библиотечную функцию вместо прямого обращения к системному вызову), фраза `Hello, world` будет благополучно напечатана, и то же самое произойдёт, если завершить `main` традиционным «`return 0;`».

Чуть более интересные результаты получатся, если добавить к печатаемому сообщению символ перевода строки, как мы это делали раньше, а для завершения использовать `_exit`. Если в таком виде программу запустить без перенаправления вывода, фраза окажется напечатана, а вот если вывод перенаправить (в файл, в канал, в сокет, куда угодно, лишь бы не на терминал) — напечатано ничего не будет. Причина в том, что вытеснение буфера по символу перевода строки производится только при выводе на терминал.

Так или иначе, и `_exit`, и `exit`, и возврат управления из `main` — это случаи, когда процесс завершается самостоятельно, указав значение кода завершения. Как было сказано выше, возможен другой случай — когда процессу приходит ***сигнал***. Подробное рассмотрение сигналов оставим для отдельной главы; сейчас отметим лишь два момента.

Во-первых, о *коде завершения* при завершении по сигналу речи идти не может, поскольку процесс не дошёл до своего вызова `_exit` и, как следствие, никакого кода завершения не указал; с другой стороны, имеется информация о *номере сигнала*, который стал причиной досрочного

[29] Буферизованный ввод-вывод мы подробно обсуждали во втором томе, см. §4.6.5.

завершения процесса. Обычно номер сигнала — это целое число от `1` до `31`.

Во-вторых, все варианты аварийного завершения программы вследствие её собственных некорректных действий, таких как нарушение защиты памяти, деление на ноль и тому подобное, представляют собой частный случай завершения по сигналу. Когда процесс, работающий под управлением ОС Unix, становится причиной возникновения внутреннего прерывания (исключения) и ядро видит, что к этому привели некорректные действия самого процесса, оно отправляет процессу один из сигналов, предусмотренных специально для этих случаев. О том, что́ же произошло, мы можем узнать по номеру сигнала.

Отметим ещё один важный момент. Как мы видим, при завершении процесса возникает некоторое количество информации об обстоятельствах его завершения: самостоятельно он завершился или был убит сигналом, если самостоятельно — то с каким кодом завершения, если сигналом, то с каким номером. К этому следует добавить ещё информацию из счётчиков потреблённых процессом ресурсов системы, и всё это нужно где-то хранить, а идентификатор (`pid`) процесса нужно защитить от повторного использования до тех пор, пока кто-то не востребует всю эту информацию — ведь она связана с конкретным процессом, а процесс можно надёжно идентифицировать только его `pid`'ом. Если под тем же `pid`'ом в системе появится другой процесс, путаницы не избежать. Поэтому **в ОС Unix процесс при его завершении не исчезает**, а переходит в так называемое ***состояние процесса-зомби***. Тому, как с этими зомби обращаться, мы посвятим следующий параграф.

5.4.6. Ожидание завершения; процессы-зомби

Итак, после завершения процесса в системе остаётся информация о том, при каких обстоятельствах он завершился (сам ли он завершился, если да — то с каким кодом завершения, если нет — то каким сигналом он уничтожен) и значения счётчиков потреблённых ресурсов. Эту информацию должен затребовать родительский процесс; если родительский процесс завершается раньше своего непосредственного потомка, функции родительского берёт на себя процесс `init` (процесс номер 1), при этом даже значение `ppid` (идентификатора родительского процесса) для «осиротевшего» процесса становится равным `1`. Отметим, что больше никто информацию о том, как процесс завершился, получить не может — это прерогатива его непосредственного предка либо «исполняющего обязанности предка» — процесса №1.

Завершённый процесс продолжает существовать в системе в виде процесса-зомби, то есть занимает место в таблице процессов до тех пор, пока находящаяся в нём информация об обстоятельствах завершения не будет затребована родительским процессом. Затребовать информа-

цию и убрать зомби-процесс из системы позволяют системные вызовы семейства `wait`. Простейший из них имеет следующий прототип:

```
int wait(int *status);
```

Если у процесса нет ни одного непосредственного потомка, вызов возвращает код ошибки (значение `-1`). Когда порождённые процессы есть, но ни один из них ещё не завершился, то есть нет ни одного зомби, которого можно снять прямо сейчас, вызов **ждёт завершения любого из порождённых процессов**; отсюда название вызова — wait, по-английски *ждать*. Дождавшись появления зомби (либо если зомби среди потомков данного процесса уже присутствовали на момент обращения к вызову), `wait` изымает из процесса-зомби хранящуюся в нём информацию об обстоятельствах завершения процесса, а сам зомби окончательно отправляет в небытие, освобождая слот таблицы процессов. При этом вызов возвращает `pid` завершившегося процесса, то есть того зомби, который только что окончательно исчез из системы. Если параметр представлял собой ненулевой указатель, то в целочисленную переменную, на которую он указывал, записывается информация о коде завершения процесса или о номере сигнала, по которому процесс был снят.

Для анализа информации, занесённой в такую переменную, используются макросы `WIFEXITED`, `WIFSIGNALED` (был ли процесс завершён обычным способом или по сигналу), `WEXITSTATUS` (если завершён обычным образом, то каков код завершения), `WTERMSIG` (если по сигналу, то каков номер сигнала). Например:

```
int status, wr;
wr = wait(&status);
if(wr == -1) {
    printf("There are no child processes at all\n");
} else {
    printf("Process with pid=%d finished.\n", wr);
    if(WIFEXITED(status)) {
        printf("It has exited with code=%d.\n",
            WEXITSTATUS(status));
    } else {
        printf("It was killed by signal %d.\n",
            WTERMSIG(status));
    }
}
```

Более гибкие возможности предоставляет системный вызов `wait4`:

```
int wait4(int pid, int *status, int opt, struct rusage *usage);
```

В качестве первого параметра вызова `wait4` можно указать идентификатор конкретного процесса, либо `-1`, если требуется дождаться любого из порождённых процессов. Учтите, что при наличии нескольких потомков ждать какого-то одного из них рискованно: остальные могут завершиться раньше, перейти в статус зомби, но убирать их будет некому.

Есть также возможность дождаться процесса из определённой группы процессов. Группы процессов мы рассмотрим позднее, но на всякий случай отметим, что нулевое значение параметра `pid` соответствует ожиданию любого из непосредственных потомков, оставшихся в одной группе с родительским процессом, тогда как отрицательное значение, ме́ньшее `-1`, означает ожидание завершения процесса из группы с заданным номером (например, `-2735` означает группу 2735). Естественно, речь по-прежнему идёт только о непосредственных потомках, всех остальных «дождаться» нельзя.

Параметр `status` используется так же, как для предыдущего вызова: через него передаётся адрес переменной типа `int`, в которую вызов записывает основную информацию об обстоятельствах завершения процесса. В качестве значения параметра `opt` можно указать число `0` или константу `WNOHANG`; в этом случае вызов не ждёт завершения процессов: если ни одного подходящего зомби нет, вызов немедленно возвращает значение `0`. Что касается параметра `usage`, то если он ненулевой, в указанную этим параметром область памяти записываются значения счётчиков ресурсов завершившегося процесса. Как выглядит структура `struct rusage`, можно узнать из документации.

Вызов `wait4` возвращает `-1` в случае ошибки, `0` в случае, если использовалась опция `WNOHANG` и завершившихся процессов не было, и `pid` завершившегося процесса, если вызов успешно получил информацию из зомби.

В ОС Unix предусмотрены также системные вызовы `waitpid` и `wait3`, оба от трёх параметров:

```
int wait3(int *status, int opt, struct rusage *usage);
int waitpid(int pid, int *status, int opt);
```

Первый эквивалентен `wait4` с `-1` в качестве параметра `pid`, второй — `wait4` с `NULL` в качестве параметра `usage`. На вопрос «зачем их столько», к сожалению, невозможно ответить иначе как сакраментальной фразой об «исторических причинах».

Отметим ещё один момент: **зомби нельзя убить, поскольку он уже мёртвый**. Если в вашей системе наблюдаются процессы-зомби, ни команда `kill`, ни какие-то иные ухищрения вам не помогут, убрать зомби из системы может лишь тот процесс, который его породил. Впрочем, вы можете попробовать «пристрелить» нерадивого «родителя», забывшего про свои обязанности, и тогда зомби, скорее всего, исчезнет — его уберёт из системы процесс № 1.

5.4.7. Пример запуска внешней программы

Приведём пример запуска внешней программы с помощью связки `fork+exec`. Наша программа сначала запустит на выполнение (в качестве внешней программы) команду `ls` с параметрами «`-l -a /var`», а затем, дождавшись её завершения, выдаст сообщение «`Ok`».

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
    int pid;
    pid = fork();
    if(pid == -1) {             /* ошибка порождения процесса */
        perror("fork");
        exit(1);
    }
    if(pid == 0) {                      /* порождённый процесс */
        execlp("ls", "ls", "-l", "-a", "/var", NULL);
        perror("ls");     /* exec вернул управление -> ошибка */
        exit(1);        /* завершаем процесс с кодом неуспеха */
    }
    /* родительский процесс */
    wait(NULL); /* дожидаемся завершения порождённого процесса,
                                        заодно убираем зомби */
    printf("Ok\n");
    return 0;
}
```

5.4.8. Выполнение процессов и время

Очевидно, что ни одна программа не может быть выполнена мгновенно; любое выполнение происходит *с течением времени*. Ядро операционной системы хранит текущее время и поддерживает ***системные часы***; это не только позволяет пользователю узнать, который час, но и при необходимости привязать выполнение программ к определённым моментам времени или временны́м промежуткам.

Узнать текущее системное время можно с помощью системного вызова `time`:

```
time_t time(time_t *t);
```

Этот вызов возвращает текущее время в виде *количества секунд, прошедших с начала 1 января 1970 года, причём по Гринвичу*. По-английски такой способ измерения времени называется «*since Epoch*». Третий том, который вы держите в руках, был подписан к печати в день, когда число

секунд since Epoch перевалило за полтора миллиарда[30]. То же число вызов записывает в переменную типа `time_t`, адрес которой передан ему параметром, если только параметр не равен `NULL`; в большинстве случаев его оставляют нулевым и используют значение, возвращаемое `time` как функцией.

Тип `time_t` — это в большинстве случаев обыкновенный `long`, но если вы собираетесь описать переменную, чтобы её адрес передать параметром вызову `time`, лучше описать её как имеющую тип `time_t`.

Конечно, работать с числом секунд, прошедших с далёкого 1970 года, не слишком удобно, но, к счастью, в библиотеке содержатся функции, несколько облегчающие нам жизнь. Функции `gmtime` и `localtime`, а также их «осовремененные» версии `gmtime_r` и `localtime_r`:

```
struct tm *gmtime(const time_t *timep);
struct tm *gmtime_r(const time_t *timep, struct tm *result);
struct tm *localtime(const time_t *timep);
struct tm *localtime_r(const time_t *timep, struct tm *result);
```

позволяют, имея время в виде секунд *since Epoch*, разложить его на составляющие — год, месяц, число, часы, минуты и секунды. Результат представляется в виде структуры `struct tm`, имеющей поля `tm_year` (номер года за вычетом 1900 — например, 2016 год представляется числом `116`), `tm_mon` (месяц, от 0 до 11), `tm_mday` (число, от 1 до 31), `tm_hour` (от 0 до 23), `tm_min` и `tm_sec` (от 0 до 59); кроме того, структура содержит поля `tm_wday` (день недели, от 0 до 6, причём 0 соответствует воскресенью), `tm_yday` (день в году, от 0 до 365) и `tm_isdst`, показывающее, действует ли *daylight saving time* (в России это явление было известно как «летнее время»). Функции `gmtime`/`gmtime_r` возвращают результат для Гринвича, `localtime`/`localtime_r` — для часового пояса, установленного в настройках системы.

Кроме того, в библиотеке присутствуют функции, позволяющие создать текстовое представление заданной даты и времени:

```
char *asctime(const struct tm *tm);
char *asctime_r(const struct tm *tm, char *buf);
char *ctime(const time_t *timep);
char *ctime_r(const time_t *timep, char *buf);
```

Как можно заметить, исходными данными для `ctime` и `ctime_r` служит уже знакомое нам число секунд since Epoch в виде переменной типа `time_t`, а для `asctime` и `asctime_r` — структура `tm`, которую мы могли бы получить с помощью рассмотренных выше `gmtime` и `localtime`, а

[30] Полтора миллиарда секунд с 1 января 1970 года исполнилось 14 июля 2017 года в 5:40 по московскому времени. Два миллиарда исполнится 18 мая 2033 года. За первый миллиард это число перевалило в 2001 году.

при необходимости — скомпоновать любым другим способом. Результат выполнения функции — строка вроде `"Sat Oct 22 00:57:53 2016\n"`.

Отметим, что функции с суффиксом `_r` используют для сохранения результата области памяти, предоставленные вызывающим, тогда как функции без этого суффикса возвращают указатель на принадлежащую им структуру данных (обычно описанную внутри в виде статической локальной переменной). Это требует определённого внимания, поскольку следующий вызов такой функции затирает результат, созданный предыдущим вызовом.

Если вам потребуется текстовое представление даты и/или времени в форме, отличной от предлагаемой функциями `gmtime`/`localtime`, можно воспользоваться также функцией `strftime`:

```
int strftime(char *s, int max, const char *format,
             const struct tm *tm);
```

Полное описание этой функции довольно громоздко, так что мы оставим её читателю для самостоятельного изучения.

Время с точностью до секунды нас во многих случаях не устраивает; в программах, особенно имеющих дело с коммуникацией через компьютерную сеть или по другим каналам связи, зачастую требуется отследить, истёк или не истёк интервал времени, составляющий несколько миллисекунд. Узнать текущее время с бóльшей точностью позволяет системный вызов `gettimeofday`, имеющий довольно странный профиль:

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Структура `timeval` состоит из двух полей: `tv_sec` для целого числа секунд (всё тех же «since Epoch») и `tv_usec` для *микросекунд*, т. е. миллионных долей секунды. Чтобы узнать текущее время, мы должны в программе описать переменную типа `struct timeval` и передать её адрес первым аргументом в вызов `gettimeofday`; про второй аргумент в документации прямо сказано, что его использование нежелательно и будет лучше, если там указать `NULL`.

В ОС Unix, как и в других многозадачных операционных системах, предусмотрены возможности *повлиять на ход выполнения процесса*; в частности, достаточно широк ассортимент способов, которыми процесс может повлиять на ход *своего собственного* выполнения. Простейший способ такого влияния — заявить операционной системе, что в течение некоторого периода времени наш процесс ничего делать не собирается и не нуждается в выделении квантов времени; иначе говоря, процесс хочет приостановить своё собственное выполнение. Процесс в таком состоянии называют ***спящим*** (англ. *sleeping*). Самая простая функция, «отправляющая спать» вызывающий процесс, так и называется `sleep`:

```
int sleep(unsigned int seconds);
```

Единственным параметром этой функции указывается число секунд, которые процесс намерен «проспать»; например, `sleep(60)` «усыпит» наш процесс на минуту, то есть в течение минуты процесс не будет ничего делать — операционная система переведёт его в состояние блокировки, а когда указанный интервал времени закончится — разблокирует.

Когда-то давно функция `sleep` представляла собой отдельный системный вызов. Сейчас это не так: система поддерживает несколько системных вызовов, через каждый из которых может быть реализована функция `sleep`, так что в специальном системном вызове нет нужды.

Отдельного комментария заслуживает возвращаемое значение функции `sleep`. Если процесс благополучно «проспал» указанное время, функция вернёт 0; но, как мы увидим позже, в некоторых случаях (именно — при доставке процессу обрабатываемого ***сигнала***, см. §5.5.2) функция `sleep` может завершиться досрочно. При этом она вернёт положительное целое число, равное количеству секунд (полных или неполных), которые ей оставалось проспать, когда её прервали.

Поскольку секунда — это по компьютерным меркам целая вечность, часто задание времени «сна» в секундах оказывается чрезмерно грубым. Функция `usleep` принимает в качестве параметра число микросекунд:

```
int usleep(long usec);
```

К сожалению, у этой функции имеется довольно серьёзный недостаток: на большинстве Unix-систем она отказывается принимать значения параметра от миллиона и выше, то есть можно попросить «усыпить» наш процесс только на интервал времени, ме́ньший одной секунды. Конечно, никто не мешает скомбинировать `sleep` и `usleep` — например, если нам нужно заснуть на 3,5 с, мы можем сделать

```
sleep(3);
usleep(500000);
```

Но так лучше не делать. Обе функции не гарантируют точности измерения временно́го промежутка хотя бы в силу того, что на передачу управления в ядро с целью блокировки процесса, а затем на возврат управления процессу тоже тратится время, а поскольку в системе могут работать и другие процессы, после выхода из блокировки процесс какое-то время может простоять в очереди, ожидая выделения очередного кванта времени. Естественно, использование двух вызовов функций вместо одного ещё сильнее ухудшит точность.

Нельзя не отметить и ещё один момент: функция `usleep` может вернуть либо 0, если она благополучно «проспала» сколько её просили, либо -1, если «проспать» нужное количество времени ей не дали. В этом случае узнать, сколько времени в действительности прошло, можно только одним способом: перед

«отходом ко сну» спросить у системы текущее время, а после «пробуждения» снова узнать текущее время и вычислить их разность.

Наиболее универсальной функцией для «сна» можно считать `nanosleep`, причём это обычно системный вызов; в частности, современные версии Linux именно через `nanosleep` реализуют библиотечные функции `sleep` и `usleep`. Профиль `nanosleep` таков:

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

Структура `timespec` состоит из двух полей: `tv_sec` для для секунд и `tv_nsec` для *наносекунд*, т. е. миллиардных долей секунды; значение второго поля должно быть заключено между `0` и $10^9 - 1$. Через первый параметр мы указываем нужный нам отрезок времени; вторым параметром мы подаём адрес структуры того же типа, в который функция `nanosleep` запишет, сколько времени осталось до изначально заданного момента пробуждения, если ей не дали «проспать» сколько было указано.

Использование наносекунд может создать впечатление, будто время в системе измеряется с точностью до миллиардных долей секунды. На самом деле это не так, интерфейс системного вызова `nanosleep` спроектирован с большим запасом. В действительности время в системе измеряется хорошо если с точностью до тысячной доли секунды; ядро в качестве основного источника данных о времени использует прерывания таймера, а их в современных системах как раз происходит 1000 в секунду.

Ещё один важный способ влияния на выполнение процесса во времени — это изменение его ***приоритета***. В §5.4.2 мы уже упоминали две составляющие приоритета — статическую и динамическую; повлиять мы можем лишь на статическую составляющую, поскольку динамическую планировщик вычисляет сам, исходя из фактического времени выполнения и ожидания для каждого процесса. Ядро Linux использует в качестве приоритета число от `-20` (наивысший приоритет) до `19` (самый низкий приоритет). В некоторых других unix-системах этот интервал составляет `-20..20`; само число 20 здесь представляет собой некую исторически сложившуюся реальность и никакими серьёзными причинами не обусловлено. По умолчанию, т. е. до тех пор, пока ядро не получит на этот счёт явных указаний, все процессы в системе выполняются с приоритетом `0`.

Процесс может *понизить* свой собственный приоритет с помощью системного вызова `nice`:

```
int nice(int inc);
```

Параметром передаётся небольшое целое число, которое *прибавляется* к текущему значению статического приоритета, тем самым *понижая* приоритет процесса. Опустить свой приоритет ниже минимального, конечно, не получится, так что если указать параметром `nice`, например, число `1000`, ваш процесс получит минимальный возможный приоритет — `19` (в некоторых системах — `20`). Вызов `nice` возвращает новое значение

приоритета, так что, например, процесс может узнать свой текущий приоритет, выполнив `nice(0)`.

Процессы, имеющие полномочия системного администратора, могут указывать *отрицательное* значение параметра при вызове `nice`, тем самым увеличивая свой приоритет в системе. Для обычных процессов это не проходит, вызов заканчивается ошибкой.

Интересно, кстати, что в случае ошибки `nice` возвращает, как и другие системные вызовы, значение `-1`, но то же самое значение вызов может вернуть, если новый приоритет стал равен `-1` — например, если процесс, выполнявшийся в системе с наивысшим приоритетом `-20`, обратился к ядру с вызовом `nice(19)`. В документации говорится, что единственный надёжный способ отличить эту ситуацию от ошибочной — заранее присвоить значение `0` переменной `errno`, а после вызова проверить, не поменялось ли оно. Впрочем, единственная ошибка, возвращаемая вызовом `nice` — это `EPERM`, и происходит она лишь в одном случае: если указано отрицательное значение параметра, а процесс не имеет полномочий суперпользователя. Очевидно, что этой ситуации легко избежать.

Отметим, что **значение приоритета наследуется при создании нового процесса вызовом `fork` и не меняется при вызове `execve`**. Это позволяет, в частности, запустить внешнюю программу как с пониженным, так и (при наличии полномочий) с повышенным приоритетом. Более того, теоретически возможно запустить с заданным приоритетом целый сеанс работы, если это сделает программа, запрашивающая пароль пользователя при входе в систему, ведь все процессы сеанса являются её потомками. Сама эта программа обычно обладает полномочиями суперпользователя, так что она могла бы установить как пониженный приоритет, так и повышенный.

Впрочем, для управления *чьим-то* (не своим) приоритетом система предлагает более удобный механизм — вызовы `getpriority` и `setpriority`, которые позволяют манипулировать приоритетом конкретного процесса, группы процессов[31], либо всех процессов, принадлежащих заданному пользователю:

```
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

В обоих вызовах параметр `which` задаёт способ, которым идентифицируются нужные процессы: `PROP_PROCESS` означает, что будет задан номер (`pid`) конкретного процесса, `PRIO_USER` предполагает, что речь пойдёт обо всех процессах, принадлежащих заданному пользователю, а `PRIO_PGRP` позволяет задать группу процессов. Значение `who` зависит от того, какой из вариантов мы избрали в первом параметре, и равняется соответственно `pid`'у нужного процесса, `uid`'у пользователя или идентификатору (`pgid`'у) группы процессов. Вызов `getpriority` возвращает текущее значение приоритета заданных процессов; если среди

[31] Группы процессов мы уже упоминали в §5.4.2, а подробному рассмотрению этого явления будет посвящён отдельный параграф в главе о работе с терминалом.

процессов, идентифицируемых параметрами вызова, есть процессы с различными приоритетами, возвращается значение самого высокого из этих приоритетов (то есть наименьшее численное значение приоритета). Вызов `setpriority` устанавливает для всех заданных процессов значение приоритета, заданное третьим параметром. Как уже говорилось, численное значение приоритета может составлять от `-20` до `19` (в некоторых системах — до `20`); понижать это значение может только системный администратор; кроме того, обычный пользователь, разумеется, может изменять приоритет только для своих процессов.

Как и для вызова `nice`, для `getpriority` возвращаемое значение `-1` может означать и приоритет, равный `-1`, и ошибку. Как с этим бороться, вы уже знаете.

5.4.9. Перенаправление потоков ввода-вывода

Новые файловые дескрипторы создаются при успешном выполнении вызова `open`, а также во многих других случаях; как мы увидим позже, дескрипторы используются также для каналов, сокетов и т. п., и вообще часто бывают связаны с объектами ядра, не имеющими отношения к файловой системе. При создании нового файлового дескриптора система всегда выбирает наименьший свободный номер; так, если закрыть нулевой дескриптор, следующий успешный вызов `open` вернёт ноль. Для закрытия дескриптора используется уже рассматривавшийся вызов `close`, и это единственный способ избавиться от дескриптора вне зависимости от его природы.

С точки зрения ядра, все дескрипторы обрабатываются одинаково, независимо от их номеров, но, как мы знаем, программы, работающие под управлением системы, обычно считают поток с номером `0` стандартным потоком ввода, поток номер `1` — стандартным потоком вывода, а поток номер `2` — стандартным потоком для сообщений об ошибках. Если говорить точнее, **библиотечные функции считают потоки `0`, `1` и `2` стандартными, что́ бы ни было под этими номерами открыто**. Следовательно, если мы тем или иным способом добьёмся открытия под этими номерами какого-нибудь файла, именно этот файл станет для нашего процесса соответствующим стандартным потоком. Например, если последовательно выполнить

```
close(1);
fd = open("file.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666);
```

и файл `file.txt` успешно откроется на запись, то он откроется под номером `1` (именно единица будет занесена в переменную `fd`) и станет стандартным потоком вывода, то есть с этого момента функции вывода, не предполагающие указания потока, такие как `printf`, `putchar` и т. д., будут выводить информацию в файл `file.txt`. Собственно говоря, все эти функции даже не узнают, что у них сменился стандартный поток.

Недостаток такого способа изменения стандартного потока очевиден: если файл по тем или иным причинам не откроется и `open` вернёт значение `-1`, указывающее на происшедшую ошибку, то закрытый нами стандартный дескриптор так и останется закрытым, и после этого очень легко ошибиться и случайно открыть под тем же номером что-то такое, что совершенно не предназначалось для работы в качестве стандартного потока. Поэтому обычно для манипуляций с таблицей файловых дескрипторов используют специально предназначенные для этого системные вызовы `dup` и `dup2`:

```
int dup(int fd);
int dup2(int fd, int new_fd);
```

Вызов `dup` создаёт новый файловый дескриптор, связанный *с тем же самым* потоком ввода-вывода, что и `fd`. Следует понимать, что нового потока ввода-вывода при этом не создаётся, оба дескриптора оказываются связаны с одним и тем же объектом ядра, реализующим поток ввода-вывода. Новый и старый дескрипторы, в частности, используют один и тот же указатель текущей позиции в файле: если на одном из них сменить позицию с помощью `lseek`, позиция на втором из них тоже изменится. С подобной ситуацией мы уже знакомы: при создании нового процесса вызовом `fork` все файловые дескрипторы родительского процесса копируются, но потоки ввода-вывода остаются прежними, так что порождённый процесс своими дескрипторами ссылается на те же самые потоки.

Вызов `dup2` отличается тем, что новый дескриптор создаётся под заданным номером (параметр `new_fd`). Если на момент выполнения вызова у процесса был поток ввода-вывода, связанный с дескриптором `new_fd`, то есть этот дескриптор был открыт, он закрывается.

Рассмотрим для примера ситуацию, когда некоторая библиотека, которую мы используем, производит вывод нужной нам информации всегда в стандартный поток вывода, а нам желательно соответствующую информацию вывести в файл. Это можно сделать с помощью такого фрагмента кода:

```
int save1, fd;
fflush(stdout);     /* на всякий случай очищаем буфер
                       стандартного вывода */
save1 = dup(1);     /* сохраняем наш стандартный вывод */
int fd = open("file.dat", O_CREAT|O_WRONLY|O_TRUNC, 0666);
                    /* открыли файл */
if(fd == -1) { /* ... обработка ошибки ... */ }
dup2(fd, 1);        /* сделали открытый файл
                       стандартным потоком вывода */
close(fd);          /* закрыли "лишний" дескриптор */
```

```
/* ... производим действия с нашей библиотекой ...
   всё это время вызовы функций, работающих со стандартным
   выводом (таких как printf, puts и т.п.), будут выводить
   информацию в наш файл
*/

dup2(save1, 1);     /* восстановили старый стандартный
                       поток вывода */
                    /* файл при этом закрылся автоматически */
close(save1);       /* лишняя копия нам не нужна */
```

Обычно (хотя и не всегда) стандартные потоки ввода-вывода перенаправляют при запуске внешней программы; например, перенаправления выполняет командный интерпретатор, когда пользователь даёт команду, содержащую соответствующие символы (<, > или >>). При этом используется свойство вызова `execve` сохранять таблицу файловых дескрипторов неизменной. Чаще всего перенаправления производятся с помощью `dup2` в порождённом процессе непосредственно перед вызовом одной из функций семейства `exec`, после чего ненужный уже исходный дескриптор закрывается.

Рассмотрим пример. Допустим, у нас возникла потребность в программе на Си смоделировать функционирование команды

```
ls -l -a -R / > flist
```

т. е., используя возможности программы `ls`, сгенерировать файл `flist`, содержащий список всех файлов в системе с расширенной информацией по каждому из них. Это можно сделать с помощью следующего фрагмента:

```
int pid, status;
pid = fork();
if(pid == 0) {       /* порождённый процесс */
    int fd = open("flist", O_CREAT|O_WRONLY|O_TRUNC, 0666);
    if(fd == -1) {
        perror("flist");
        exit(1);
    }
    dup2(fd, 1);
    close(fd);
    execlp("ls", "ls", "-l", "-a", "-R", "/", NULL);
    perror("ls");
    exit(1);
}
/* родительский процесс */
wait(&status);
if(!WIFEXITED(status) || WEXITSTATUS(status)!=0) {
```

```
    /* ... обработка ошибки ... */
}
```

В этом примере вызов `open` для открытия файла мы тоже выполнили в порождённом процессе, но часто файлы открывают в родительском процессе — до выполнения `fork`; это позволяет в родительском же процессе обработать возможные ошибки, связанные с открытием файла, и, например, вообще не запускать новый процесс, если файл не открылся. Нужно только не забыть закрыть дескриптор также и в родительском процессе:

```
int pid, status, fd;
int fd = open("flist", O_CREAT|O_WRONLY|O_TRUNC, 0666);
if(fd == -1) {
    perror("flist");
    exit(1);
}
pid = fork();
if(pid == -1) { /* ... обработка ошибки ... */ }
if(pid == 0) {       /* порождённый процесс */
    dup2(fd, 1);
    close(fd);
    execlp("ls", "ls", "-l", "-a", "-R", "/", NULL);
    perror("ls");
    exit(1);
}
/* родительский процесс */
close(fd);           /* про это важно не забыть */
wait(&status);
```

Отметим, что сдублировать дескриптор можно также с помощью знакомого нам `fcntl` (см. стр. 57), используя команду `F_DUPFD`; в этом случае вызов принимает третий параметр (почему-то типа `long`), задающий *наименьший желаемый номер дублирующего дескриптора*. Вызов отыскивает первый свободный номер дескриптора, не меньший, чем значение этого параметра, и использует его в качестве номера для нового дескриптора. Так, следующие два вызова эквивалентны:

```
nfd = dup(fd);
nfd = fcntl(fd, F_DUPFD, 0);
```

Вызов `fcntl(fd, F_DUPFD, 20)` создаст дескриптор, дублирующий `fd`, под номером 20, если этот номер свободен, в противном случае найдёт первый свободный номер дескриптора, бóльший, чем 20.

5.4.10. Полномочия процесса

Ранее (см., напр.,§§5.2.2, 5.4.2) мы многократно подчёркивали, что именно процесс — это тот, кто обладает в системе теми или иными

полномочиями, и упомянули такие параметры процесса, как `uid`, `gid`, `euid` и `egid`. Как водится, в реальности всё несколько сложнее. Процессу в системе приписываются:

- «настоящие» (англ. *real*) `uid` и `gid`, как правило, соответствующие идентификаторам пользователя, запустившего процесс;
- «эффективные» или «действующие» (англ. *effective*) `euid` и `egid`, которые как раз используются в большинстве случаев для проверки полномочий процесса;
- «сохранённые» идентификаторы (*saved set-user-ID, saved set-group-ID*);
- массив *дополнительных* идентификаторов групп (*supplementary group IDs*).

Во многих системах этим дело не ограничивается; так, в ОС Linux процесс имеет ещё отдельные «файловые» идентификаторы `fsuid` и `fsgid`, в большинстве случаев равные «эффективным», но допускающие отдельную установку; кроме того, ядро Linux поддерживает механизм *capabilities*, позволяющий разрешать или запрещать процессам определённые привилегированные действия независимо от наличия или отсутствия у них суперпользовательских полномочий. Всё это мы оставим за рамками нашей книги; при желании читатель может обратиться к технической документации.

При старте системы ядро запускает процесс с номером `1` (обычно это программа `/bin/init`); все шесть основных идентификаторов для этого процесса — «настоящие», «эффективные» и «сохранённые» — равны нулю, массив дополнительных групп пуст. Все остальные процессы в системе являются потомками процесса № `1`, хотя, возможно, очень дальними; иное невозможно, поскольку ядро, запустив первый процесс, больше по своей инициативе никаких процессов не создаёт. При порождении нового процесса вызовом `fork` (см. §5.4.3) все идентификаторы наследуются новым процессом без изменений, так что непосредственные потомки `init`'а обычно также имеют `0` в качестве значения всех шести идентификаторов и пустой список дополнительных групп.

Процесс, обладающий полномочиями суперпользователя (т. е. `euid` которого равен нулю), имеет право с помощью системных вызовов сделать со своими полномочиями что угодно, в том числе, что важно, полностью или частично отказаться от своих привилегий. Именно так поступают программы, создающие в системе сеанс работы пользователя: прежде чем запустить командный интерпретатор или оконный менеджер, такие программы устанавливают себе полномочия в соответствии с настройками для данного пользователя. Вернуть себе полномочия суперпользователя после этого уже нельзя, и это правильно: пользователь, работающий в системе, должен обладать полномочиями, которые ему положены в соответствии с конфигурацией системы, и не больше. Чаще всего при таком «сбросе полномочий» все три пары идентификаторов изменяются синхронно, то есть одно и то же число, соответствующее `uid`'у выбранного пользователя, заносится и в «настоящий» `uid`, и в

euid, и в *saved set-user-ID*, а идентификатор основной группы данного пользователя заносится синхронно в «настоящий» gid, в egid и в *saved set-group-ID*. Перед этим (не после, после будет уже поздно) при необходимости формируется список дополнительных групп.

Наиболее очевидный случай, когда uid и gid могут отличаться от euid и egid — это запуск с помощью вызова execve (или любой библиотечной функции из семейства exec, ведь все они в конечном итоге вызывают именно execve) программы, имеющей установленные флаги SetUid и/или SetGid (см. стр. 47). В этом случае «настоящие» идентификаторы остаются те же, что были у запускающего, т. е. у процесса, сделавшего exec, на момент обращения к exec; в то же время, если у исполняемого файла (запускаемой программы) был установлен бит SetUid, то в euid и в *saved set-user-ID* записывается идентификатор владельца исполняемого файла; аналогично при установленном бите SetGid идентификатор группы исполняемого файла заносится в в egid и в *saved set-group-ID*. Массив дополнительных групп остаётся без изменения.

Идея всех этих танцев с тремя парами идентификаторов состоит в том, что процесс может (т. е. имеет право с точки зрения ядра) менять свой euid туда-сюда между «настоящим» и «сохранённым» значениями, то есть (для случая выполняющейся SUID'ной программы) при необходимости часть работы выполнять с правами владельца программы, а часть — с правами пользователя, запустившего программу. Надо сказать, что это ограничение, естественно, перестаёт действовать, если процесс выполняется с правами суперпользователя, т. е. когда его euid равен нулю: в этом случае он имеет право делать со своими идентификаторами всё что угодно.

Пусть, например, пользователь Вася с входным именем vasya uid'ом 1003 создал в системе исполняемый файл vasyaprog и поставил на него атрибут SetUid, а пользователь Петя с входным именем petya и uid'ом 1012 этот файл запустил. В этом случае программа vasyaprog сразу после запуска обнаружит, что её «настоящий» uid равен 1012 (это отражает тот факт, что запустил её Петя), тогда как «эффективный» и «сохранённый» uid'ы оба равны 1003 — это результат действия атрибута SetUid. Если программа не будет предпринимать действий по изменению своего uid'а, то она будет работать в системе от имени Васи и с его полномочиями. В то же время система позволит процессу, в котором работает vasyaprog, изменить свой «эффективный» uid, установив в качестве такового uid Пети (1012): если это проделать, «настоящий» и «эффективный» идентификаторы будут равны 1012 и программа будет продолжать работу уже от имени Пети и с полномочиями Пети, а не Васи. «Сохранённый» идентификатор останется равен 1003, так что программа сможет снова поменять свой «эффективный» uid обратно и продолжить работу от имени Васи, и так далее.

Процесс может узнать свои «настоящие» и «эффективные» идентификаторы с помощью системных вызовов, имеющих очевидные названия:

```
int getuid();
int geteuid();
int getgid();
int getegid();
```

Эти вызовы всегда работают успешно, возвращая численное значение запрошенного идентификатора.

На самом деле тип возвращаемого значения этих системных вызовов — не `int`, а `uid_t` и `gid_t`; в современных версиях того же Linux эти типы определены как `unsigned int`. Использование `int` может создать вам проблемы, если в системе найдётся пользователь или группа, идентификаторы которых превышают $2^{31} - 1$, т. е. 2147483647. Если вы найдёте такую систему, настоятельно рекомендуем вам держаться подальше от её администраторов — они опасны для окружающих.

Стандартного способа узнать «сохранённые» идентификаторы не предусмотрено, но во многих системах, включая Linux, существуют нестандартные вызовы `getresuid` и `getresgid` (буквы `res` обозначают *real, effective, saved*), которые позволяют запросить у ядра сразу все три идентификатора пользователя или группы. Впрочем, если вам это зачем-то понадобилось, скорее всего вы делаете что-то не то.

Узнать свои идентификаторы дополнительных групп процесс может с помощью системного вызова `getgroups`:

```
int getgroups(int size, gid_t *arr);
```

К сожалению, поскольку здесь задействован указатель (и массив), игнорировать разницу между `int` и `gid_t` уже не получится. Вторым параметром в вызов нужно передать указатель на начало массива (из элементов типа `gid_t`), первым — размер этого массива, чтобы вызов «знал», сколько идентификаторов можно записать в указанный вами массив. Первым параметром можно передать ноль, в этом случае второй параметр игнорируется, а вызов возвращает количество дополнительных групп. Если места в массиве не хватает, вызов завершается ошибкой, но этого можно избежать, заранее узнав, какой длины должен быть массив. Вы можете применить примерно такой код:

```
int n;
gid_t *p;
n = getgroups(0, NULL);
p = malloc(n * sizeof(gid_t));
getgroups(n, p);
```

После этого в переменной `n` будет находиться общее количество дополнительных групп, а в массиве по адресу `p` — сами идентификаторы этих групп. Спецификация вызова `getgroups` не указывает, будет ли в число групп включена основная группа, т. е. значение вашего `euid`; это значит, что исчерпывающую информацию о группах пользователей, права которых даны вашему процессу, вы можете узнать только с помощью комбинации вызовов `getgroups` и `geteuid`.

Для манипуляции полномочиями процесса предусмотрены следующие системные вызовы:

```
int setuid(int uid);
int setgid(int gid);
int seteuid(int euid);
int setegid(int egid);
int setgroups(int size, const gid_t *list);
```

С первыми четырьмя из них часто возникает изрядная путаница. Проще всего дело обстоит с вызовами `seteuid` и `setegid`: они в полном соответствии со своими названиями предназначены для изменения «эффективных» идентификаторов пользователя и группы вызвавшего процесса (параметров `euid` и `egid`). Если текущий `euid` процесса отличен от нуля, то процесс может потребовать от системы изменить свой «эффективный» идентификатор (пользователя или группы) на соответствующий «настоящий», «эффективный» или «сохранённый» идентификатор. Иначе говоря, непривилегированный процесс имеет право в качестве параметра вызова `seteuid` указывать любое из трёх значений своего пользовательского идентификатора, в противном случае возникнет ошибка; указывать текущее значение `euid`'а может показаться бессмысленным, но система это ошибкой не считает, просто `euid` останется тем же, что и был. То же касается вызова `setegid` и идентификатора группы: здесь непривилегированный процесс может в качестве параметра указать свой «настоящий», «эффективный» или «сохранённый» идентификатор группы, иное приведёт к ошибке. На процессы с `euid == 0` ограничение, как обычно, не распространяется.

Теперь, собственно, источник путаницы: **вызовы `setuid` и `setgid`, вопреки своим названиям, тоже меняют прежде всего именно «эффективные» идентификаторы пользователя и группы**, а вовсе не «настоящие», как можно было бы ожидать, и меняют их по совершенно тем же правилам, что и `seteuid`/`setegid`: непривилегированный процесс имеет право указать в качестве аргумента свой «настоящий», «эффективный» или «сохранённый» идентификатор пользователя или группы. Единственное отличие этих двух вызовов от двух предыдущих проявляется, когда процесс обладает полномочиями суперпользователя, то есть его `euid` равен нулю. В этом случае вызовы `setuid` и `setgid` изменяют синхронно все три значения соответствующего идентифика-

тора — и «эффективное», и «настоящее», и «сохранённое», тогда как вызовы `seteuid` и `setegid` изменяют только значения «эффективных» идентификаторов.

С вызовом `setgroups` всё, к счастью, довольно просто: второй параметр должен указывать на массив идентификаторов дополнительных групп, первый параметр задаёт длину массива. Вызов, что вполне естественно, доступен только процессам, работающим с полномочиями суперпользователя. Старый список дополнительных групп, каков бы он ни был, бесследно исчезает, вместо него процессу приписывается новый. Следует отметить, что на значения «эффективного», «настоящего» и «сохранённого» идентификатора группы этот вызов никакого влияния не оказывает.

В некоторых случаях более удобными могут оказаться вызовы `setreuid`, `setregid`, `getreuid` и `getregid`, которые мы оставим читателю для самостоятельного изучения.

5.4.11. Количественные ограничения

Помимо ограничений вида «можно-нельзя», которые мы рассматривали в предыдущем параграфе, ядро операционной системы может накладывать на процессы ограничения *количественные*, при которых процесс вправе использовать те или иные ресурсы системы, но только до определённого предела. В §5.2.2 мы приводили примеры таких ограничений: предельное количество виртуальной памяти для отдельно взятого процесса, количество одновременно открытых (одним процессом) файлов, количество самих процессов, запущенных от имени данного пользователя и т. д.

Чтобы получить представление о действующих лимитах, можно воспользоваться командой `ulimit` в командной строке. В частности, `ulimit -a` выдаст список всех возможных в данной системе количественных ограничений и их текущие значения:

```
avst@host:~$ ulimit -a
core file size          (blocks, -c) unlimited
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 20
file size               (blocks, -f) unlimited
pending signals                 (-i) 16382
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
```

```
max user processes              (-u) unlimited
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
avst@host:~$
```

Следует отметить, что все эти ограничения представляют собой *свойства отдельно взятого процесса* — как обычно в таких случаях, наследуемые при создании новых процессов и сохраняющиеся при замене выполняемой программы с помощью `exec`. Иначе говоря, процесс может установить то или иное ограничение, но действовать оно будет только на сам этот процесс и на его потомков, а всю систему в целом не затронет. Важно понимать, что этот принцип распространяется в том числе и на такое (вроде бы) «глобальное» ограничение, как количество процессов, разрешённых для отдельно взятого пользователя (`uid`'а): для процесса, установившего ограничение, и для его потомков ядро при обработке вызова `fork` будет проверять, не превышен ли лимит процессов, учитывая при этом *все* процессы, принадлежащие данному пользователю, и если лимит превышен, `fork` вернёт `-1`, а в `errno` будет занесено значение `EAGAIN`. В то же самое время другие процессы, в том числе принадлежащие тому же пользователю, но не являющиеся потомками процесса, установившего ограничение, будут по-прежнему беспрепятственно порождать новые процессы.

Для каждого вида количественного ограничения система позволяет установить два предельных уровня — мягкий и жёсткий (*soft/hard*). Ядро в своих проверках использует мягкий уровень, но любой процесс может изменить этот уровень (как уменьшить, так и увеличить) для любого из своих ограничений в пределах от нуля до установленного жёсткого уровня. Что касается жёсткого уровня, то любой процесс может его уменьшить, но увеличивать его разрешается только процессам, работающим с правами суперпользователя; для всех прочих процессов уменьшение жёсткого уровня любого количественного лимита необратимо.

Для установки количественных ограничений используется системный вызов `setrlimit`:

```
int setrlimit(int resource, const struct rlimit *rlim);
```

Структура `rlimit` имеет два поля: `rlim_cur` (мягкий уровень) и `rlim_max` (жёсткий уровень); оба поля имеют некий целочисленный тип. Имена полей образованы от слов *current* и *maximum*, т. е. имеется в виду «текущее» и «максимальное» значение ограничения. Для обозначения ситуации отсутствия ограничения используется константа `RLIM_INFINITY`; обычно это число `-1`, преобразованное к типу, используемому для полей `rlim_cur` и `rlim_max`, а поскольку эти поля в большинстве реализаций беззнаковые, `RLIM_INFINITY` оказывается

равно наибольшему возможному беззнаковому целому соответствующей разрядности.

Первый параметр вызова (`resource`) указывает, какое именно ограничение следует установить (изменить). Перечислим некоторые возможные значения этого параметра:

- `RLIMIT_AS` — максимальное количество виртуальной памяти для отдельно взятого процесса (в байтах);
- `RLIMIT_CORE` — максимальный возможный размер core-файла[32], создаваемого при авариях (в байтах); если установить нулевой лимит, core-файлы создаваться не будут;
- `RLIMIT_CPU` — максимальное количество процессорного времени, которое может использовать процесс (в секундах); забегая вперёд отметим, что при достижении мягкого лимита система отправит процессу ***сигнал*** `SIGXCPU`, который по умолчанию убивает процесс, но если процесс перехватит или проигнорирует этот сигнал, то система продолжит посылать процессу сигнал `SIGXCPU` каждую секунду, пока не будет достигнут жёсткий лимит, после чего уничтожит процесс сигналом `SIGKILL`, который нельзя ни перехватить, ни проигнорировать; подробное рассмотрение сигналов ждёт нас в следующей главе;
- `RLIMIT_FSIZE` — максимальный размер создаваемых дисковых файлов (в блоках по 512 байт); если при записи в какой-либо файл этот размер окажется превышен, процесс получит сигнал `SIGXFSZ`, если же этот сигнал не убьёт процесс, то соответствующий системный вызов (например, `write`) вернёт `-1`, а `errno` получит значение `EFBIG`;
- `RLIMIT_NOFILE` — предельное количество одновременно открытых потоков ввода-вывода для одного процесса; если говорить точнее, процесс не может создать (в том числе с помощью `dup2`, см. стр. 96) файловый дескриптор с номером, равным или превосходящим установленный лимит, даже если открытых потоков на этот момент гораздо меньше;
- `RLIMIT_NPROC` — предельное количество процессов, разрешённых для создания от имени конкретного пользователя (т. е. с тем же `uid`'ом); как уже отмечалось, вызов `fork` учитывает при этом все процессы с данным `uid`'ом, имеющиеся в системе, но проверка производится только когда `fork` сделал процесс, для которого установлен соответствующий лимит.

[32]На всякий случай напомним, что core-файл — это файл с именем `core` или `prog.core`, создаваемый операционной системой в текущем каталоге при аварийном завершении процесса. В этот файл полностью записывается содержимое сегментов данных и стека на момент аварии. Core-файлы позволяют с помощью отладчика проанализировать причины аварии, в том числе узнать точку кода, в которой произошла авария, посмотреть значения переменных на момент аварии и т. д.; мы обсуждали это ранее, см. т. 2, §4.17.2.

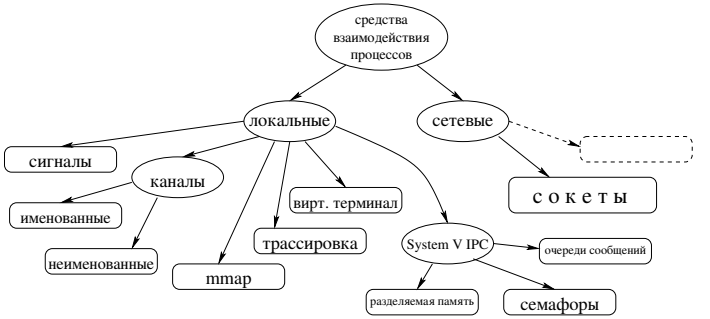

Рис. 5.4. Классификация средств взаимодействия процессов

Существуют и другие лимиты; в наш список мы включили только те, которые проще всего объяснить. Полный список вы найдёте в технической документации, в том числе в тексте справочной страницы по вызову `setrlimit` (команда `man 2 setrlimit`).

Узнать текущие значения ограничений позволяет вызов `getrlimit`:

```
int getrlimit(int resource, struct rlimit *rlim);
```

Помимо количественных ограничений, налагаемых вызовом `setrlimit`, большинство unix-систем поддерживает также ограничения на использование дискового пространства (так называемое *квотирование*); такие ограничения устанавливаются отдельно по каждому диску для заданных пользователей и/или пользовательских групп. В Linux и FreeBSD интерфейсом для этого механизма служит системный вызов `quotactl`. Оставляем его заинтересованным читателям для самостоятельного изучения.

5.5. Базовые средства взаимодействия процессов

5.5.1. Общая классификация

В рамках одной unix-системы процессы могут так или иначе взаимодействовать между собой. Вообще говоря, один процесс может повлиять на работу другого, не прибегая к специализированным средствам; например, процесс может модифицировать файл, читаемый другим процессом, и для этого не потребуется ничего, кроме уже знакомых нам вызовов, позволяющих работать с файлами. Кроме того, системный вызов `mmap`, как уже говорилось, позволяет создать область памяти, доступную нескольким процессам. Такая область памяти называется ***разделяемой***, а процессы, работающие с ней, считаются *взаимодействующими через разделяемую память*, поскольку действия одного из них очевидным образом влияют на работу других.

Тем не менее, для организации взаимодействия процессов удобней пользоваться возможностями системы, которые для этого специально предназначены. Наиболее примитивным из них можно считать уже встречавшиеся нам ***сигналы***. Сигнал не несёт в себе никакой информации, кроме ***номера сигнала*** — целого числа из предопределённого множества.

Для передачи данных между процессами можно использовать однонаправленные ***каналы***, различающиеся на ***именованные*** (FIFO) и ***неименованные*** (pipe). С каналом связываются два файловых дескриптора, один из которых открыт на запись, другой — на чтение, так что данные, записанные в канал одним процессом, могут быть прочитаны другим.

При отладке программ используется режим ***трассировки***, когда один процесс (обычно отладчик) контролирует выполнение другого (отлаживаемой программы).

Как уже говорилось ранее, важную роль в системах семейства Unix играет понятие ***терминала***. При необходимости функциональность терминала как устройства может имитировать пользовательский процесс: так работает, например, программа `xterm`, а также серверы, отвечающие за удалённый доступ к машине, такие как `sshd` или `telnetd`. Взаимодействие процесса, эмулирующего терминал, с процессами, для которых эмулируемый (то есть программно реализованный) терминал является управляющим, называется взаимодействием через ***виртуальный терминал***.

Несколько особое место в классификации занимают средства, объединённые общим названием ***System V IPC***[33]. Эти средства включают механизмы создания разделяемой памяти, массивов семафоров и очередей сообщений. Следует отметить, что в практическом программировании System V IPC используется редко. Эрик Реймонд в книге [4] называет эти средства устаревшими, но это не совсем верно, правильнее было бы назвать их мертворождёнными.

Основным средством *взаимодействия через компьютерную сеть* (то есть взаимодействия процессов, находящихся в разных системах) можно считать ***сокеты*** (*sockets*). Сокеты представляют собой универсальный интерфейс, пригодный для работы с широким спектром протоколов; это означает, что область применения сокетов не ограничена сетями на основе TCP/IP или какого-либо другого стандарта; более того, при добавлении в систему поддержки новых протоколов нет нужды расширять набор системных вызовов. Системы семейства Unix поддерживают также специальный вид сокетов, который можно использовать внутри одной системы, даже если поддержка компьютерных сетей в системе отсутствует.

[33] Символ «V» в данном случае означает римское «пять»; термин читается как «си́стэм файв ай-пи-си».

5.5.2. Сигналы

Один из простейших способов повлиять на работу процесса — это отправить ему ***сигнал*** из некоторого предопределённого множества. Изначально сигналы были предназначены для снятия процессов с выполнения, но с развитием системы Unix приобрели другие функции. Перечислим некоторые наиболее употребительные сигналы.

Сигнал `SIGTERM` предписывает процессу завершиться; процесс может перехватить или игнорировать этот сигнал. `SIGKILL` уничтожает процесс; в отличие от `SIGTERM`, этот сигнал ни перехватить, ни игнорировать нельзя. Не лишним будет запомнить, что сигнал `SIGKILL` имеет номер 9, а `SIGTERM` — 15; некоторые «продвинутые» пользователи Unix могут ожидать от вас такого знания, используя фразы вроде «сначала попробуй пятнадцатым, если через секунду не сдохнет — бей девятым». К счастью, номера всех остальных сигналов не помнит большинство даже самых «продвинутых» любителей Unix.

Два «уничтожающих» сигнала — перехватываемый и неперехватываемый — введены, чтобы можно было снять процесс более гибко. Так, при перезагрузке системы всем процессам рассылается сначала `SIGTERM`, а затем через 5 секунд — `SIGKILL`. Это позволяет процессам «привести дела в порядок»: например, редактор текстов может сохранить редактируемый текст во временном файле, чтобы потом (в начале следующего сеанса редактирования) предложить пользователю восстановить пропавшие изменения.

Сигналы `SIGILL`, `SIGSEGV`, `SIGFPE` и `SIGBUS` система отправляет процессам, чьи действия привели к возникновению исключения (внутреннего прерывания): попытка выполнить несуществующую или недопустимую команду процессора, нарушение защиты памяти, деление на ноль и обращение к памяти по некорректному адресу соответственно. По умолчанию любой из этих сигналов уничтожает процесс с созданием хорошо знакомого нам core-файла[34] для последующего анализа причин происшествия. Однако любой из этих сигналов можно перехватить, например, чтобы попытаться перед завершением записать в файл результаты работы; впрочем, в большинстве случаев из подобной отчаянной попытки всё равно ничего не выходит — например, если наш процесс перехватил сигнал `SIGSEGV`, то структуры данных в его памяти с хорошей вероятностью уже безнадёжно разрушены, и обращение к ним с целью что-то спасти, увы, приведёт лишь к новому сигналу с тем же номером.

Сигналы `SIGSTOP` и `SIGCONT` позволяют соответственно приостановить и продолжить выполнение процесса. Отметим, что `SIGSTOP`, как и `SIGKILL`, нельзя ни перехватить, ни игнорировать. `SIGCONT` перехватить

[34] См. сноску 32 на стр. 105.

можно, но свою основную роль — продолжить выполнение процесса — он в любом случае сыграет.

Сигналы `SIGINT` и `SIGQUIT` отправляются текущей группе процессов данного терминала[35] при нажатии на клавиатуре комбинаций `Ctrl+C` и `Ctrl-\` соответственно. По умолчанию оба сигнала приводят к завершению процесса, причём `SIGQUIT` ещё и создаёт core-файл.

Сигнал `SIGCHLD` система присылает родительскому процессу при завершении его непосредственного потомка; по умолчанию с родительским процессом при получении этого сигнала ничего не происходит.

Сигнал `SIGALRM` присылается по истечении заданного интервала времени после вызова `alarm`. Таким образом процесс может взвести для себя «напоминание», например, на случай чрезмерно долгого выполнения тех или иных действий. Обычно этот сигнал отправляет операционная система. По умолчанию получение `SIGALRM` приводит к завершению процесса, так что следует заранее позаботиться о настройке реакции на этот сигнал.

Сигналы `SIGUSR1` и `SIGUSR2` предназначены для использования программистом для своих целей. По умолчанию эти сигналы также завершают процесс.

Отметим один очень важный момент: все названия сигналов, такие как `SIGKILL` или `SIGCHLD`, представляют собой целочисленные константы, описанные в библиотечных заголовочных файлах и соответствующие номерам сигналов; например, идентификатор `SIGKILL` вводится примерно так:

```
#define SIGKILL 9
```

На всякий случай подчеркнём, что вам в своих программах ничего подобного писать не надо, соответствующие макроопределения уже есть в системных заголовочных файлах.

Отправителем сигнала может быть как процесс, так и операционная система, получатель — всегда процесс. Для отправки сигнала служит системный вызов `kill`:

```
int kill(int target_pid, int sig_no);
```

Параметр `sig_no` задаёт номер сигнала, который следует отправить. Для лучшей ясности программы рекомендуется использовать вместо чисел макроконстанты с префиксом `SIG`, такие как `SIGINT`, `SIGUSR1` и т. п. Параметр `target_pid` задает процесс(ы), которому (которым) следует отправить сигнал. Если в качестве этого параметра передать

[35] Сеансы и группы процессов будут рассмотрены позже. Пока можно считать, что сигналы `SIGINT` и `SIGQUIT` при нажатии соответствующих клавиш получает тот процесс, который вы запустили, набрав команду, а также его потомки (если они не приняли специальных мер).

положительное число, это число будет использоваться как номер процесса, которому следует послать сигнал. Если передать число `-1`, сигнал будет послан всем процессам, кроме самого вызвавшего `kill`, а также процесса № `1` (init). Отрицательное число, бóльшее единицы по модулю, означает передачу сигнала группе процессов с соответствующим номером. Ноль означает передачу сигнала всем процессам своей группы.

Процессы с администраторскими полномочиями (имеющие нулевой `euid`), могут отправлять сигналы любым процессам; все прочие процессы имеют право отправлять сигналы только процессам, принадлежащим тому же пользователю. Как следствие, для непривилегированного процесса вызов `kill(-1, SIGTERM)` означает отправку сигнала `SIGTERM` всем процессам того же пользователя, кроме самого себя.

Если не предпринять специальных мер, большинство сигналов завершают процесс, причём некоторые из них ещё и создают core-файл. Некоторые сигналы (например, `SIGCHLD`) по умолчанию игнорируются. Процесс может для любого сигнала, кроме `SIGKILL` и `SIGSTOP`, установить свой режим обработки: вызов функции-обработчика, игнорирование или обработка по умолчанию. Говорят, что процесс может установить ***диспозицию сигнала***.

Функция-обработчик должна принимать один целочисленный параметр и иметь тип возвращаемого значения `void`, т. е. это должна быть функция вида

```
void handler(int s)
{
    /* ... */
}
```

Для изменения диспозиции сигнала, в том числе для установки обработчика, можно использовать системный вызов `signal`:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signo, sighandler_t hdl);
```

Параметр `signo` задаёт номер сигнала, параметр `hdl` — новую диспозицию для этого сигнала; это может быть адрес функции-обработчика, которая должна быть вызвана при получении соответствующего сигнала; также можно использовать специальные значения диспозиции `SIG_IGN` (игнорировать сигнал) и `SIG_DFL` (вернуть диспозицию по умолчанию). Вызов `signal` возвращает значение, соответствующее предыдущей диспозиции данного сигнала, либо специальное значение `SIG_ERR` в случае ошибки. Отметим, что `SIG_ERR` на самом деле представляет собой число `-1`, преобразованное к типу `sighandler_t`.

Следует подчеркнуть, что **установка диспозиции сигнала влияет на то, что будет, если/когда кто-то пришлёт процессу соответствующий сигнал**; иначе говоря, никаких немедленных послед-

ствий от изменения диспозиции не наступает, и если сигнал с данным номером не придёт процессу никогда, то от изменения его диспозиции вообще не будет видимых эффектов. В частности, установка функции-обработчика не приводит к вызову этой функции. Обработчик будет вызван лишь в том случае, если кто-либо отправит нашему процессу сигнал; вполне возможно, что этого никогда не произойдёт.

Когда процесс получает сигнал, для которого в качестве диспозиции назначена функция-обработчик, эта функция вызывается *асинхронно* по отношению к выполнению остального процесса, то есть, попросту говоря, в самый неожиданный момент. При этом параметр функции будет равен номеру сигнала, что позволяет один и тот же обработчик использовать для нескольких разных сигналов.

Дальнейшее поведение процесса после получения первого экземпляра обрабатываемого сигнала зависит от версии операционной системы (а иногда, как в случае Linux, и от версии системных библиотек). В классических версиях Unix режим обработки сигнала при получении такового (непосредственно перед передачей управления функции-обработчику) сбрасывался в режим по умолчанию. Обычно обработчики первым своим действием снова обращались к вызову `signal`, чтобы переустановить себя в качестве диспозиции, то есть чтобы следующий сигнал с тем же номером тоже привёл к вызову этой функции. Формально говоря, следующий сигнал мог успеть прийти за тот промежуток времени, пока действует диспозиция по умолчанию. К примеру, процесс перехватывает какой-нибудь `SIGUSR1`, чтобы при его получении выполнить какие-то действия, время от времени требующиеся пользователю (скажем, сбросить в файл промежуточные результаты вычислений). Затем первый пришедший сигнал с этим номером приводит к вызову функции-обработчика, обработчик пытается переустановить сигнал, но не успевает, поскольку сразу же за первым `SIGUSR1` приходит второй, и этот второй благополучно убивает процесс, ведь именно такова для этого сигнала диспозиция по умолчанию.

Разработчики ранних версий системы сигналов не беспокоились по этому поводу, поскольку в реальной жизни, если сделать переустановку сигнала самым первым действием функции-обработчика, то времени для этого хватит заведомо. Дело в том, что сигнал всегда обрабатывается в начале очередного кванта времени, и следующий сигнал не может быть обработан (и прийти, собственно, тоже не может) раньше, чем текущий квант времени истечёт и начнётся следующий, ну а на *одно обращение к системному вызову* кванта времени достаточно, более короткие кванты выделять бессмысленно, да и невозможно — не хватит частоты таймера. Однако всё это обусловлено одним конкретным подходом к реализации сигналов в ядре системы; включить соответствующие гарантии в спецификацию интерфейса сигналов означало бы фактически запретить любые другие подходы к реализации, что само по себе плохо. Кроме того, возникли бы неприятные вопросы на тему, *сколько же* времени есть у обработчика сигнала, чтобы успеть себя переустановить. Допустим, переустановка диспозиции первым действием

успевает всегда; а если она будет не первым, а *вторым* действием? А третьим? Конечно, программист, досконально понимающий реализацию сигналов и устройство планировщика в данной конкретной системе, сможет ответить на такие вопросы достаточно достоверно; но ведь суть интерфейсов именно в том, чтобы их пользователь (программист, обращающийся к системным функциям) мог позволить себе не думать, как это всё реализовано «по ту сторону». Кроме того, отличительной особенностью программ на Си считается их *переносимость*, то есть неизменность свойств программы при её компиляции и запуске на совершенно другой системе, и в этом плане намертво завязывать поведение программы на недокументированные свойства одной конкретной реализации ядра тоже не вполне правильно.

К тому, как именно реализована обработка сигналов, мы вернёмся при обсуждении принципов реализации ядра системы.

Ещё один довольно ощутимый недостаток классической семантики обработки сигналов состоит в том, что (и это уже отнюдь не теоретическая возможность) в случае повторного прихода сигнала с тем же номером обработчик, переустанавливающий диспозицию сигнала снова на себя, но при этом (уже после переустановки диспозиции) работающий достаточно долго, мог быть прерван вызовом самого себя. Это не составляет проблемы, если не использовать глобальные переменные, но ведь для общения обработчика со всей остальной программой просто нет никаких средств, кроме глобальных переменных.

Классическую семантику сигналов часто называют «семантикой System V». Как мы уже знаем из вводной части первого тома, в конце 1970-х семейство систем Unix разделилось на две основные ветви — точнее, от «основной» линии развития unix-систем отпочковалась ветвь университета Беркли, известная как BSD. Господствующей версией Unix «основной» линии на тот момент была System V, так что программисты при обозначении наиболее заметных различий между «основной» линией и линией Беркли упоминают с одной стороны BSD (что и понятно), а с другой — обычно как раз System V.

Создатели `BSD` ликвидировали «шероховатость» семантики сигналов, введя сразу два новшества. Во-первых, в BSD-системах диспозиция сигнала могла быть изменена только явным образом, то есть пока процесс сам не обратится к системе с просьбой об изменении диспозиции, никто другой её не изменит. Иначе говоря, при вызове функции-обработчика диспозиция в BSD-системах остаётся прежней, то есть повторный приход сигнала снова вызовет тот же обработчик, переустанавливать его не нужно. Кроме того, на время работы функции-обработчика сигнал с данным номером блокируется, то есть функция не может быть вызвана второй раз из-за пришедшего сигнала, пока первый её вызов не закончился. К сожалению, в качестве интерфейса к системе сигналов в BSD оставили системный вызов с тем же именем и параметрами, что привело к одновременному существованию систем как со старой («одно-

разовой» и без блокировки), так и с новой (постоянной и с блокировкой) семантикой.

Интересно, что ядро Linux обрабатывает системный вызов `signal`, имеющий классическую семантику System V, но большинство версий стандартной библиотеки скрывает этот факт от пользователя, реализуя библиотечную функцию с таким же именем, но имеющую семантику BSD. Обращение к ядру при этом производится через другой системный вызов, `sigaction`.

В современных программах для изменения диспозиции сигналов обычно рекомендуют использовать именно вызов `sigaction`, а не `signal`; отметим, что при переходе от 32-битной архитектуры к 64-битной набор системных вызовов ядра Linux слегка изменился, и в новый набор вызов `signal` не вошёл; впрочем, 64-битные ядра Linux продолжают поддерживать полный набор 32-битных системных вызовов, куда `signal`, естественно, как входил, так и входит; кроме того, функция `signal` (реализованная через `sigaction`) по-прежнему предоставляется стандартной библиотекой, так что изъятие вызова `signal` из интерфейса ядра не должно была привести ни к каким нарушениям совместимости.

Вызов `sigaction` появился позже, имеет стандартную семантику во всех unix-системах и существенно более гибок. В частности, он позволяет задействовать сигналы, несущие на себе дополнительную информацию, а также очереди сигналов (для этого потребуется вызов `sigqueue` вместо привычного `kill`); вся эта функциональность появилась в системах семейства Unix относительно недавно — в начале XXI века. К сожалению, подробное описание вызова `sigaction` оказывается громоздким и перегруженным техническими деталями, поэтому мы оставляем его читателю для самостоятельного изучения.

Чтобы написанная программа вела себя более-менее одинаково при любом из двух вариантов семантики вызова `signal`, следует переустанавливать режим обработки каждый раз в начале функции-обработчика, либо, наоборот, сбрасывать режим обработки в `SIG_DFL`, если требуется перехватить только один сигнал.

Подчеркнём, что функция-обработчик не вызывается из основной программы и не возвращает ей управление; в определённом смысле можно сказать, что её вызывает ядро операционной системы. Так или иначе, не имея ни возможности передать информацию обработчику через параметры, ни получить информацию от неё через возвращаемое значение, основная программа вынуждена взаимодействовать с функцией-обработчиком через глобальные переменные, ведь других способов связи с ней не остаётся.

Сигнал может, как мы уже отмечали, прийти в самый неожиданный момент, когда программа находится в произвольной точке выполнения, и это также порождает определённые сложности. Большинство функций стандартной библиотеки изнутри обработчиков сигналов вызывать нельзя, ведь сигнал может прийти процессу во время выполнения той же самой функции или какой-то другой функции, работающей с теми же данными; функция, выполняемая в основном процессе, может

привести некие «глобальные» данные в нецелостное состояние, которое никогда не наблюдается и не должно наблюдаться извне, возникает лишь на короткий промежуток времени, и которое ни сама эта функция, ни другие функции не расчитывают «увидеть», когда бы их ни вызвали. Поскольку из обработчика никак нельзя определить, в какой момент была прервана основная программа, дальнейшее вмешательство в нецелостные структуры данных приведёт к непредсказуемым последствиям.

Прежде всего это касается динамической памяти, ведь динамическая память как целое — так называемая ***куча*** — это довольно сложная структура данных. Вызывая из обработчика сигнала `malloc` или `free`, мы рискуем попасть ровно на тот момент, когда основная программа тоже находится внутри `malloc`'а или `free`, так что, не зная о присутствии друг друга, экземпляр, вызванный из основной программы, и экземпляр, вызванный из обработчика сигнала, вполне могут разрушить кучу. Наша программа после этого, скорее всего, «свалится», причём не сразу. Такие ошибки проявляются с достаточно низкой вероятностью, ведь для этого нужно, чтобы обработчик сигнала оказался вызван именно тогда, когда основная программа что-то делает с кучей; как мы уже знаем, ошибка, которую тяжело или вообще невозможно воспроизвести — это самый кошмарный вариант с точки зрения трудоёмкости и утомительности отладки.

По тем же причинам нельзя использовать в обработчиках сигналов функции высокоуровневого ввода-вывода. Причина здесь кроется в буферизации потоков; для её организации с каждым потоком ввода-вывода библиотека вынуждена связывать буфер, который тоже представляет собой нетривиальную структуру данных.

С другой стороны, ввод-вывод изнутри обработчиков сигналов можно произвести с помощью системных вызовов, но здесь тоже имеется своя сложность, заключающаяся в наличии глобальной переменной `errno`. Представьте себе, что основная программа выполнила системный вызов, который завершился с ошибкой, но воспользоваться значением `errno` не успела, и тут как раз процессу пришёл сигнал, запустился обработчик, и в нём тоже выполнился какой-то системный вызов, закончившийся ошибкой. Естественно, он запишет результат своего выполнения в `errno`, так что диагностика, которую потом выдаст основная программа, не будет иметь ничего общего с действительностью.

Даже простые обращения к глобальным переменным из обработчика сигнала могут оказаться неприемлемыми. Так, если мы работаем с 32-битным процессором, присваивание значения переменной типа `long long` будет выполняться в две команды, как и извлечение значения из такой переменной. Если представить, что основная программа начала извлекать значение из переменной, но успела выполнить только одну машинную команду из двух, а в это время функция-обработчик

занесла в переменную новое значение, то значение, которое в итоге извлечёт из переменной основная программа, окажется состоящим из обрывков двух значений: четыре байта от старого значения, четыре байта от нового. Правильных результатов после этого уже не получить.

Мало того, в дело могут вмешаться ещё и особенности компилятора Си. Как известно, современные компиляторы предполагают оптимизацию машинного кода; в частности, компилятор может предположить, что, коль скоро в некую переменную никто ничего не записывает «прямо здесь», то её значение измениться не может, и если несколько команд назад это значение было прочитано из памяти в регистр, оптимизатор вполне может выбросить из кода повторное чтение той же памяти, воспользовавшись значением, всё ещё находящимся в регистре. Чтобы избежать подобных эффектов, язык Си предусматривает ключевое слово `volatile` для пометки переменных, значение которых может «неожиданно» измениться.

Современные спецификации системы сигналов предлагают программисту воспользоваться специальным типом `sig_atomic_t` (обычно это синоним простого int, введённый с помощью `typedef`), гарантируя, что обращение к такой переменной проходит *атомарно*, то есть за одно неделимое действие — одну машинную команду. Кроме того, переменную, которую будет менять обработчик сигнала, предлагается обязательно помечать словом `volatile`.

Ещё один важный момент, который необходимо учитывать при обработке сигналов — это поведение блокирующих системных вызовов. Если процесс выполнил системный вызов, который перевёл его в состояние блокировки — например, запросил с помощью `read` чтение данных из потока, в котором сейчас нет данных, или с помощью функции `sleep` сообщил операционной системе, что в ближайшее время ничего делать не собирается, и т. п — то при получении процессом обрабатываемого сигнала такой системный вызов немедленно вернёт управление. В большинстве случаев системный вызов, прерванный сигналом, вернёт значение `-1`, то есть заявит о происшедшей ошибке. Отличить эту ситуацию от настоящих ошибок можно по значению переменной `errno`: она будет равна константе `EINTR`.

Дело в том, что для запуска функции-обработчика процессу нужно быть в состоянии обычного выполнения, а реализованы сигналы так, что после завершения функции-обработчика неизбежно будет продолжено выполнение кода основной программы — с того места, где оно было прервано пришедшим сигналом. Если при этом процесс находился в состоянии «выполнения в ядре», в том числе в состоянии блокировки, то вернуться обратно в ядро он может только по своей инициативе. Ядро не может (во всяком случае, надёжно) отследить момент завершения выполнения обработчика и отнять у процесса инициативу, принудитель-

но вернув его «в ядро»; почему это так, станет ясно при рассмотрении механизма реализации вызова обработчиков сигналов.

С неблокирующими вызовами проблем не возникает: ядро завершает обработку такого вызова, что обычно не занимает много времени, после чего возвращает управление коду процесса, и в этот момент как раз отрабатывает функция-обработчик сигнала. С блокирующими вызовами этот номер не проходит: в блокировке процесс может «провисеть» сколь угодно долго, и если допустить, чтобы всё это время процесс не реагировал на сигналы, это сделало бы всю систему сигналов бессмысленной. Поэтому операционная система предпочитает немедленно прекратить обработку имеющегося системного вызова, чтобы дать возможность процессу получить и обработать сигнал, но о том, что вызов не был завершён, естественно, процессу сообщает — вызов сигнализирует об «ошибке» `EINTR`. Как правило, программы пишут так, чтобы в этом случае снова обратиться к «неожиданно прерванному» системному вызову.

На самом деле, как ни странно, это происходит не всегда. Уже упоминавшийся вызов `sigaction` позволяет при установке диспозиции сигнала указать ряд настроек, в число которых входит флажок `SA_RESTART`; если он указан, то при поступлении обрабатываемого сигнала процессу, который находится в состоянии блокировки *в одном из явно перечисленных в документации блокирующих системных вызовов*, система должна вывести процесс из состояния блокировки, дать ему квант времени для обработки сигнала, а затем — сразу после выхода из обработчика — вернуть процесс в состояние блокировки. Картина осложняется тем, что так происходит далеко не для всех блокирующих системных вызовов; например, `wait` и `write` перезапускаются, тогда как `usleep` и `nanosleep` при поступлении обрабатываемого сигнала всегда возвращают ошибку `EINTR`; с вызовом `read` всё ещё интереснее: в большинстве случаев он перезапускается, *но не всегда*. К тому же процесс может «обмануть» ядро, не дав ему шанса что-то перезапустить.

Некоторые (опять же, не все) версии библиотеки, которые эмулируют вызов `signal` через обращение к `sigaction`, вдобавок зачем-то устанавливают флажок `SA_RESTART`, что окончательно запутывает дело.

Правильнее всего, конечно, было бы отказаться от `signal` в пользу `sigaction`, а `SA_RESTART` либо не использовать вовсе, либо перед его использованием проштудировать списки системных вызовов, умеющих и не умеющих перезапускаться при поступлении сигнала; но всё это влечёт заметные трудозатраты, а сигналов в жизни вашей программы обычно достаточно скромна. Поэтому можно посоветовать другой вариант: если в своей программе вы обрабатываете сигналы, для каждого блокирующего системного вызова будьте готовы к тому, что он завершится с «ошибкой» `EINTR` (которая на самом деле вовсе никакая не ошибка), но при этом не полагайтесь на то, что так обязательно произойдёт.

Напишем для примера программу, которая при нажатии `Ctrl-C` сначала выдаёт сообщение и лишь на 25-й раз завершается. Для начала сделаем это так, как обычно такие программы пишут начинающие:

```
#include <stdio.h>
#include <signal.h>
int n = 0;
void handler(int s)
{
    n++;
    printf("Press it again, I like it\n");
}
int main()
{
    signal(SIGINT, handler);
    while(n<25)
        ;
    return 0;
}
```

Несмотря на то, что эта программа успешно пройдёт компиляцию и даже будет работать в соответствии с условиями задачи, правильной она не является. Попробуем исправить ошибки. Тип глобальной переменной заменим на `sig_atomic_t` и пометим её словом `volatile`, чтобы исключить сюрпризы со стороны оптимизатора. В начало обработчика вставим переустановку диспозиции. Вместо `printf` воспользуемся системным вызовом `write`, для чего, во-первых, опишем сообщение в виде глобальной строковой константы, и, во-вторых, снабдим обработчик действиями по сохранению и восстановлению значения `errno`. Наконец, обратим внимание, что наша программа непроизводительно расходует процессорное время в бесконечном цикле и вставим туда вызов `sleep(1)`, так что программа будет просыпаться один раз в секунду, чтобы снова заснуть; сколько-нибудь заметной нагрузки на процессор она уже не создаст.

```
/* pressagain.c */
#include <signal.h>
#include <unistd.h>
#include <errno.h>
volatile static sig_atomic_t n = 0;
const char message[] = "Press it again, I like it\n";
void handler(int s)
{
    int save_errno = errno;
    signal(SIGINT, handler);
    n++;
    write(1, message, sizeof(message)-1);
    errno = save_errno;
}
int main()
{
    signal(SIGINT, handler);
```

```
        while(n<25)
            sleep(1);
        return 0;
    }
```

Разумеется, мы могли бы использовать и другой аргумент для `sleep`, например, при использовании `sleep(3600)` программа просыпалась бы раз в час, но и один раз в секунду — это уже достаточно редко, чтобы паразитной нагрузки не было заметно. Напомним, что при получении процессом обрабатываемого сигнала `sleep` всегда возвращает управление, так что, сколь бы долгий период сна мы ни задали, сразу после обработки сигнала произойдёт очередная проверка условия `while`.

ОС Unix предусматривает специальный системный вызов `pause` как раз для таких случаев. Этот вызов, не принимающий параметров, блокирует процесс до тех пор, пока не будет получен обрабатываемый сигнал. Интересно, что единственный вариант возвращаемого значения для вызова `pause` — это `-1`, то есть «ошибка», при этом в `errno` заносится значение `EINTR`. Если заменить в нашей программе строку «`sleep(1);`» на «`pause();`», она станет «ещё правильнее».

С помощью вызова `alarm` можно затребовать от ядра отправки вызвавшему процессу сигнала `SIGALRM` через определённое количество секунд времени. Прототип вызова таков:

```
int alarm(unsigned int seconds);
```

Параметр задаёт количество секунд, через которое следует прислать сигнал. Когда заказанный период времени истекает, система присылает процессу сигнал, а сам активный «заказ» сбрасывается. Возвращаемое вызовом `alarm` значение зависит от того, имеется ли уже для данного процесса активный заказ на отправку `SIGALRM`. Если нет, вызов возвращает ноль. Если же активный заказ уже есть, возвращено будет количество секунд (полных или неполных, то есть всегда не менее `1`), оставшееся до момента его исполнения.

Система может помнить только об одном заказанном сигнале `SIGALRM` для каждого процесса; иначе говоря, текущее значение «заказа» представляет собой свойство процесса. Если по результатам предыдущего вызова сигнал процессу ещё не прислали, новый вызов отменит старый заказ и установит новый. Нулевое значение параметра `seconds` отменит активный заказ, не установив новый.

Следует учитывать, что по умолчанию сигнал `SIGALRM` *убивает* процесс, так что в большинстве случаев нужно *сначала* изменить диспозицию `SIGALRM`, установив функцию-обработчик, и лишь после этого обращаться к вызову `alarm`.

В описании функции `sleep` говорится, что она может быть реализована через вызов `alarm` и обработку `SIGALRM`, так что использование этих двух механизмов не следует смешивать в одной программе. В реальности, конечно, ничего

подобного не происходит; например, в большинстве версий Linux функция `sleep` реализуется через обращение к системному вызову `nanosleep`, но с таким же успехом её можно было бы реализовать через системный вызов `select`, который мы будем рассматривать позже; есть и другие способы.

Про `usleep` ничего подобного в спецификации не говорится, а про `nanosleep` в явном виде сказано, что она никак не пересекается с обработкой сигналов. Кроме того, стоит заметить, что `usleep` и `nanosleep`, будучи прерваны пришедшим сигналом, ведут себя как обычные блокирующие системные вызовы: возвращают `-1`, занося в `errno` значение `EINTR`. Функция `sleep` в такой ситуации *возвращает число секунд (полных или неполных), оставшихся до предполагавшегося момента пробуждения*.

В целом можно сделать вывод, что лучше вообще не использовать `sleep`, отдавая предпочтение `usleep` или `nanosleep`; впрочем, это скорее дело вкуса.

Несмотря на кажущуюся простоту, активная работа с сигналами требует высокой квалификации. При использовании сигналов часто возникают ситуации гонок, сами сигналы ненадёжны, при отправке двух одинаковых сигналов прийти может только один и т. д. Если ваша программа обрабатывает сигналы, необходимо помнить об этом при обращении ко всем блокирующим системным вызовам, ведь каждый из них может «ошибиться» просто из-за того, что во время его выполнения процесс получил сигнал. Написание корректной программы, активно использующей сигналы, может оказаться весьма нетривиальным делом.

5.5.3. Каналы

Канал — это объект ядра операционной системы, представляющий собой средство однонаправленной передачи данных от одного процесса другому. Можно представлять себе, что канал имеет два конца, один для записи, другой для чтения. С каждым концом канала могут быть связаны файловые дескрипторы, принадлежащие, возможно, разным процессам. Иначе говоря, канал — это такой объект ядра, с которым связываются поток ввода и поток вывода, причём информация, переданная каким-то процессом в канал через поток вывода, поступает в его поток ввода, откуда может быть считана другим процессом.

В ОС Unix различают *именованные* и *неименованные* каналы. Для работы с именованными каналами в файловой системе создаются файлы специального типа, открытие которых с помощью вызова `open` приводит к *подключению* к каналу; сам канал как объект ядра создаётся в тот момент, когда какой-то из процессов пытается открыть на запись или чтение файл именованного канала, который пока не открыт ни в каком другом процессе.

В противоположность этому, неименованные каналы сразу создаются открытыми с обоих концов; при этом ни с каким файлами они не связаны, так что «подключиться» к уже созданному неименованному каналу невозможно.

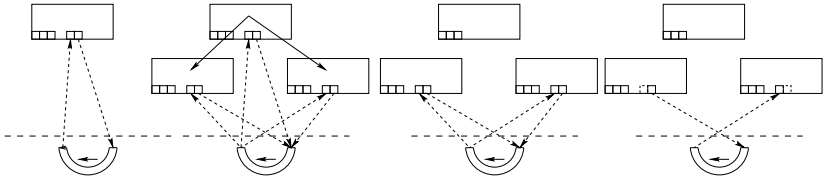

Рис. 5.5. Связывание двух процессов-потомков через неименованный канал

Неименованный канал создаётся системным вызовом `pipe`:

```
int pipe(int fd[2]);
```

На вход ему подаётся адрес массива из двух элементов типа `int`; в этот массив вызов `pipe` записывает дескрипторы, связанные с созданным каналом: `fd[0]` — для чтения, `fd[1]` — для записи.

Поскольку такой канал не имеет ни имени, ни какой-либо иной идентифицирующей информации, подключиться к нему из другого процесса невозможно. Единственный способ добиться того, чтобы к одному и тому же каналу оказались подключены разные процессы — это породить копию процесса, создавшего канал, с помощью вызова `fork`. Иначе говоря, использовать неименованный канал для взаимодействия между собой могут только «родственные» процессы, а точнее — процесс, создавший канал, и его потомки.

На рис. 5.5 показано связывание двух порождённых процессов с помощью неименованного канала. На первом шаге (до порождения новых процессов) родительский процесс создаёт канал. Затем родительский процесс порождает двух потомков, в результате чего во всех трёх процессах оказываются дескрипторы как одного, так и второго концов канала. После этого родительский процесс закрывает свои экземпляры дескрипторов, чтобы не мешать потомкам использовать канал. В свою очередь, в первом потомке закрывается дескриптор, предназначенный для чтения, во втором — дескриптор, предназначенный для записи. В результате первый потомок может передавать второму данные через канал. Соответствующий код на языке Си будет выглядеть приблизительно так:

```
int fd[2];
int p1, p2;
pipe(fd);
p1 = fork();
if(p1 == 0) { /* child #1 */
    close(fd[0]);
    /* ... */
    write(fd[1], /* ... */);
```

```
        /* ... */
        exit(0);
    }
    p2 = fork();
    if(p2 == 0) { /* child #2 */
        close(fd[1]);
        /* ... */
        rc = read(fd[0], /* ... */);
        /* ... */
        exit(0);
    }
    /* parent */
    close(fd[0]);
    close(fd[1]);
    /* ... */
```

Рассмотрим ситуацию, когда в системе присутствуют открытые дескрипторы обоих концов канала. При попытке чтения из канала, в который пока ничего не записано, читающий процесс будет заблокирован (то есть `read` не вернёт управление) до тех пор, пока либо кто-нибудь не запишет данные в канал, либо все дескрипторы, открытые на запись в этот канал, не окажутся закрыты. Отметим, что, если в канале доступны для чтения данные (независимо от их количества, хотя бы один байт), функция `read` при попытке чтения из канала вернёт управление немедленно; если третий параметр `read` (количество запрашиваемых байтов) больше, чем на момент вызова оказалось доступно данных, прочитаны будут все доступные данные и `read` вернёт их количество, которое при этом будет меньше заказанного.

Попытки записи в канал, из которого никто не читает (при условии, что в системе имеются дескрипторы, связанные с противоположным концом канала), некоторое время будут успешными. Дело в том, что канал имеет внутренний буфер, размер которого зависит от реализации; так, в старых версиях ядра Linux он обычно составлял 4096 байт, а в современных его зачем-то расширили до 65536 байт. Когда буфер окажется заполнен, очередной вызов `write` заблокирует процесс до тех пор, пока кто-нибудь не начнёт из канала читать, тем самым освобождая место в буфере.

Рассмотрим теперь случаи, когда все дескрипторы, связанные с одним из концов канала, оказались закрыты. Ясно, что данный конкретный канал более никогда не удастся использовать, поскольку способа вновь связать дескриптор с одним из концов неименованного канала в системе нет. **Если оказались закрыты все дескрипторы, через которые можно было записывать данные в канал, операции чтения (вызовы `read`) сначала опустошат внутренний буфер канала, а затем будут возвращать 0, сигнализируя о наступлении ситуации «конец файла»).**

Если, наоборот, оказались закрыты все дескрипторы, через которые можно было из канала читать, то первая же попытка записи в канал приведёт к тому, что попытавшийся осуществить запись процесс получит сигнал SIGPIPE. По умолчанию этот сигнал завершает процесс. Вызов `write` при этом возвращает `-1`, что может быть обнаружено только если процесс перехватывает или игнорирует сигнал `SIGPIPE`.

Наиболее активно неименованные каналы используются для реализации ***конвейеров*** — групп программ, которые запускаются на одновременное выполнение, причём информация, выдаваемая первой программой на стандартный вывод, поступает второй программе на стандартный ввод, вывод второй программы — на ввод третьей программе и т. д. (см. пример на стр. 21).

Для построения конвейера следует создать нужное число каналов (на один меньше, чем в конвейере будет элементов) и соответствующим образом связать потоки стандартного ввода и вывода (то есть дескрипторы 0 и 1) в процессах, составляющих конвейер, с концами каналов. Очень важно при этом закрыть все лишние дескрипторы, связанные с данным каналом, во всех процессах, вовлечённых в решение задачи. Программы в ОС Unix часто пишутся так, чтобы работать до возникновения ситуации «конец файла» в потоке стандартного ввода; такая ситуация может возникнуть на канале только если *все* дескрипторы записи окажутся закрыты. Наличие лишнего открытого дескриптора записи нарушит нормальную работу конвейера. С другой стороны, после исчезновения процесса, для которого предназначены генерируемые программой данные, продолжать выполнение программы обычно бессмысленно. Если с исчезновением следующего элемента конвейера закроется *последний* дескриптор, открытый на чтение из канала, то пишущий процесс будет снят сигналом `SIGPIPE`. Если же где-то останется ещё хотя бы один открытый дескриптор для чтения, процесс будет просто заблокирован; возможно, это блокирует выполнение ещё каких-то задач, которые дожидаются завершения этого процесса.

Рассмотрим для примера конвейер

```
ls -lR  |  grep '^d'
```

Программа на Си, выполняющая те же действия (то есть запускающая те же программы с теми же параметрами и в таком же режиме взаимодействия), будет выглядеть так:

```
int main()
{
    int fd[2];
    pipe(fd);                          /* создаем канал для связи */
    if(fork()==0) {           /* процесс для выполнения ls -lR */
```

```
        close(fd[0]);            /* читать из канала не нужно */
        dup2(fd[1], 1);             /* станд. вывод - в канал */
        close(fd[1]);                /* fd[1] больше не нужен */
        execlp("ls", "ls", "-lR", NULL);  /* запускаем ls -lR */
        perror("ls");    /* не получилось, сообщаем об ошибке */
        exit(1);
    }
    if(fork()==0) {            /* процесс для выполнения grep */
        close(fd[1]);              /* писать в канал не нужно */
        dup2(fd[0], 0);            /* станд. ввод - из канала */
        close(fd[0]);                /* fd[0] больше не нужен */
        execlp("grep", "grep", "^d", NULL); /* запускаем grep */
        perror("grep");  /* не получилось, сообщаем об ошибке */
        exit(1);
    }
        /* в родительском процессе закрываем оба конца канала */
    close(fd[0]);
    close(fd[1]);
    wait(NULL);             /* дожидаемся завершения потомков */
    wait(NULL);
    return 0;
}
```

Каналы, создаваемые с помощью `pipe`, не позволяют организовать передачу данных между неродственными процессами. Этого недостатка лишены ***именованные*** каналы, которые в англоязычных источниках называются просто FIFO (*first in, first out*, т. е. «очередь»). Англоязычное название оправдывается тем, что байты, переданные в такой канал (как, впрочем, и в любой поток ввода-вывода, имеющий «другой конец») при передаче сохраняют свой порядок, так что байт, переданный раньше других, извлечён (прочитан) из канала будет также раньше других.

Именованные каналы подобны неименованным, с той разницей, что именованному каналу соответствует размещаемый в файловой системе файл специального типа; этот тип так и называется FIFO. К именованному каналу могут присоединяться процессы, не имеющие родственных связей; более того, закрытие всех дескрипторов, отвечающих за чтение из такого канала или за запись в такой канал, ещё не означает, что канал более не пригоден для работы, т. к. в любой момент такие дескрипторы могут появиться вновь.

Для создания файла типа FIFO используется функция `mkfifo`:

```
int mkfifo(const char *name, int permissions);
```

Первый параметр задаёт имя файла, второй — права доступа к нему (аналогично вызовам `open` и `mkdir`). Права, естественно, модифицируются параметром `umask`. Функция возвращает `-1` в случае ошибки, ноль — в случае успеха.

При создании файла FIFO система не создаёт сам объект канала в ядре; это происходит только тогда, когда какой-нибудь процесс открывает файл FIFO с помощью вызова `open` на чтение или запись; от того, в каком режиме (`O_RDONLY` или `O_WRONLY`) процесс попытается открыть файл типа FIFO, зависит, к какому концу канала будет присоединён полученный файловый дескриптор. Открывать файл FIFO в режиме `O_RDWR` не следует, несмотря на то, что в некоторых системах (включая Linux) это вполне сработает, то есть созданный файловый дескриптор будет ассоциирован с обоими концами канала; но так, во-первых, происходит не во всех существующих системах, и, во-вторых, само по себе такое действие выглядит несколько глупо.

Канал как объект ядра продолжает существовать до тех пор, пока существует хотя бы один связанный с ним дескриптор, после чего уничтожается. Уничтожение канала не означает уничтожения файла FIFO: после закрытия всех дескрипторов файл остаётся на месте и может быть снова открыт каким-нибудь процессом, после чего в ядре системы снова появится объект канала.

Прежде чем начать передачу данных, канал надо открыть с обоих концов. Обычно попытка открыть канал с одной из сторон блокируется до тех пор, пока кто-то не откроет второй конец канала; иначе говоря, вызов `open`, применённый к одному концу канала, не возвращает управление до тех пор, пока кто-то не откроет второй конец. Это поведение можно изменить, добавив во втором параметре вызова `open` флаг `O_NONBLOCK`; в этом случае открытие на чтение пройдёт успешно без блокировки вне зависимости от того, открыт ли кем-нибудь противоположный конец канала, но вот попытка неблокирующего открытия на запись может привести к неудаче: если канал никем не открыт на чтение, то вызов `open`, выполненный с флагами `O_WRONLY|O_NONBLOCK`, вернёт `-1`, а в переменную `errno` будет записан код `ENXIO`.

Поведение именованного канала при закрытии последнего из дескрипторов, отвечающих за один из концов, полностью аналогично поведению неименованного канала в таких же случаях, то есть попытка читать из канала, у которого закрылся последний пишущий дескриптор, приводит к ситуации «конец файла», а попытка писать в канал, у которого закрылся последний читающий дескриптор, приводит к получению сигнала `SIGPIPE`. Отличие в том, что здесь оба случая не являются фатальными; так, после получения ситуации «конец файла», вообще говоря, возможно, что один из следующих вызовов `read` прочитает из того же потока какие-то данные. Это произойдёт, если какой-то другой процесс снова откроет тот же канал на запись. При этом всё время, пока ни одного пишущего дескриптора в системе нет, `read` будет продолжать возвращать `0`, сигнализируя о ситуации конца файла. С записью и `SIGPIPE` ситуация аналогична, хотя использовать это свойство на практике трудно.

5.5.4. Краткие сведения о трассировке

Трассировка обычно применяется при отладке программ. В режиме трассировки один процесс (отладчик) контролирует выполнение другого процесса (отлаживаемой программы), может остановить его, просмотреть и изменить содержимое его памяти, выполнить в пошаговом режиме, установить точки останова, продолжить выполнение до точки останова или до системного вызова и т. п.

В ОС Unix трассировка поддерживается системным вызовом `ptrace`:

```
int ptrace(int request, int pid, void *addr, void *data);
```

В качестве параметра `request` вызов получает одну из возможных команд (какие конкретно действия, связанные с трассировкой, требуются). Интерпретация остальных параметров зависит от команды.

Начать трассировку можно двумя способами: запустить трассируемую программу с начала или присоединиться к существующему (работающему) процессу. В первом случае отладчик делает `fork`, порождённый процесс сначала устанавливает режим отладки (вызывает `ptrace` с командой `PTRACE_TRACEME`), затем делает `exec` программы, подлежащей трассировке. Сразу после `exec` система останавливает трассируемый процесс и отправляет родительскому процессу (отладчику) сигнал `SIGCHLD`; собственно говоря, для этого и нужна команда `PTRACE_TRACEME`, она предписывает системе приостановить процесс после `exec`. Отладчик должен дождаться нужного момента с помощью вызовов семейства `wait`, которые в этом случае будут ожидать не окончания порождённого процесса, а его останова для трассировки. Далее отладчик может заставить отлаживаемый процесс выполнить один шаг с помощью команды `PTRACE_SINGLESTEP`, продолжить его выполнение с помощью `PTRACE_CONT`, узнать содержимое регистров с помощью `PTRACE_GETREGS` и т. п.

Для присоединения к существующему процессу используется вызов `ptrace` с командой `PTRACE_ATTACH`. При этом трассирующий во многих смыслах начинает выполнять роль родительского процесса по отношению к отлаживаемому; в частности, сигналы `SIGCHLD` посылаются теперь трассирующему, а не исходному родительскому процессу, хотя функция `getppid` в отлаживаемом процессе продолжает возвращать идентификатор настоящего родительского процесса.

5.6. Терминал и сеанс работы

5.6.1. Драйвер терминала и дисциплина линии

Понятие ***терминала*** в unix-системах было и остаётся одним из центральных, в особенности когда речь идёт об управлении процесса-

Рис. 5.6. Сеанс работы с текстом книги на терминале vt420[36]

ми. Изначально терминал представлял собой устройство, к которому подключаются клавиатура, экран и линия связи; текст, набираемый на клавиатуре, терминал отправлял в линию, а информацию, полученную «с того конца» линии, отображал на экране. Соединив линией связи два терминала, их операторы могли бы между собой пообщаться, но обычно между собой связывались не два терминала, а терминал и компьютер, причём один компьютер мог обслуживать несколько десятков терминалов одновременно; клавиатура терминала использовалась для набора команд командной строки, экран — для получения результатов.

Потоки ввода-вывода, связанные с терминалом, часто воспринимаются как некий особый случай, что и понятно: работа с терминалом подразумевает наличие «по ту сторону» живого пользователя. В частности, функции высокоуровневого ввода-вывода из стандартной библиотеки языка Си применяют для терминалов совершенно иную стратегию

[36]Терминал vt420 производства Digital Equipment Corporation (1993 г.) с прошитыми символами кодировки koi8-r. Экземпляр из коллекции автора. Пользуясь случаем, автор хотел бы поблагодарить Александра Песляка, известного также под творческим псевдонимом Solar Designer, за сделанный лет двадцать назад подарок.

буферизации ввода-вывода, нежели для любых других потоков (см. т. 2, §4.6.5). Более того, некоторые хорошо знакомые нам команды командной строки работают по-разному в зависимости от того, связан ли их поток вывода с терминалом или перенаправлен куда-то ещё; так, команда `ls` при работе с терминалом выстраивает имена файлов в несколько аккуратных колонок и (при соответствующих настройках) может сделать свой вывод разноцветным, вставив в него управляющие Esc-последовательности, тогда как при выводе в файл, канал или любой другой поток, не связанный с терминалом, та же команда выдаст все имена файлов в один столбик (собственно говоря, это будет просто список имён, разделённых символом перевода строки) и, конечно же, никаких спецсимволов, меняющих цвета, в свой вывод вставлять не будет. Чтобы увидеть эту разницу, перенаправьте вывод `ls` на вход программе `cat`, дав команду «`ls | cat`»; как известно, `cat` без параметров просто копирует стандартный ввод на стандартный вывод, но поскольку канал, связывающий эти две программы, терминалом не является, вы увидите выдачу команды `ls` в том варианте, который она применяет, работая не с терминалом.

Узнать, связан ли данный конкретный поток ввода-вывода с терминалом, позволяет функция `isatty`:

```
int isatty(int fd);
```

Параметром этой функции служит файловый дескриптор; если он связан с терминалом, функция возвращает `1`, в противном случае — `0`. Это библиотечная функция; нужную информацию она получает через уже знакомый нам системный вызов `fstat`.

Как мы уже обсуждали (см. т. 1, §1.4.1), алфавитно-цифровые терминалы в наше время не производятся, поскольку при необходимости с их ролью может справиться любой ноутбук, и это обойдётся гораздо дешевле специального устройства, выпускаемого малыми сериями; встретить аппаратные терминалы сегодня можно разве что в музеях и частных коллекциях. С логической точки зрения это ничего не меняет, функциональность терминала эмулируется программно, но терминал как сущность (пусть и виртуальная) при этом сохраняется практически в неизменном виде. Чаще всего в современных условиях отсутствует и линия связи; точнее говоря, она существует только в виде виртуального, программно эмулируемого объекта. Так, ядра Linux и FreeBSD эмулируют набор виртуальных терминалов, используя видеокарту и клавиатуру компьютера, на котором запущена система. Это те самые «консоли», доступные до запуска графической подсистемы; между ними можно переключаться нажатием комбинаций `Alt-F1`, `Alt-F2` и т. д., и если вам не требуется графика (например, для просмотра фотографий), то возможностей этих консолей вполне может хватить для работы с

компьютером. При работе с серверами системные администраторы часто предпочитают не запускать на них X Window — бóльшую часть работы выполняют в режиме удалённого доступа через сеть, а в тех редких случаях, когда с серверной машиной приходится иметь дело непосредственно, довольствуются текстовыми виртуальными консолями.

После запуска X Window виртуальные консоли никуда не деваются, перейти на них можно нажатием `Ctrl-Alt-F1`, `Ctrl-Alt-F2` и т. д, а вернуться в графику — нажатием `Alt-F7`, `Alt-F8` или чего-то вроде, конкретная комбинация зависит от конфигурации вашей системы (точнее, от того, какую виртуальную консоль использует X Window), но вместо них становится удобнее использовать специальные оконные приложения, эмулирующие терминал, такие как `xterm`, `rxvt`, `konsole` и т. п. В отличие от виртуальных консолей, где за эмуляцию терминала отвечает ядро системы, здесь функциональность терминала эмулирует обычная программа, выполняющаяся в виде процесса; но, как и в случае с виртуальными консолями, линия связи между терминалом и компьютером оказывается воображаемой.

Сравнительно редко возникает ситуация, требующая наличия реальной линии связи. Так, компьютеры, предназначенные для использования в качестве серверов, иногда выпускаются без видеокарты и порта для подключения клавиатуры; всё, что есть у такого компьютера для связи с внешним миром — это сетевые интерфейсы и консольный порт для подключения терминала. С аппаратной точки зрения это обычный COM-порт (известный также под названием RS-232), так что вместо терминала можно использовать обычный компьютер (чаще всего ноутбук), подключённый к такому порту трёхпроводным шнуром (раньше такой шнур называли «нуль-модемом»); на большинстве современных ноутбуков встроенного COM-порта уже нет, так что приходится применять переходник USB2COM, а функциональность терминала реализуется одной из существующих «терминальных программ», умеющих работать с COM-портом, таких как `minicom`, `hyperterm` или `termite`. Обычно через консольный порт выполняют только первичную настройку системы, а с того момента, когда начинает работать доступ к сети, дальнейшие действия производят удалённо.

Подробный рассказ о COM-портах мы опустим, но некоторые пояснения всё же будут уместны. Для передачи информации здесь используется три провода — по одному на каждое направление и ещё один в качестве общей «земли». Информация передаётся последовательно, то есть по одному биту; биты объединяются в группы (чаще всего — 8-битные байты, хотя это зависит от настроек), каждая группа снабжается служебными битами для обозначения начала и окончания передачи, а также, возможно, контрольным битом чётности, который служит для обнаружения ошибок, происходящих из-за электрических помех.

Изначально COM-порты предназначались для передачи данных через аналоговые (голосовые) телефонные линии, для чего с помощью специальных


Рис. 5.7. Порт RS-232: разъём (слева[38]) и штекер (справа)

устройств — ***модемов***[37] — цифровая информация переводилась в звуковой сигнал такого частотного диапазона, чтобы по возможности избежать потерь данных при передаче через телефонную сеть. Голосовые телефонные линии для передачи цифровых данных подходят весьма условно как раз из-за существенных искажений звукового сигнала; это не мешает различать слова при разговоре по телефону, но информационная ёмкость голосового разговора ничтожна по меркам компьютерных сетей. Самые лучшие аналоговые модемы на самых качественных голосовых линиях могли работать со скоростью 38400 ***бод*** (англ. *baud* — бит/с); учитывая наличие служебных битов в каждом байте, а также заголовков пакетов и прочих накладных расходов, на такой скорости передача одной фотографии с современного цифрового фотоаппарата занимает 15-20 минут, а то и больше, а на более низких скоростях, например, 9600 или 19200 бод, и вовсе растягивается на несколько часов, ну а о передаче видеофайлов можно было даже не думать. Несмотря на это, модемный доступ по коммутируемым телефонным линиям (диал-ап) на рубеже веков благодаря доступности именно этого вида линий связи был основным способом подключения к Интернету для широкой публики, его популярность пошла на спад лишь с появлением дешёвого и сравнительно быстрого доступа через сети мобильной связи.

Поскольку COM-порты предназначались в основном для подключения телефонных модемов, а скорость передачи данных через такие модемы никогда не была высокой, не было особого смысла бороться за увеличение скорости самих COM-портов. Большинство аппаратных реализаций COM-портов не поддерживает скорости выше, чем 115200 бод, что, конечно, очень мало для передачи данных при их современных объёмах, но при этом с хорошим запасом превосходит потребности в скорости при подключении терминала. В самом деле, мало кто умеет набирать текст со скоростью, превышающей 400 знаков в минуту, а для передачи текста с такой скоростью хватило бы (даже с некоторым запасом) скорости в 100 бод. Текст, передаваемый на терминал со стороны компьютера, имеет существенно бóльшие объёмы, поскольку может включать элементы оформления, управляющие escape-последовательности и т. п., к тому же работающие программы часто генерируют потоки текста, из которых человек

[38] Фото с сайта Викимедиа Коммонс; `https://commons.wikimedia.org/wiki/File:Serial_port.jpg`.

[37] Слово *modem* образовано как сокращение от *modulator and demodulator*, что буквальным образом отражает функцию этого устройства.

выхватывает только самые важные фрагменты; но и здесь скорость в 9600 бод обеспечивает вполне комфортные условия работы.

Отметим заодно, что при предоставлении удалённого доступа к машине также используется программная эмуляция терминала; в роли эмулятора выступает серверная программа (так называемый ***демон***), предоставляющая доступ — `sshd`, `telnetd` или их аналог.

Какова бы ни была природа терминального устройства, будь это физический терминал или эмулируемый (любым из имеющихся способов), между ним и программами, выполняющими операции ввода-вывода, находится специфический слой программного управления, называемый ***дисциплиной линии***[39]. Чтобы понять, в чем заключается её роль, рассмотрим простейший пример — нажатие комбинации клавиш `Ctrl-C`. Известно, что при этом активная программа получает сигнал `SIGINT`; вопрос в том, откуда этот сигнал берётся. Ясно, что обычный терминал — устройство, передающее и принимающее данные, — ничего не знает о сигналах ОС Unix и вообще может работать с разными операционными системами. Поэтому логично предположить, что сигнал генерирует сама система, получив от терминала некий специальный символ. Это действительно так: по нажатию `Ctrl-C` терминал просто генерирует символ с кодом `3`. Кстати, это верно не только для `Ctrl-C`: `Ctrl-A` генерирует `1`, `Ctrl-B` — `2` и т. д., комбинации `Ctrl-Z` соответствует код `26`, далее идут `Ctrl-[`, `Ctrl-\` и `Ctrl-]`, порождающие коды `27`, `28` и `29`. Получив символ с кодом `3`, драйвер терминала, подчиняясь настройкам дисциплины линии, рассылает процессам, работающим под управлением данного терминала[40], сигнал `SIGINT`.

Функции дисциплины линии не ограничиваются рассылкой сигналов пользовательским процессам. Например, когда активная программа читает текстовую информацию, вводимую пользователем, терминал рассматривает некоторые коды символов как команды для редактирования вводимой строки: нажатие клавиши Backspace приводит к удалению последнего введённого символа, и то же самое произойдёт, если вы нажмёте `Ctrl-Shift-?` (и то, и другое генерирует символ с кодом `127`), нажатие `Ctrl-W` удаляет целиком последнее введённое слово, `Ctrl-R` приводит к тому, что вводимая строка повторно отображается на экране (этим можно воспользоваться, если вывод какой-то другой программы испортил изображение текущей строки), `Ctrl-U` очищает вводимую строку, так и не передав её активной программе. Всё это происходит без участия активной программы, все необходимые действия выполняет дисциплина линии.

[39] Этот термин — не слишком удачная калька английского *line discipline*, но более удачного или хотя бы устойчивого перевода так и не появилось.

[40] Точнее, сигнал отправляется процессам текущей группы в сеансе, связанном с данным терминалом; разговор о группах процессов и сеансах у нас ещё впереди.

Как мы вскоре увидим, дисциплину линии можно перепрограммировать, чтобы драйвер терминала отправлял SIGINT по какой-либо другой комбинации клавиш или не отправлял его вовсе; после такого перепрограммирования символ с кодом 3 будет просто передан работающей программе на стандартный ввод. Точно так же могут быть изменены или отменены особые роли символов, редактирующих вводимую строку, и этим пользуются «продвинутые» программы, чтобы сделать редактирование строки удобнее. Вы уже наверняка привыкли к возможности использовать клавиши «стрелок», автодописыванию команд и имён файлов, повторению ранее введённых команд «стрелкой вверх»; программы, предоставляющие все эти возможности, такие как командный интерпретатор bash и отладчик gdb, вынуждены делать всё это сами, поскольку дисциплина линии так не умеет; зато для всего этого имеются библиотеки, наиболее известная из них называется GNU Readline.

Рассмотрим ситуацию с программой xterm. Ясно, что при нажатии Ctrl-C получить SIGINT должна не сама программа xterm, а те процессы, которые запущены в её окошке. Собственно говоря, сама по себе программа xterm, будучи оконным приложением, и не получает никакого SIGINT, по крайней мере, когда активно именно её окно и мы нажали Ctrl-C. Вместо этого она получает ***клавиатурное событие*** от системы X Window, свидетельствующее о нажатии комбинации клавиш. Но генерировать сигнал для запущенных под её управлением процессов ей не нужно, достаточно передать символ с кодом 3 драйверу псевдотерминала, и драйвер (благодаря настройкам дисциплины линии) поступит так же, как если бы на месте программы был настоящий терминал — то есть перехватит символ и вместо него выдаст сигнал SIGINT.

Аналогично обстоят дела и при нажатии Ctrl-D (имитация конца файла). Программе xterm не нужно закрывать канал связи с активной программой, выполняющейся в её окошке, тем более что этого и нельзя делать, ведь сеанс одним EOF'ом не заканчивается, в нём могут быть и другие запущенные программы: представьте, что вы в окошке xterm'а запустили команду cat, потом нажали Ctrl-D, но вместо того, чтобы вернуться в командный интерпретатор, xterm закрылся; вряд ли вам такое понравится. Программа xterm действует иначе: она просто передаёт драйверу терминала символ, соответствующий комбинации Ctrl-D, то есть символ с кодом 4. Получив его, драйвер терминала — та его часть, которую называют дисциплиной линии — обеспечит, чтобы ближайший вызов read, выполненный на этом логическом терминале, вернул 0, то есть сигнализировал о ситуации «конец файла».

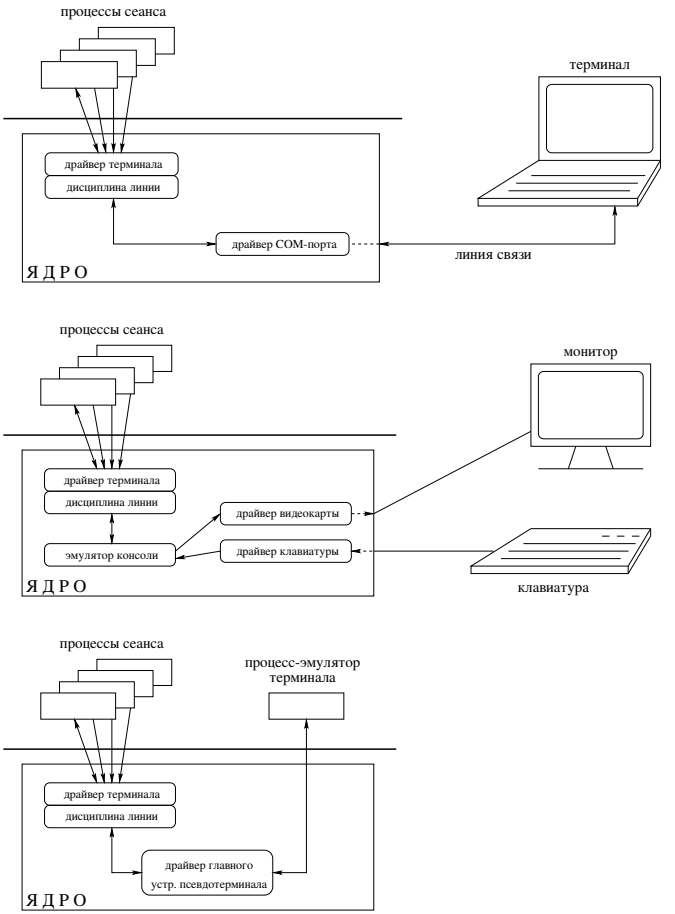

Рис. 5.8. Сеанс под управлением реального терминала (вверху), виртуальной консоли (в середине) и псевдотерминала (внизу)

5.6.2. Сеансы и группы процессов

Для объединения процессов, запущенных пользователем в ходе работы с одного терминала, в unix-системах введено понятие ***сеанса***. Реализуется эта сущность очень просто: у каждого процесса есть ***идентификатор сеанса*** (`sid`, *session id*), а сам сеанс (буквально) состоит из процессов, имеющих одинаковый идентификатор сеанса, то есть сеанс представляет собой определённое множество процессов. Сеансу может принадлежать ***управляющий терминал***, не более чем один (возможно, ни одного); с другой стороны, терминал может служить управляющим для сеанса, но не более чем одного (или ни для одно-

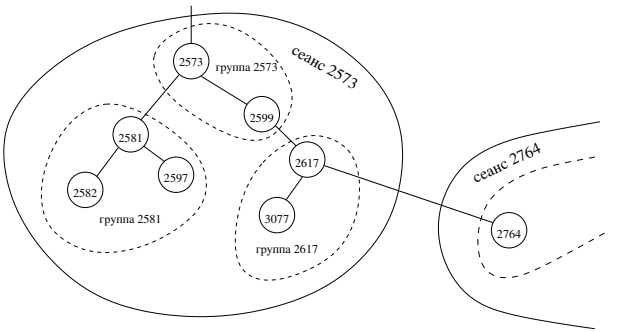

Рис. 5.9. Сеансы и группы процессов

го). Сеанс обычно создаётся в тот момент, когда пользователь вошёл в систему через виртуальную консоль или с использованием удалённого доступа (или с помощью настоящего терминала, хотя в наше время такое встречается редко). Менее очевидно, что новый сеанс создаётся, когда пользователь запускает программу, эмулирующую терминал, например, `xterm`; иначе говоря, при работе в оболочке X Window у вас будет по меньшей мере столько сеансов, сколько открыто окошек `xterm` и других подобных программ, плюс ещё один, созданный при запуске оконного менеджера.

В рамках одного сеанса процессы могут быть разбиты на отдельные ***группы***. Группа процессов входит в сеанс целиком, то есть в одну группу не могут входить процессы из разных сеансов. Минимум одна группа в сеансе всегда присутствует, она возникает при создании сеанса. В каждом сеансе, имеющем управляющий терминал, одна группа процессов называется ***текущей группой***[41], а остальные группы того же сеанса — ***фоновыми группами***[42]. Любая из групп сеанса, имеющего управляющий терминал, может быть в любой момент назначена текущей, тогда бывшая текущая становится фоновой. С некоторой натяжкой можно считать, что в сеансе, не имеющем управляющего терминала, все группы являются фоновыми, но на самом деле текущая группа — это, как мы вскоре увидим, свойство терминала, а не сеанса, так что в сеансе без терминала понятие текущей группы просто некому поддерживать.

Группы процессов изначально задуманы для объединения процессов, работающих над общей задачей. Командный интерпретатор обычно создаёт новую группу процессов для выполнения каждой поданной команды, так что все процессы, порождённые в ответ на каждую конкретную команду пользователя, оказываются объединены в одну группу; в частности, при запуске конвейера все его элементы объединяются в

[41] Английский оригинал этого термина — *foreground group*, что можно буквально перевести как «группа переднего плана».

[42] Англ. *background group*.

группу. Группа процессов имеет свой идентификатор, который называют `pgid` (*process group id*).

Процесс наследует принадлежность к сеансу и группе от своего непосредственного предка — при вызове `fork` параметры `sid` и `pgid` не изменяются. Однако процесс может при желании уйти в новую группу или даже в новый сеанс. Говоря конкретнее, процесс может:

- создать сеанс; одновременно создаётся и группа, причём процесс, создавший сеанс, автоматически становится лидером и сеанса, и группы; идентификатор сеанса и идентификатор группы всегда равны идентификатору процесса, их создавшего;
- создать группу в рамках того же сеанса, при этом процесс становится лидером группы;
- перейти в другую группу того же сеанса.

Отметим ещё один момент: процесс не может создать новый сеанс, если он уже является лидером сеанса, то есть его `sid` совпадает с `pid`'ом, и не может создать новую группу, если уже является лидером группы, то есть его `pgid` совпадает с `pid`'ом. Это утверждение можно усилить: если процесс является лидером группы (или тем более лидером сеанса), то он не может создавать ни групп, ни сеансов. В самом деле, всякий лидер сеанса является также и лидером группы, возникшей в момент создания сеанса; как следствие, он не может создавать ни групп, ни сеансов, ведь идентификатор того и другого уже занят (им же самим). С другой стороны, лидер группы, не являющийся лидером сеанса, не может создать новый сеанс, ведь при этом должна возникнуть группа с тем же идентификатором, а этот идентификатор уже занят.

Все эти механизмы введены в ОС Unix для облегчения управления процессами. Так, если пользовательская сессия работы с терминалом по той или иной причине завершилась (например, пользователь выключил терминал, разорвал соединение удалённого доступа или закрыл окно программы `xterm`), то всем процессам сеанса, связанного с этим терминалом, рассылается сигнал `SIGHUP`. Что касается групп, то они, например, используются при рассылке сигнала `SIGINT` по нажатию `Ctrl-C`: сигнал получают только процессы текущей группы, а фоновые продолжают работу. Более того, сам по себе ввод с терминала разрешён только процессам текущей группы. Фоновый процесс при попытке чтения с терминала приостанавливается сигналом `SIGTTIN`. Можно (но не обязательно) запретить фоновым группам также и вывод на терминал. Вообще механизм сеансов и групп создан в основном для того, чтобы можно было установить, каких процессов непосредственно касаются действия, выполняемые пользователем терминала.

Рассмотрим кратко системные вызовы и функции, имеющие отношение к управлению сеансами и группами. Узнать для процесса параметры `sid` и `pgid` можно с помощью вызовов `getsid` и `getpgid`:

```
int getsid(int pid);
```

```
int getpgid(int pid);
```

где `pid` — идентификатор интересующего нас процесса. Специальное значение `0` означает вызывающий процесс. Отметим, что идентификатор сеанса совпадает с идентификатором (`pid`) процесса, создавшего этот сеанс; обычно идентификатор группы также совпадает с `pid`'ом создавшего группу процесса. Соответствующие процессы называются ***лидерами*** соответственно сеанса и группы.

Создание нового сеанса производится вызовом `setsid`:

```
int setsid();
```

Вызов не проходит, если данный процесс уже является лидером сеанса или хотя бы группы. Чтобы гарантировать успешное создание сеанса, следует вызвать `fork` и завершить родительский процесс, сменив, таким образом, свой `pid`:

```
if(fork() > 0)
    exit(0);
setsid();
```

При успешном выполнении `setsid` будут созданы одновременно новый сеанс и новая группа, идентификаторы которых будут совпадать с `pid`'ом процесса, выполнившего `setsid`, причём вызвавший процесс окажется единственным членом и того, и другого.

Как уже говорилось, каждый терминал может быть управляющим для не более чем одного сеанса и каждый сеанс может иметь не более одного управляющего терминала. При успешном выполнении `setsid` процесс теряет управляющий терминал, даже если у него остаются связанные с терминалом открытые дескрипторы. Чтобы снова получить управляющий терминал, процессу следует открыть вызовом `open` файл терминального устройства, не управляющего другим сеансом; интересно, что получить управляющий терминал может только процесс, создавший сеанс (лидер сеанса). С другой стороны, лидер сеанса, не обладающего управляющим терминалом, должен соблюдать известную осторожность при открытии файлов, чтобы случайно не захватить себе свободный терминал, когда это не требуется; в частности, можно во всех вызовах `open` добавлять во втором параметре флаг `O_NOCTTY`, который блокирует превращение открываемого терминала в управляющий, даже если все остальные условия выполнены (вызывающий процесс является лидером сеанса, сеанс не обладает управляющим терминалом, открываемый файл является терминальным устройством, с которым не связан существующий сеанс).

Для смены группы предназначен вызов `setpgid`:

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

Параметр `pid` задаёт номер процесса, который нужно перевести в другую группу; `pgid` — номер этой группы. Сменить группу процесс может либо самому себе, либо своему непосредственному потомку, если только этот потомок ещё не выполнил вызов `execve`. Группу можно менять либо на уже существующую в рамках сеанса, либо можно задать параметр `pgid` равным параметру `pid`, в этом случае создаётся новая группа, а процесс становится её лидером.

Если у сеанса есть управляющий терминал, можно указать драйверу терминала, какую группу процессов считать текущей. Это делается с помощью библиотечной функции `tcsetpgrp`:

```
int tcsetpgrp(int fd, int pgrp);
```

Здесь `fd` — файловый дескриптор, который должен быть связан с управляющим терминалом; обычно используется 0 или 1. Дескриптор нужен, потому что смена основной группы реализуется через вызов `ioctl` для терминала как логического устройства; дело в том, что за понятие *текущей группы* в сеансе отвечает драйвер управляющего терминала, что и понятно, ведь именно ему нужно знать, каким процессам, например, разослать тот же `SIGINT`, если пользователь нажал Ctrl-C.

5.6.3. Управление драйвером терминала

Из предшествующего опыта нам уже известны некоторые особенности работы терминала. Так, обычно активная программа не получает символы пользовательского ввода до тех пор, пока пользователь не завершит ввод строки, после чего программе передаётся сразу вся строка; при вводе символы, набираемые на клавиатуре, отображаются на экране; нажатие определённых комбинаций клавиш приводит к специфическим эффектам и т. д.

Создавая полноэкранные программы с помощью паскалевского модуля `crt`, а позже — на Си с использованием библиотеки `ncurses`[43], мы много раз упоминали возможность *перепрограммирования* драйвера терминала. Этому вопросу мы и уделим сейчас внимание.

Как и для любого устройства, представленного в системе файлом специального типа, для управления терминалом — а точнее, для управления *линией связи с терминалом* — используется системный вызов `ioctl`. Для удобства в библиотеку языка Си включена целая группа функций под общим названием `termios`, специально предназначенных для управления терминалом; отметим, что в эту группу входит упоминавшаяся в предыдущем параграфе функция `tcsetpgrp`. Создавался интерфейс `termios` ещё в те годы, когда аппаратные терминалы встречались чаще виртуальных, а поскольку подключались они в большинстве

[43] См. т. 1, §2.11; т. 2, §4.15.

случаев через линии связи RS-232, функции модуля `termios` предоставляют целый ряд возможностей, которые сегодня могут показаться рудиментарными, как, например, установка скорости передачи данных. На самом деле все эти возможности по-прежнему используются в тех не столь уж редких случаях, когда к компьютеру всё-таки подключают физическую линию RS-232 или аналогичную (например, интерфейс RS-485 в наши дни довольно часто используется для управления промышленной автоматикой). На другом конце такой линии связи сейчас вряд ли окажется терминал, но ещё 20–25 лет назад именно терминалы были самым популярным видом оборудования, подключаемого к unix-машинам через RS-232, и это хорошо знают специалисты, работающие с последовательными интерфейсами (например, в сфере промышленной автоматики), поскольку весь интерфейс функций `termios` построен в предположении, что скорее всего по ту сторону COM-порта находится терминал, а если нет — то это исключение из общего правила.

Взаимодействие с физическим последовательным портом — предмет достаточно увлекательный, но требуется в наши дни редко и в специфических случаях, поэтому рассматривать его мы не будем. Совсем другое дело — управление режимом работы терминала (дисциплиной линии). Конечно, при написании полноэкранных программ всё общение с драйвером терминала берут на себя соответствующие библиотеки, но иногда требуется решить совсем простенькую задачу вроде ввода пароля или какого-нибудь пин-кода (так, чтобы вводимые символы при этом не отображались на экране), и ради этого попросту глупо подключать к проекту монстров вроде `ncurses`.

При управлении терминалом следует помнить, что основных режимов работы терминала существует ровно два: ***канонический*** и ***неканонический***. Основное различие между ними в том, будет ли драйвер терминала накапливать вводимую строку символов, чтобы отдать её активной программе сразу всю, или же каждый введённый символ окажется доступен активной программе немедленно. Кроме основных режимов, драйвер терминала поддерживает несколько десятков менее значимых опций, многие из которых в современных условиях неприменимы, но некоторые (вроде отображения вводимых символов, *local echo*) достаточно важны и умение ими управлять может оказаться полезно. Наконец, драйвер терминала (точнее, дисциплина линии) позволяет полностью переназначить роли всех спецсимволов; например, при желании вы можете заставить дисциплину линии имитировать «конец файла» не по `Ctrl-D`, а по `Ctrl-N`, сигнал `SIGINT` посылать по нажатию клавиши Escape, удалять последний введённый символ по `Ctrl-B` и т. п. — иной вопрос, *нужно ли* так делать.

Все настройки дисциплины линии собраны в одну структуру `struct termios`; вы можете получить текущие настройки термина-

ла с помощью функции `tcgetattr` и установить новые с помощью `tcsetattr`:

```
int tcgetattr(int fd, struct termios *tp);
int tcsetattr(int fd, int options, const struct termios *tp);
```

В качестве первого параметра обе функции получают файловый дескриптор потока ввода-вывода, связанного с настраиваемым терминалом. Скорее всего, вам подойдёт дескриптор `0`, если только пользователь, запустивший вашу программу, не перенаправил ей поток стандартного ввода, но это можно проверить с помощью `isatty` (см. стр. 127).

При установке нового режима работы терминала с помощью `tcsetattr` нужно её вторым параметром (`options`) указать, в какой момент драйверу следует действительно перейти на новый режим работы. Здесь имеется три варианта: `TCSANOW` (применить новые настройки немедленно), `TCSADRAIN` (дождаться, пока все данные, записанные в поток вывода, связанный с `fd`, не будут физически переданы в линию связи) и `TCSAFLUSH` (дождаться, пока все записанные данные уйдут в линию, *сбросить все данные, пришедшие из линии, но пока не прочитанные*, и после этого применить новые настройки). В большинстве случаев годится `TCSANOW`.

Структура `termios` содержит как минимум следующие поля:

```
int c_iflag;
int c_oflag;
int c_cflag;
int c_lflag;
char c_cc[NCCS];
```

На самом деле полей больше, но функции, с которыми мы будем работать, их не используют. Кроме того, иногда может оказаться важно, что поля флагов в системных заголовочных файлах описаны как имеющие тип `tcflag_t`, а не `int`, а элементы массива `c_cc` имеют тип `cc_t`, а не `char`; например, в системе автора этих строк `tcflag_t` был определён как синоним для `unsigned int`, `cc_t` — как синоним `unsigned char`. В большинстве случаев это несущественно.

Поле `c_iflag` содержит флаги, отвечающие за преобразование входящего потока информации (буква `i` означает *input*), то есть информации, пришедшей со стороны терминала. Здесь единственный флаг, эффект от которого понятен и полезен — это `IXON`; если он установлен, то вывод на терминал можно в любой момент приостановить нажатием комбинации `Ctrl-S`, а затем продолжить нажатием `Ctrl-Q` (если, конечно, мы не перепрограммировали коды для этих двух операций). Эта возможность полезна, если какая-то программа выдаёт текст на экран слишком быстро, а пользователю нужно внимательно прочитать фрагмент выдачи, который норовит исчезнуть за верхним краем экрана.

Остальные флаги из набора `c_iflag` выполняют не столь очевидные функции. Например, установка флага `IGNCR` позволяет игнорировать символы «возврат каретки» (`CR`, код 13), т. е. если установить этот флаг, драйвер терминала перехватит все символы с кодом 13, поступающие с терминала, и ни один из них активной программе не отдаст. В современных условиях этот флаг довольно бесполезен; он позволял работать с терминалами (естественно, аппаратными), которые при нажатии Enter передавали в линию связи два байта — возврат каретки и перевод строки. Флаг `ISTRIP` позволяет от всех входящих символов «откусить» старший бит; между прочим, это сделает нечитаемым текст на русском языке в любой кодировке, кроме koi8-r (см. т. 1, стр. 181), но вот как извлечь из этого практическую пользу, остаётся неясным. Подчеркнём, что эти два флага мы привели лишь для примера, чтобы создать некоторое впечатление относительно возможностей поля `c_iflag`. Назначение большинства других флагов из этого поля вообще невозможно понять без досконального анализа принципов работы с асинхронной последовательной линией связи, и вряд ли хотя бы один из них вам пригодится.

Флаги, содержащиеся в поле `c_oflag` (`o` от слова *output*), позволяют управлять обработкой потока, *выдаваемого* на терминал. К примеру, `ONLRET` запрещает вывод символа возврата каретки, то есть драйвер терминала «скушает» все символы с кодом 13, выдаваемые активными программами, и в линию связи ни один из них не попадёт; флаг `OLCUC` все строчные латинские буквы превратит в заглавные; от остальных флагов из этого набора, пожалуй, пользы ещё меньше.

Среди флагов из поля `c_cflag` (`c` от слова *control*) вообще нет ни одного такого, который можно было бы описать в отрыве от тонкостей работы с физическими последовательными линиями связи и модемами. Когда линия связи с терминалом существует лишь в нашем воображении, ничего интересного с этими флагами сделать не удастся.

Совершенно иначе обстоят дела с полем `c_lflag` (`l` от слова *local*); всё самое интересное, что можно сделать с драйвером терминала, сосредоточено именно здесь. Пожалуй, самым «сильным» можно считать флаг `ICANON`: пока он взведён, терминал работает в каноническом режиме, то есть дожидается ввода строки целиком, прежде чем отдать её активной программе, обрабатывает специальные символы, предназначенные для удаления последнего символа и очистки всей строки (по умолчанию Backspace и `Ctrl-U`). Если добавить флаг `IEXTEN`, терминал будет обрабатывать также спецсимволы для удаления последнего введённого слова (`Ctrl-W`) и для восстановления (повторной печати) введённой части строки (`Ctrl-R`); следует учесть, что флаг `IEXTEN` без `ICANON` не работает.

Флаг `ISIG` указывает драйверу терминала, следует ли обрабатывать специальные символы, предполагающие отправку сигнала текущей груп-

пе процессов. Это хорошо знакомые нам комбинации Ctrl-C, Ctrl-\ и Ctrl-Z, отправляющие соответственно сигналы SIGINT, SIGQUIT и SIGTSTP.

Довольно интересен флаг TOSTOP: он определяет, разрешено ли неактивным (фоновым) процессам выдавать данные на терминал. Если этот флаг сброшен, вывод на терминал могут осуществлять все процессы, у которых для этого достаточно полномочий; иногда из-за этого вывод разных программ создаёт путаницу. Если же флаг TOSTOP взвести, то попытка фонового процесса что-то напечатать приведёт к отправке всей его группе сигнала SIGTTOU; по умолчанию это приостановит работу процессов. Отметим, что аналогичного флага для *ввода* данных с терминала не предусмотрено, делать это имеет право только текущая группа процессов, а фоновые группы при попытке что-то прочитать со своего управляющего терминала немедленно получают сигнал SIGTTIN, что также приводит к приостановке работы. Возобновить работу остановленных процессов можно сигналом SIGCONT.

Пожалуй, один из самых употребимых флагов здесь — ECHO; он определяет, следует ли выдавать на экран вводимые символы по мере их ввода (так называемый режим *local echo*). В каноническом режиме этот флаг обычно установлен, так что пользователь видит, что́ он вводит; но иногда желательно, чтобы какая-то информация вводилась «вслепую» — например, если это пароль. В такой ситуации мы можем сбросить флаг ECHO на время ввода секретного текста, а затем включить его обратно:

```
/* blindpsw.c */
#include <stdio.h>
#include <unistd.h>  /* for isatty */
#include <termios.h>
#include <string.h>  /* for memcpy */
enum { bufsize = 128 };
int main()
{
    struct termios ts1, ts2;
    char buf[bufsize];
    if(!isatty(0)) {
        fprintf(stderr, "Not a terminal, sorry\n");
        return 1;
    }
    tcgetattr(0, &ts1);         /* получаем текущие настройки */
    memcpy(&ts2, &ts1, sizeof(ts1));         /* создаём копию */
    ts1.c_lflag &= ~ECHO;             /* сбрасываем флаг ECHO */
    tcsetattr(0, TCSANOW, &ts1);      /* выключаем local echo */
    printf("Please blind-type the code: ");
    if(!fgets(buf, sizeof(buf), stdin)) {
        fprintf(stderr, "Unexpected end of file\n");
```

```
            return 1;
        }
        printf("\nThe code you entered is [%s]\n", buf);
        tcsetattr(0, TCSANOW, &ts2); /* восстанавливаем настройки */
        return 0;
    }
```

В этой программе мы сначала получаем текущие настройки терминала, затем модифицируем их и устанавливаем модифицированную версию, а позже восстанавливаем исходные настройки. Этот подход типичен при «хитрых» операциях с терминалом, поскольку заполнить структуру `termios` «с чистого листа» не так просто. На всякий случай напомним, что отдельные флаги, относящиеся к одному полю структуры, следует объединять операцией побитового «или» («`|`»). Установить в заданном поле заданные флаги можно с помощью операции «`|=`» (побитовое «или», совмещённое с присваиванием); например, установить несколько флагов в поле `c_lflag`, можно примерно так:

```
ts1.c_lflag |= (ICANON | IEXTEN | ISIG | ECHO);
```

Для принудительного сброса заданных флагов в заданном поле поступают следующим образом. Сформированную комбинацию флагов побитово инвертируют с помощью операции «`~`», так что биты, соответствующие нужным флагам, оказываются равны нулю, а все остальные — единице, и полученное значение используют в качестве «маски» для побитового «и»; обычно маску накладывают операцией «`&=`»:

```
ts1.c_lflag &= ~(ICANON | IEXTEN | ISIG | ECHO);
```

Именно так мы и поступили с флагом `ECHO` в примере, приведённом выше. На всякий случай подчеркнём, что все эти операции только изменяют поля вашей структуры, но никак не влияют на рабочие настройки терминала, которые меняются только когда будет вызвана функция `tcsetattr`.

Последнее документированное поле структуры `termios` называется `c_cc` (`cc` образовано от слов *control codes* — управляющие коды). Элементы этого массива имеют тип `char`; большинство из них (но не все!) содержат коды символов, которые дисциплина линии должна обрабатывать особым образом. Чтобы не нужно было помнить, какой из элементов массива содержит тот или иной код, заголовочные файлы определяют целый ряд констант, значения которых соответствуют индексам в массиве `c_cc`. Например, `c_cc[VEOF]` обычно содержит значение `4`, что соответствует использованию `Ctrl-D` для имитации ситуации «конец файла» (напомним, что комбинации вида `Ctrl-`X генерируют «символ» с кодом, соответствующим номеру буквы X в латинском алфавите; в частности, `Ctrl-D` генерирует код `4`). Изменив значение этого

поля, можно заставить дисциплину линии использовать для имитации конца файла какой-то другой код — конечно, при условии, что в поле `c_lflag` установлен флаг `ICANON`.

Аналогично `c_cc[VINTR]`, `c_cc[VQUIT]` и `c_cc[VSUSP]` задают коды, используемые для порождения сигналов `SIGINT`, `SIGQUIT` и для приостановки активной задачи сигналом `SIGTSTP`; обычно в этих элементах содержатся коды `3`, `28` и `26`, соответствующие комбинациям `Ctrl-C`, `Ctrl-\` и `Ctrl-Z`. Естественно, эти комбинации работают при установленном флаге `ISIG`. Коды из `c_cc[VSTOP]` и `c_cc[VSTART]` используются для приостановки и возобновления вывода на экран терминала (при установленном `IXON` в поле `c_iflag`), по умолчанию это `19` (`Ctrl-S`) и `17` (`Ctrl-Q`).

Коды `c_cc[VERASE]`, `c_cc[VKILL]`, `c_cc[VWERASE]` и `c_cc[VREPRINT]` используются соответственно для удаления последнего символа, удаления всей текущей строки, удаления последнего слова и повторной печати вводимой строки. Первые два из них требуют только включённого флага `ICANON`, для двух остальных нужен ещё и `IEXTEN`. По умолчанию это коды `127` (Backspace), `21` (`Ctrl-U`), `23` (`Ctrl-W`) и `18` (`Ctrl-R`).

Интересно, что код, используемый для перевода строки, перепрограммировать нельзя, это всегда `10`. Присутствующие среди констант `VEOL` и `VEOL2` обозначают элементы массива `c_cc`, задающие *дополнительные* символы для перевода строки; в большинстве случаев они не задействуются.

Если обработку того или иного специального кода нужно отключить, в соответствующий элемент массива `c_cc` заносится ноль; например, если занести ноль в `c_cc[VINTR]`, комбинация `Ctrl-C` работать перестанет, причём сигнал `SIGINT` не будет генерироваться вообще никакой комбинацией клавиш. В последние годы для обозначения этого нуля ввели константу `_POSIX_VDISABLE`, но вероятность того, что когда-то где-то эта константа окажется отличной от нуля, пожалуй, не стоит рассмотрения.

Если отключить канонический режим — а его, как правило, отключают вместе с `ISIG` — то все вышеперечисленные коды работать перестанут; зато при этом в работу включатся `c_cc[VMIN]` и `c_cc[VTIME]`, позволяющие управлять тем, когда (при каких условиях) будет возвращать управление вызов `read` в активной программе. Значение `c_cc[VMIN]` обозначает минимальное количество прочитанных из порта символов, а значение `c_cc[VTIME]` — время ожидания (в десятых долях секунды). В большинстве случаев `c_cc[VMIN]` устанавливают равным единице, а `c_cc[VTIME]` — равным нулю; при этом `read` будет работать так, как мы привыкли — немедленно возвращать управление, если доступен (введён на клавиатуре) хотя бы один байт, в противном случае блокироваться и ждать, пока этот байт не появится.

Второй достойный упоминания вариант — нули в обоих элементах; в этом случае `read` всегда будет возвращать управление немедленно, причём если ни одного байта прочитать не удалось, он вернёт `0`. Заметим, что практически такого же эффекта можно достичь, переведя дескриптор ввода в неблокирующий режим с помощью `fcntl` (см. стр. 57), только при этом `read` будет возвращать `-1`, занося в `errno` значение `EAGAIN`.

Варианты с отличным от нуля `c_cc[VTIME]` обычно применяются при коммуникации по COM-порту с устройством, отличным от терминала. На всякий случай отметим, что семантика этого параметра зависит от значения `c_cc[VMIN]`: если там ноль, то установленный отрезок времени отсчитывается от входа в вызов `read`, и он возвращает управление не позднее чем через заданное время; если же в `c_cc[VMIN]` не ноль, отсчёт начинается после поступления первого символа и сбрасывается при поступлении каждого следующего, а возврат управления происходит, если превышен заданный лимит времени *между* двумя символами.

Читателю, заинтересованному в дополнительных подробностях, мы можем порекомендовать статью «The TTY demystified» [10], а также справочную страницу, доступную по команде `man 3 termios`.

5.6.4. По ту сторону псевдотерминала

Итак, линия связи с терминалом в современных условиях — вещь чаще всего сугубо виртуальная, а сам терминал эмулируется теми или иными программами, причём все случаи такой эмуляции можно разделить на два класса: виртуальные консоли, предоставляемые ядром операционной системы, и, собственно говоря, *всё остальное* — когда за эмуляцию терминала отвечает обычная пользовательская программа. Как система создаёт виртуальные консоли — внутреннее дело самой системы, а вот для эмуляции терминала пользовательскими программами предусмотрен довольно интересный механизм ***псевдотерминалов*** — специальных объектов ядра, имитирующих как раз ту самую воображаемую линию связи с терминалом.

Псевдотерминал как объект ядра имеет два двусторонних канала связи, один — для программы, эмулирующей функционирование терминала (например, `xterm`), второй — для программ, выполняющихся под управлением терминала. Программа, эмулирующая терминал, в этом виде взаимодействия называется *главной* (master), а работающие под управлением терминала — *подчинёнными* (slaves).

Чтобы создать псевдотерминал, главная программа должна вызвать специально предназначенную для этого функцию. Большинство версий стандартной библиотеки, следуя за `glibc`, предоставляют для этого функцию `getpt`:

```
int getpt();
```

но в документации к ней сказано, что она специфична для `glibc` и в переносимых программах следует использовать функцию с корявым именем `posix_openpt`:

```
int posix_openpt(int flags);
```

В качестве параметра эта функция может принимать комбинацию флагов `O_RDWR` и `O_NOCTTY`[44], и обычно (а точнее, судя по всему, просто всегда) при вызове `posix_openpt` указывают оба этих флага: `posix_openpt(O_RDWR|O_NOCTTY)`. Такой вызов эквивалентен вызову `getpt` без параметров, и в обоих случаях на самом деле происходит системный вызов `open`, применённый к специальному файлу виртуального устройства `/dev/ptmx`. При этом ядро создаёт пару связанных между собой устройств, которые называются ***главное устройство псевдотерминала*** (англ. *pseudoterminal master*, ptm) и ***подчинённое устройство псевдотерминала*** (*pseudoterminal slave*, pts). Первое из них ядро открывает в виде потока ввода-вывода и возвращает файловый дескриптор этого потока. Одновременно в файловой системе появляется файл подчинённого устройства, открытие которого позволит присоединиться к тому же псевдотерминалу уже со стороны программ, для которых он будет управляющим.

Подчинённый псевдотерминал сразу открыть не получится, поскольку для него ещё не настроены права доступа — во всяком случае, эти права могут не соответствовать задаче, которую пытается решить создатель псевдотерминала. Чтобы сделать подчинённое устройство доступным для открытия, нужно применить к *дескриптору главного устройства* (то есть тому дескриптору, который нам вернула функция `getpt`, `posix_openpt` или просто `open`, применённый к `/dev/ptmx`) последовательно функции `grantpt` и `unlockpt`:

```
int grantpt(int fd);
int unlockpt(int fd);
```

Первая из них изменяет принадлежность файла устройства подчинённого псевдотерминала так, что он становится доступен владельцу текущего процесса. Если программа, эмулирующая терминал, изначально работала с правами нужного пользователя (как, например, `xterm`), она может вызвать `grantpt` сразу после получения дескриптора ведущего терминального устройства. Часто бывает и так, что программа, создающая псевдотерминал, исходно работала в системе с полномочиями администратора (например, программа-сервер для удалённого доступа к системе); такие программы, убедившись, что имеют дело с определённым пользователем системы — например, проверив пароль или криптографическую подпись — отказываются от привилегий, установив

[44] Флаг `O_NOCTTY` мы уже обсуждали на стр. 135.

себе `uid`, `euid`, `gid`, `egid` того пользователя, для которого запускается сеанс работы, и уже потом вызывают `grantpt`.

Функция `unlockpt` разрешает открыть файл псевдотерминала с помощью вызова `open`; до этого он недоступен к открытию. Прежде чем вызвать эту функцию, главная программа может установить права доступа к подчинённому псевдотерминалу с помощью вызова `chmod`. После выполнения `unlockpt` псевдотерминал готов к работе и его можно открыть с помощью `open`. Например, можно создать для этого отдельный процесс, там закрыть потоки стандартного ввода, вывода и ошибок, создать новый сеанс вызовом `setsid`, после чего открыть псевдотерминал и связать его дескриптор со всеми тремя потоками, а потом выполнить `exec` для запуска программы, которая будет лидером нового сеанса. Узнать имя файла устройства подчинённого псевдотерминала можно с помощью функции

```
char *ptsname(int master_fd);
```

где `master_fd` — дескриптор, полученный от `getpt` или `posix_openpt`.

Всю работу, связанную с созданием описанной связки главный-подчинённый, можно выполнить и проще, с помощью одной функции `openpty`:

```
int openpty(int *master, int *slave, char *name,
            struct termios *termp, struct winsize *winp);
```

Параметры `master` и `slave` задают адреса переменных, в которые следует записать дескрипторы, связанные соответственно с главным и подчинённым каналами связи с псевдотерминалом. Параметр `name` указывает на буфер, куда следует записать имя подчинённого псевдотерминала, параметры `termp` и `winp` задают режим работы псевдотерминала. В качестве любого из последних трёх параметров можно передать нулевой указатель.

Функция `openpty` считается особенностью систем семейства BSD; в литературе можно встретить рекомендации воздержаться от её использования.

5.6.5. Процессы-демоны

Под ***демонами***[45] понимаются процессы, не предназначенные для непосредственного взаимодействия с пользователями системы. Примерами демонов могут служить серверные программы, обслуживающие WWW или электронную почту. Существуют демоны и для выполнения внутрисистемных задач: так, демон `crond` позволяет автоматически запускать различные программы в заданные моменты времени, а демон системы печати собирает от пользователей задания на печать и отправляет их на принтеры. Демоны обычно расчитаны на длительное функционирование; в некоторых системах демоны могут годами

[45] Слово «демон» в данном случае представляет собой прямой перевод английского *daemon*. Такой вариант перевода не совсем удачен, но другого всё равно нет.

работать без перезапуска. При старте демона принимаются меры, чтобы его функционирование не мешало работе и администрированию системы. Так, текущий каталог обычно меняется на корневой, чтобы не мешать системному администратору при необходимости удалять каталоги, монтировать и отмонтировать диски и т. п.

Хотя демону не нужен управляющий терминал (и вообще дескрипторы стандартного ввода, вывода и выдачи сообщений об ошибках), желательно, чтобы дескрипторы 0, 1 и 2 оставались открыты, потому что демоны, естественно, работают с файлами, и если тот или иной файл будет открыт с номером дескриптора 0, 1 или 2 (а это обязательно произойдёт, если дескрипторы просто закрыть), какая-нибудь процедура может случайно испортить файл, попытавшись выполнить ввод-вывод на стандартных дескрипторах. Поэтому все три стандартных дескриптора обычно связывают с устройством `/dev/null` (см. §5.3.5).

Чтобы действия, производимые с каким-либо терминалом, не сказались на функционировании демона (например, было бы нежелательно получить в некий момент `SIGHUP` из-за того, что пользователь прекратил работу с терминалом), он обычно работает в отдельном сеансе. Наконец, во избежание ошибочного получения управляющего терминала демон обычно делает лишний `fork`, чтобы перестать быть лидером сеанса. В целом процедура старта процесса-демона (так называемая «демонизация», англ. *daemonization*) может выглядеть приблизительно так:

```
close(0);
close(1);
close(2);
open("/dev/null", O_RDONLY); /* stdin  */
open("/dev/null", O_WRONLY); /* stdout */
open("/dev/null", O_WRONLY); /* stderr */
chdir("/");
if(fork() > 0)
    exit(0);
setsid();
if(fork() > 0)
    exit(0);
```

Обратите внимание, что первая связка `fork-exit` выполняется перед вызовом `setsid`, чтобы ничто не мешало процессу создавать новый сеанс. Если этого не сделать, имеется риск, что `setsid` не пройдёт: например, процесс уже быть лидером сеанса или хотя бы группы. Вторая такая же связка выполняется уже после `setsid`, чтобы процесс, породив сеанс, перестал быть его лидером.

Поскольку с терминалами процесс-демон не связан, всевозможные сообщения об ошибках, предупреждения, информационные сообщения

и т. п., адресованные системному администратору, приходится передавать неким альтернативным способом. Обычно это делается через инфраструктуру ***системной журнализации***. Для этого используются библиотечные функции[46] `openlog`, `syslog` и `closelog`:

```
void openlog(const char *ident, int option, int facility);
void syslog(int priority, const char *format, ...);
void closelog(void);
```

Чтобы начать работу с системой журнализации, программа вызывает `openlog`, передавая первым параметром своё название или иную идентифицирующую строку, указывая некоторые дополнительные опции в параметре `options` (в большинстве случаев достаточно передать число 0) и указывая, к какой подсистеме относится данная программа, через параметр `facility`. Наиболее популярные подсистемы, такие как почтовый сервер, имеют специальные значения этого параметра, прочим программам следует использовать константу `LOG_USER`.

Функция `syslog` похожа на функцию `printf`. Первым параметром указывается *степень важности* сообщения; например, `LOG_ERR` используется при возникновении неустранимой ошибки, `LOG_WARN` — для предупреждений, `LOG_INFO` — для простых информационных сообщений. Системный администратор может настроить систему журнализации так, чтобы в файлы журналов попадали только сообщения с уровнем важности не ниже определённого. Второй параметр — это форматная строка, аналогичная используемой в функции `printf`. Например, вызов функции `syslog` может выглядеть так:

```
syslog(LOG_INFO, "Daemon started, pid == %d", getpid());
```

Функция `closelog` завершает работу с системой журнализации, закрывая открытые файлы и т. п.

Управление процессами-демонами может осуществляться через сигналы; так, многие демоны в ответ на сигнал `SIGHUP` перечитывают конфигурационные файлы и при необходимости меняют режим работы. В более сложных случаях возможны и другие схемы управления.

[46] Их реализация зависит от конкретной системы. Например, демон, отвечающий за системную журнализацию, может держать открытый общедоступный сокет, в который функции журнализации будут выдавать свои сообщения.

Часть 6

Сети и протоколы

6.1. Компьютерные сети как явление

Строго говоря, ***компьютерной сетью*** считается любое соединение компьютеров (хотя бы двух), позволяющее передавать между ними данные. В истории известны примеры, когда с некоторой натяжкой сетевым соединением считалась даже организованная на регулярной основе передача файлов через внешние носители — попросту говоря, дискеты. Компьютеры, участвовавшие в таком «сетевом взаимодействии», вообще не были друг с другом соединены, но при этом пользователи этих «сетей» посылали друг другу электронную почту, участвовали в текстовых телеконференциях, обменивались файлами. Конечно, прибегать к обмену данными через дискеты приходилось только от недостатка технических возможностей по организации соединений между компьютерами; в наше время, когда цифровые сети связи доступны практически везде и всегда, пользователям нет нужды идти на подобные ухищрения.

6.1.1. Сети и сетевые соединения

Если рассмотреть все существующие сети связи, безотносительно компьютерного контекста, то среди них можно выделить сети с фиксированными соединениями, сети коммутации соединений и сети коммутации пакетов. Фиксированные соединения в наше время встречаются редко; примером такой сети можно считать системы громкой связи на вокзалах и в аэропортах. Распространённые до недавнего времени у военных полевые телефоны, где для соединения двух аппаратов прокладывался телефонный кабель, тоже представляют собой сеть связи с фиксированным соединением. Суть этого вида сетей связи в том, что узлы соединяются друг с другом каналами, схема включения которых не предусматривает динамического изменения; любые поправки можно

внести только вручную. Каждый абонент в такой сети постоянно связан с одним или несколькими другими, всегда одними и теми же.

Более гибкое устройство имеют сети коммутации соединений; хрестоматийный пример такой сети — хорошо известная нам сеть проводных («городских») телефонов. В сети коммутации соединений все конечные абоненты подключены к одному коммутатору либо (чаще) к одному из многих соединённых между собой коммутаторов, образующих *коммутирующую среду*. Посылая управляющие команды коммутаторам, абоненты таких сетей могут по своему усмотрению устанавливать соединения между собой и разрывать их.

В компьютерных сетях используется совершенно иной принцип работы — ***коммутация пакетов***. Следует отметить, что коммутация пакетов возможна только в сетях, рассчитанных на передачу цифровых данных. Информация, которую нужно передать в сеть, делится на ***пакеты***[1], каждый пакет снабжается адресом получателя, после чего оборудование, составляющее сеть связи, передаёт пакет нужному абоненту. По одному каналу связи могут за одну секунду пройти пакеты, адресованные миллионам разных участников сетевого взаимодействия. Внешне сеть коммутации пакетов напоминает сеть с фиксированными соединениями, поскольку физически каналы связи никто не переключает, но суть её, как видим, совершенно другая.

Если хорошо поискать, то можно найти действующие компьютерные сети, относящиеся как к сетям коммутации соединений, так и к сетям с фиксированными соединениями (и без всякой коммутации пакетов!), но рассматривать такие сети мы не будем: их области применения настолько специфичны, что вы вряд ли когда-нибудь с ними столкнётесь, а если всё-таки столкнётесь, то вам в любом случае придётся изучать документацию на конкретную сеть. Итак, **в компьютерных сетях, которые мы будем рассматривать, вся информация передаётся небольшими порциями (пакетами данных); каждый такой пакет содержит идентификатор (адрес) получателя, для которого он предназначен**; оборудование и программное обеспечение компьютерных сетей обеспечивает передачу каждого пакета по назначению, хотя и не даёт гарантии успеха такой передачи.

Существует множество разнообразных способов соединения компьютеров и всевозможных технологий передачи цифровой информации между ними. Чаще всего данные передаются по проводам в виде электрических сигналов, либо в виде радиосигналов — без всяких проводов; некоторые авторы заявляют, что наличие или отсутствие «материаль-

[1] Английский оригинал этого термина — *packet*. Не следует путать термин «пакет», используемый в контексте передачи данных по сети, с термином, звучащим по-русски точно так же, но относящимся к типу планирования времени центрального процессора в многозадачных операционных системах; там оригинальный английский термин — *batch* (буквально «колода»).

ного» соединения (провода, кабеля и т. п.) представляет собой главный признак, по которому следует классифицировать сети. Мы позволим себе с этим не согласиться, ведь, например, технология передачи данных *световыми* сигналами через оптоволоконный кабель отличается от любой из множества технологий, подразумевающих электрические провода, ничуть не меньше, чем они, в свою очередь, отличаются от соединений через радиоволны; беспроводные соединения тоже не всегда основаны именно на радиоволнах, существуют методы передачи информации через инфракрасное излучение, через сфокусированные световые лучи и так далее. Исследуя вопрос об электрических соединениях, мы обнаружим широчайший ассортимент подходов, различающихся количеством проводов (от двух до нескольких десятков), применяемыми способами модуляции (представления цифровой информации электрическими сигналами), методами синхронизации, допустимыми типами кабелей и их длиной. Попытавшись вникнуть в построение радиоканалов для передачи компьютерной информации, мы и там обнаружим существенное технологическое разнообразие.

Самое интересное, что разнообразие обитателей всего этого зоопарка нас может не волновать: не всё ли равно, как именно осуществляется передача данных с одного компьютера на другой? Пользователи мобильных устройств иногда забывают, каким из двух способов — через сотовую сеть или через WiFi — они в настоящий момент соединены с Интернетом; пользователь домашнего фиксированного канала вполне может не иметь никакого понятия о разнице между ADSL, Ethernet и прочими видами физического соединения «последней мили».

Конечно, разные способы соединения могут обладать особенностями, которые придётся учитывать; так, если все компьютеры в вашей квартире или офисе соединены проводами, вы можете более-менее достоверно предполагать, что кто попало к вашей сети не подключится, и, например, разрешить сетевому принтеру принимать задания на печать от любых компьютеров, находящихся с ним в одном сегменте. С WiFi такой номер не проходит, даже если сеть защищена паролем: как правило, пароль там один на всех, то есть его заведомо знает больше одного человека, а раз так — можете считать, что пароль вообще не представляет собой никакого секрета, так что потенциально к вашей сетке может «присосаться» кто угодно, проходивший мимо по коридору или сидящий в соседнем помещении, а в некоторых случаях даже прохожий с улицы. В такой ситуации доступ к принтеру придётся как-то ограничить, если вы не хотите однажды обнаружить на полу груду бумаги, на которой напечатано непонятно что, а в принтере — пустой картридж. Но в целом разницу между технологиями физического уровня трудно считать определяющей для целей классификации сетей.

В учебных пособиях можно встретить долгие рассуждения о возможных видах *топологии* компьютерной сети; обычно при этом рассказывают про «шину», «звезду», «кольцо», «соединение каждого с каждым» и — в качестве белого флага — «смешанную топологию»; последнюю, как правило, описывают некими наукообразными эпитетами,

смысл которых сводится к сакраментальному «как попало». Отметим, что топология «кольцо», непременно упоминаемая в таких случаях, в реальности не встречается, как и соединение каждого с каждым. Ещё интереснее обстоят дела с «шинами» и «звёздами»: одна и та же сеть может оказаться «звездой», если нарисовать схему соединения проводов, но при этом «шиной», если рассматривать логику взаимодействия компьютеров.

В первой половине 1990-х годов самым популярным способом соединения компьютеров можно было считать Ethernet, основанный на ***коаксиальном кабеле***. В таком кабеле предусмотрены два проводника — проходящий по центру *внутренний* и отделённая от него слоем изоляции *оплётка*, служащая одновременно вторым проводом и экраном, предотвращающим потери на излучение и защищающим от электромагнитных помех. Коаксиальные кабели применяются в технике для передачи высокочастотных электрических сигналов — например, для подключения антенн к приёмопередающим устройствам; если у вас дома есть телевизор, принимающий эфирные телеканалы, то антенна к нему подведена коаксиальным кабелем, хотя и не такой конструкции, как у кабелей, применявшихся для соединения компьютеров. В области компьютерных сетей коаксиальный кабель практически вышел из употребления, но произошло это не так давно: в первой половине 2000-х годов сети на коаксиальном кабеле ещё довольно часто встречались.

Сетевые интерфейсы компьютеров подключались к коаксиальному кабелю параллельно с помощью специальных Т-образных соединителей, так называемых *Т-коннекторов*, так что сеть, основанная на коаксиальном кабеле, представляла собой *шину* в чистом виде. Скорость передачи данных по такой сети составляла 10 Мбит в секунду, что по современным меркам не так много, но основной недостаток коаксиальной сети состоял не в абсолютном значении скорости, а в неизбежности *коллизий*, когда один из участников взаимодействия начинает передачу данных, а другой участник, не успев на это среагировать, тоже начинает передачу и портит пакет данных, передаваемый первым участником. Чем плотнее загружена сеть, организованная в виде шины, тем выше вероятность коллизий и тем бóльшая часть пропускной способности сети на этих коллизиях теряется.

Был у коаксиального соединения и другой серьёзный недостаток. Коаксиальный кабель должен был представлять собой замкнутую среду передачи сигнала, завершающуюся с обеих сторон *терминаторами* — специальными насадками, в которых стояли резисторы по 50 Ом. Без терминатора сигнал, проходящий по кабелю, отражается от его оконечности; отражённый сигнал накладывается на основной сигнал и искажает его, так что сеть теряет работоспособность. Нетрудно догадаться, что такая сеть перестанет работать при любом обрыве кабеля или разъединении коннекторов, ведь в месте разрыва терминатору

Рис. 6.1. Коаксиальные соединения: кабель с соединителем, Т-коннектор, терминатор, узел для подключения компьютера в сборе

взяться неоткуда. По закону подлости обрывы и разъединения чаще всего происходили в запертой комнате, единственный ключ от которой находится у сотрудника, уехавшего в командировку. При этом нельзя было защитить кабель от случайных нарушений целостности, убрав его куда-нибудь в стену, короб и т. п., ведь каждый компьютер нужно было подключить непосредственно к кабелю[2]; иначе говоря, кабель приходилось прокладывать так, чтобы он проходил через каждое из рабочих мест, подключённых к сети, и окрестности всех этих рабочих мест неизбежно оказывались зоной риска.

Естественно, столь неудобному способу соединения машин между собой специалисты довольно быстро нашли замену — так называемую ***витую пару***. Связь здесь производится по четырём проводам, попарно перевитым между собой для избавления от электромагнитных помех (отсюда название); по одной паре происходит передача, по другой — приём информации. Два компьютера можно соединить кабелем витой пары, подключив пары, используемые для приёма и передачи, крест-накрест; по-английски кабель, специально предназначенный для такого соединения, называется *crossover*. Когда компьютеров больше двух, ситуация становится существенно более сложной: необходимо специальное устройство, которое по-английски называется *hub*; русскоговорящие специалисты обычно называли это устройство попросту «хаб», используя прямую транслитерацию английского термина, но такое именование следует отнести к разряду профессионального жаргона; имеется устоявшийся официальный перевод термина *hub* на русский язык — ***концентратор***. Каждый компьютер локальной сети при таком способе её построения соединяется с концентратором (хабом) отдельным кабелем, подключаемым к так называемому *порту*; встречались концентраторы, имеющие от 3 до 24 портов. Сигнал, получаемый от любого из компьютеров (или, правильнее говоря, на любом из своих портов) по проводам, ответственным за передачу, концентратор транслирует

[2]Спецификация коаксиального Ethernet'а допускала подключение сетевого интерфейса к основному кабелю с помощью отводного провода, но длина его не могла превышать 4 см.; очевидно, это не решало никаких проблем.

Рис. 6.2. Витая пара: кабель, концентратор, коммутатор

всем остальным компьютерам, подключённым к нему (на все остальные порты), но уже через провода, ответственные за приём.

В наше время концентраторы считаются устаревшим видом оборудования, вместо них используются ***коммутаторы*** (англ. *switch*; чаще всего для обозначения этих устройств в русском языке используется жаргонизм «свитч»). Коммутатор отличается от концентратора тем, что транслирует получаемый пакет не всем компьютерам сети, а только тем из них, которым этот пакет предназначен (то есть чаще всего — только одному компьютеру). Это естественным образом снижает нагрузку на сеть: например, если в сети одновременно обмениваются информацией две пары компьютеров, то при использовании концентратора им придётся делить между собой общую пропускную способность сети, тогда как при работе через коммутатор компьютеры одной пары могут вообще не обращать внимания на существование другой пары.

Конечно, коммутатор технически представляет собой устройство гораздо более сложное, нежели концентратор, ведь ему приходится анализировать содержимое каждого передаваемого по сети пакета, чтобы узнать, кому этот пакет предназначается, и плюс к тому нужно помнить, к какому порту подключён какой компьютер; только так можно понять, через какой порт транслировать каждый отдельно взятый пакет. Следует также принять во внимание, что и концентраторы, и коммутаторы можно соединять между собой для получения сетей большого размера; с точки зрения коммутатора это значит, что к одному порту может быть подсоединён не один компьютер, а (в общем случае) произвольное их число. Всё это приводит к необходимости использования процессора, памяти и программы, управляющей работой коммутатора, то есть технически коммутатор — это компьютер специального назначения. Наиболее «продвинутые» коммутаторы имеют в сети свой собственный адрес, как у обычного компьютера, и позволяют в режиме удалённого доступа управлять своей работой — например, задавать явным образом, какие компьютеры могут подключаться к отдельным портам, разбивать множество портов на группы, логически представляющие собой отдельные сети (англ. *VLAN*, *virtual local area network*; устоявшегося русского перевода пока нет) и т. п.

Благодаря совершенствованию производственных технологий в наше время простейшие коммутаторы — неуправляемые и имеющие обычно от 4 до 8 портов — стоят достаточно дёшево[3], что делает полностью бессмысленным использование концентраторов; дело в том, что корпус, разъёмы портов, блок питания, основная монтажная плата нужны как в коммутаторе, так и в концентраторе, а разница в сложности электроники (технически очень высокая, ведь концентратору не нужен ни процессор, ни память, ни программа) с производственной точки зрения почти не даёт различия в себестоимости изделия. Никто не отдаст предпочтения концентратору ради двух-трёх процентов денежной экономии.

В те годы, когда сети на основе витой пары только начинали развиваться, ситуация выглядела совершенно иначе. Идея коммутатора возникла, естественно, практически сразу в силу своей очевидности (даже раньше перехода с коаксиального кабеля на витую пару; известны коммутаторы для коаксиальных сегментов), но цены на первые такие устройства были совершенно заоблачными. Даже концентраторы («хабы») вплоть до конца 1990-х годов стоили достаточно дорого и вытеснить коаксиальный кабель попросту не могли: несмотря на все неудобства эксплуатации сетей на коаксиале, концентратор для таких сетей не требуется, и это обстоятельство оставалось во многих случаях определяющим.

Возвращаясь к вопросу о классификации сетей, отметим один интересный момент. Сети на основе коаксиального кабеля очевидным образом относятся к топологии «шина»; при переходе к сетям на витой паре, использующим концентраторы, топология физических соединений превращается в «звезду»: имеется центральный узел (концентратор), к которому подключены все остальные (в данном случае компьютеры). Но с точки зрения передачи информации при этом *ничего не меняется*, то есть, если можно так выразиться, «логическая» топология сети так и остаётся шиной! В самом деле, при передаче в сеть пакета данных этот пакет через концентратор получают *все* остальные абоненты сети — точно так же, как это происходило в коаксиальной сети.

Неизменность «логической топологии» сети при переходе от коаксиальной шины к витой паре с концентратором лучше всего иллюстрирует такой факт. Существовало довольно много сетевых карт (печатных плат, вставляемых в компьютер и обеспечивающих аппаратное подключение компьютера к сети), имевших одновременно разъёмы для подсоединения витой пары и коаксиального кабеля (рис. 6.3). Зачастую администратор сети, устав от постоянных неполадок коаксиального кабеля, покупал концентратор и переводил свою сеть на витую пару. Для этого было достаточно переключить сетевые карты на работу с портом витой пары

[3]На момент написания этого текста в 2017 году стоимость самого дешёвого коммутатора составляла около 500 рублей, тогда как каждый из уже вышедших томов нашей книги продавался более чем за 1000 р.

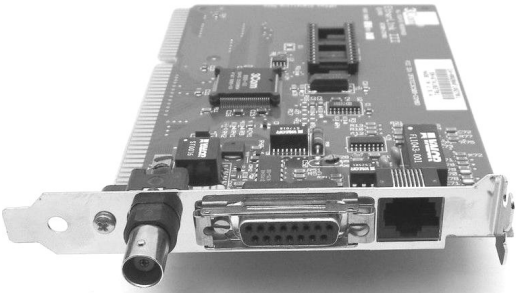

Рис. 6.3. Сетевая карта 3c509 (3COM) с разъёмами для подключения коаксиального кабеля (слева), внешнего адаптера (в центре) и кабеля витой пары (справа).

вместо коаксиального порта и заменить провода, а *программное обеспечение компьютеров никакой перенастройки не требовало*, то есть программы, использующие выход в сеть, вообще не замечали смены физического носителя соединения.

С учётом всего сказанного можно заключить, что топология физического соединения — это тоже совершенно не тот признак, который должен нас волновать при попытках классифицировать сети. Если цель, стоящая перед нами — научиться писать программы, то в нашу область интересов входят скорее такие особенности сетей, которые наша программа может обнаружить и которые, следовательно, необходимо учитывать при программировании.

Лет двадцать назад нас ещё могла бы с этой точки зрения заинтересовать разница между локальными, глобальными и прочими сетями в зависимости от их размера. Сейчас, когда фактически единственным используемым ***стеком протоколов***[4] остался TCP/IP, на котором основана работа сети Интернет, в большинстве случаев программам, работающим через сеть, оказывается безразлично, взаимодействуют ли они с компьютером, находящимся в другом полушарии Земли, в соседней комнате, на соседнем столе или даже их партнёр — другая программа, работающая на том же самом компьютере.

С повсеместным проникновением Интернета становится не всегда понятно, где кончается одна сеть и начинается другая; любая локальная сеть, если она имеет выход в Интернет, в определённом смысле представляет собой часть Интернета, причём в большинстве случаев можно выделить ещё несколько уровней, на каждом из которых можно указать

[4]Будучи центральным в области компьютерных сетей, понятие протокола вызывает неожиданные сложности у многих новичков. Мы подробно рассмотрим это понятие чуть позже, а пока ограничимся замечанием, что *протокол* — это некий набор соглашений, которым следуют участники сетевого взаимодействия, чтобы понять друг друга.

компьютерную сеть, в которую входят составными частями другие сети и которая сама по себе является частью более крупной сети. Один из основных критериев отнесения компьютеров к единой сети базируется не на технических, а на организационных критериях: отдельной сетью считаются компьютеры, имеющие между собой соединения для передачи данных и при этом находящиеся под единой политикой управления, то есть, проще говоря, эксплуатируемые одной командой системных администраторов. Отметим, что, несмотря на нетехнический характер этого критерия, нам как будущим программистам он весьма интересен, поскольку программы, работающие внутри сети, и программы, работающие за её пределами, оказываются с нашей точки зрения в существенно различных категориях: первым в некоторых случаях можно доверять (хотя и с ограничениями), вторым же доверять a priori нельзя, ведь мы не знаем и не можем знать, кто и с какой целью их написал и запустил. Далеко не все пользователи компьютерных сетей соответствуют нашим представлениям об ангелах с крылышками. Конечно, злоумышленник может оказаться и внутри периметра нашей сети, и такую возможность тоже приходится учитывать, но это всё же скорее чрезвычайное происшествие, тогда как наличие злоумышленников в Интернете «по ту сторону» нашего периметра — это обычная рабочая обстановка.

Возвращаясь к техническим аспектам построения компьютерных сетей, отметим, что простейшая (наименьшая) компьютерная сеть — это такая сеть, в которой имеется всего одно сетевое соединение; такую сеть при всём желании не получится разделить на сети меньшего размера. И именно здесь мы неожиданно натыкаемся на самый, пожалуй, интересный (с точки зрения программистов и сетевых администраторов) критерий классификации, относящийся, правда, не к сетям как таковым, а как раз к отдельно взятым сетевым соединениям. Дело в том, что, как мы уже видели, одно сетевое соединение может быть рассчитано на достаточно большое количество компьютеров — во всяком случае, более чем на два. Именно так работали и работают сети Ethernet, и совершенно не важно, используется ли для соединения компьютеров коаксиальный кабель (теоретически можно себе представить, что где-то коаксиальные сети ещё работают) или витая пара, применяются ли концентраторы или коммутаторы. Любые два компьютера, подключённые к одной локальной сети Ethernet, могут устанавливать между собой сетевые соединения и обмениваться информацией, не прибегая к помощи посредников; в сетях на коаксиальном кабеле такие посредники действительно отсутствовали, в сетях на витой паре посредники, формально говоря, есть — это так называемое активное оборудование в виде «хабов» или «свитчей», но участники сетевого взаимодействия их не видят и могут никак не учитывать их существование.

С другой стороны, достаточно часто нам приходится иметь дело с сетевыми соединениями, в которых задействовано только два ком-

пьютера. Такое соединение можно построить, естественно, на основе технологий семейства Ethernet (выше мы уже упоминали вариант соединения двух компьютеров кабелем витой пары без использования активного оборудования), но чаще такие соединения возникают при использовании технологий, по самой своей сути не предполагающих появления «третьего лишнего». Именно так в большинстве случаев организованы «нелокальные» каналы передачи данных; иначе говоря, если вы рассмотрите любой цифровой канал, выходящий за пределы отдельного здания (например, канал, связывающий ваш компьютер или вашу локальную сеть с провайдером доступа в Интернет), то с хорошей вероятностью этот канал окажется соединением вида «точка-точка» — именно так называют сетевые соединения «на двоих» (соответствующий англоязычный термин *point-to-point* обычно сокращают до *PtP*).

Различие между двухточечным и многоточечным вариантами сетевого соединения оказывается неожиданно принципиальным. При передаче данных через соединение вида «точка-точка» сразу понятно, кому предназначены передаваемые данные; но если данные передаются через сетевой интерфейс, к которому непосредственно подключено (кроме нас самих) больше одного потенциального получателя информации, нам приходится уточнять, кому конкретно предназначен тот или иной пакет данных. Как мы увидим позже, признаки, по которым отдельные сетевые интерфейсы идентифицируют друг друга при работе через многоточечное соединение, отличаются от адресов, используемых взаимодействующими по сети программами для идентификации своих партнёров по соединениям; для установления соответствия одних адресов другим в многоточечных соединениях используется специальный протокол, в котором нет надобности при работе через соединение «точка-точка».

Отметим ещё один момент. Мы уже встречались с понятием *сетевой карты*; так вот, строго говоря, её наличие не обязательно для работы с компьютерной сетью, поскольку передачу данных между двумя компьютерами можно организовать через имеющиеся порты едва ли не любого вида. Так, в предыдущей части книги при обсуждении терминалов мы упоминали последовательные порты (COM-порты) и модемы для подключения к телефонным линиям; этот вид оборудования вышел из употребления не так давно, а в конце 1990-х и начале 2000-х годов модемное соединение оставалось основным способом доступа в Интернет для большинства пользователей. Если же два компьютера находятся физически недалеко друг от друга — например, в соседних комнатах,— то можно соединить их COM-порты шнуром из трёх проводов, который обычно называли «нуль-модемом», и организовать через него сетевое соединение, единственным недостатком которого будет низкая скорость передачи данных (как правило, не выше 115200 бод, что почти в сто раз медленнее старых типов Ethernet-сетей и в тысячу раз медленнее,

чем наиболее распространённые в наше время 100-мегабитные соединения). Существует программное обеспечение для организации сетевого соединения через порт USB, хотя при этом на одном из двух соединяемых компьютеров должен присутствовать «ведомый» порт, что для обычных компьютеров редкость, но совершенно не редкость, например, для планшетников. В те времена, когда оборудование Ethernet стоило дорого, но практически каждый компьютер был оснащён так называемым параллельным портом, использовавшимся для подключения принтера, можно было встретить сетевые соединения с использованием параллельных портов. Шина SCSI, часто встречающаяся на серверных компьютерах и в основном предназначенная для подключения внешних устройств хранения данных — жёстких дисков, CD, ленточных накопителей (стриммеров) и т. п. — физически позволяет связать два компьютера, и энтузиасты немедленно воспользовались этим для организации сетевого соединения (например, IP over SCSI).

Наконец, даже аппаратура, специально предназначенная для соединения с компьютерной сетью, будь то Ethernet или WiFi, в наше время далеко не всегда имеет вид карты. Большинство современных компьютеров имеет порт для подключения витой пары на материнской плате, если же говорить о ноутбуках и нетбуках, то там — опять-таки, на материнской плате — обычно имеется также и WiFi-адаптер. По правде говоря, со схематической точки зрения это всё те же сетевые карты, просто смонтированные непосредственно на основной плате, а не в виде платы-расширения; но бывают ведь ещё и адаптеры Ethernet и Wifi, подключаемые к USB-порту, и тут уже ни о какой «карте» речи не идёт.

С учётом всего сказанного явно требуется какой-то другой термин, и мы его сейчас введём. **Аппаратное устройство (в составе компьютера или подключённое к нему), используемое для соединения с одним или несколькими другими компьютерами для передачи данных, обычно называют *сетевым интерфейсом*.**

Сразу же предостережём читателя от неправильного восприятия сказанного. Фразу, выделенную в предыдущем абзаце, не следует воспринимать как *определение* сетевого интерфейса, поскольку во многих контекстах тот же термин имеет несколько иной смысл. Так, часто встречаются *виртуальные сетевые интерфейсы*, которые с точки зрения программного обеспечения практически ничем не отличаются от обычных (физических), но при этом в физическом смысле не существуют; такие интерфейсы возникают при создании *туннелей* и *виртуальных частных сетей* (VPN'ов, от английского *virtual private network*). Кроме того, один физический сетевой интерфейс может в некоторых случаях быть настроен как *транковый*; в этом случае с программной точки зрения мы увидим не один интерфейс, а несколько, каждый из которых

участвует в своём сетевом соединении[5], хотя с аппаратной точки зрения интерфейс останется один. Как обычно, наша задача сейчас не в том, чтобы дать строгое определение термина, а в том, чтобы понять, о чём идёт речь.

6.1.2. Шлюзы и маршрутизация

Отдельное сетевое соединение может объединять несколько десятков, а иногда даже пару сотен компьютеров, но наращивать это количество дальше оказывается нецелесообразно, причём как технически, так и организационно. В частности, с технической точки зрения коммутаторы, образующие одну локальную сеть, не допускают циклических соединений, то есть от каждого компьютера к каждому другому в графе соединений должен быть ровно один путь; это значит, что выход из строя любого элемента инфраструктуры нарушит работу сети, а обеспечивать бесперебойное функционирование кабелей (в том числе соединяющих между собой коммутаторы) тем труднее, чем больше их общая длина. Кроме того, чем больше к сети подключено компьютеров, тем больше в ней должно быть коммутаторов и тем труднее спроектировать сеть так, чтобы бóльшая часть информации передавалась между компьютерами, соединёнными одним коммутатором; как следствие, с ростом числа компьютеров в одной сети растёт и нагрузка на кабели, соединяющие между собой отдельные коммутаторы, так что рано или поздно их пропускной способности становится недостаточно. Наконец, в едином сетевом соединении в определённых случаях неизбежно использование широковещательных сообщений — таких, которые должен получить и как минимум проанализировать каждый из компьютеров; чем больше компьютеров в локальной сети, тем больше будет таких пакетов.

С организационной точки зрения слишком большие локальные сети тоже порождают определённые проблемы. В рамках одного сетевого соединения, как правило, невозможно ввести никакие ограничения по видам и количеству информации, передаваемой между компьютерами: все доступны всем без ограничений. Из этого вытекает необходимость мер по недопущению в сеть злоумышленников, и чем сеть больше, тем эти меры сложнее; если в одну локальную сеть (одно сетевое соединение) объединить компьютеры разных организаций, обеспечить безопасность в такой сети будет нереально.

[5] Такой режим работы интерфейса становится возможен благодаря использованию «продвинутых» коммутаторов, которые позволяют сгруппировать их порты в две или более виртуальные Ethernet-сети. Один из портов может быть объявлен как *транковый*, то есть обслуживающий несколько виртуальных интерфейсов в рамках одного физического, причём каждый из виртуальных интерфейсов оказывается «подключён» к своей виртуальной сети.

Обратим теперь внимание на то, что на отдельно взятый компьютер никто не мешает поставить больше одного сетевого интерфейса; как следствие, один и тот же компьютер может одновременно быть участником нескольких сетевых соединений. С такой ситуацией вполне можно столкнуться в сугубо бытовых условиях, ведь большинство современных ноутбуков имеют в своём составе как обычный сетевой интерфейс, рассчитанный на подключение витой пары, так и беспроводной адаптер для работы с WiFi-сетями. Конфигурация систем, предназначенных для конечного пользователя, обычно построена так, что из двух (и более) сетевых интерфейсов компьютер реально использует только один, даже когда из них активно больше одного, но такое ограничение не имеет никаких технических причин.

Ситуация становится гораздо интереснее, если компьютер, участвующий в нескольких сетевых соединениях одновременно, настроить так, чтобы он мог перебрасывать между ними пакеты. Такой компьютер обычно называют ***сетевым шлюзом*** или ***маршрутизатором***; часто можно услышать жаргонное слово *роутер*, транслитерированное с английского *router*[6]. Полезно помнить, что английский оригинал словосочетания «сетевой шлюз» звучит как *network gateway*, что в разговорной речи часто превращается в краткое «*gate*» (буквально «ворота»; читается «гейт»). В настройках, имеющих отношение к компьютерным сетям, часто встречается аббревиатура «GW», означающая, как нетрудно догадаться, «gateway»; если в какой-нибудь сети вам попался компьютер, имеющий такое сетевое имя — скорее всего, именно через него происходит связь с внешним миром.

С хорошей степенью вероятности читатель уже сталкивался с маршрутизаторами. Если ваш дом или квартира подключены к Интернету, то в наши дни в большинстве случаев линия связи, которую к вам привёл провайдер, подключается к устройству, которое затем «раздаёт» подключение к Интернету через WiFi, а при желании позволяет также подключить несколько компьютеров проводами. Это невзрачное устройство, выглядящее как небольшая коробочка, как раз и есть маршрутизатор. Пусть вас не обманывает его скромный вид: внутри это полноценный компьютер со своей операционной системой, причём в роли этой системы довольно часто выступает Linux, хотя со стороны этого может быть и не видно.

Будучи частью нескольких сетей одновременно, компьютер-шлюз служит своеобразным «пропускным пунктом» для пакетов, направляющихся из одной сети в другую (отсюда название). В простейшем случае шлюз просто принимает из одной сети пакеты, адресованные компьютерам другой сети, и передаёт их по назначению; конечно, при этом все компьютеры в сетях, имеющих соединение через шлюз, должны знать о существовании компьютеров, доступных через шлюз, а также и

[6] Буквально слово *route* означает «маршрут»; поскольку словоформа «маршрутер» по-русски звучит несколько диковато, слово «маршрутизатор» следует считать буквальным переводом английского слова *router*.

о самом шлюзе. В реальной жизни большинство шлюзов не ограничивается простым перебрасыванием пакетов туда-сюда: на компьютере, выполняющем роль шлюза, можно настроить разнообразные фильтры, пропускающие только часть пакетов, можно ограничить количество передаваемых пакетов, можно организовать всевозможный мониторинг того, какая информация в какое время проходила через шлюз, и т. д. Достаточно часто шлюзы даже изменяют содержащуюся в пакетах адресную информацию. Кроме того, в одном сетевом соединении могут участвовать несколько шлюзов, так что пакетам из одной сети, чтобы достичь другой сети, придётся совершить путешествие от шлюза к шлюзу, прежде чем они попадут куда нужно. Схемы передачи пакетов через шлюзы как раз и называют «маршрутами», отсюда второе название для шлюзов — «маршрутизаторы».

Попробуем проиллюстрировать сказанное. Пусть в некоем доме (представьте себе для наглядности, что это коттедж в коттеджном посёлке) живут Анна, Борис и Вера; их родственные отношения нас совершенно не волнуют, так что можете сами придумать, кем они друг другу приходятся. У каждого из них есть свой компьютер, причём эти компьютеры соединены в локальную сеть — например, витой парой через коммутатор. В другом доме того же посёлка живут Геннадий, Дмитрий, Елена и Жанна, у них тоже есть компьютеры, и эти компьютеры объединены в свою локальную сеть.

В некий момент Борис и Дмитрий решают, что им было бы интересно связать свои компьютеры, например, чтобы играть в многопользовательские игры. Кроме того, Борис вспоминает, что у Геннадия есть большая коллекция музыки и фильмов в виде файлового архива и Геннадий с удовольствием делится своими файлами со всеми знакомыми[7], но при отсутствии связи между сетями таскать файлы приходится на флешках, что не очень удобно. Жанна, узнав о планах Бориса и Дмитрия, заявляет, что Вера — её лучшая подруга и они хотят общаться между собой, используя микрофоны и видеокамеры. После этого становится очевидно, что обитателям обоих домов просто необходима связь между их локальными сетями.

Если бы все компьютеры находились в одном здании, можно было бы просто соединить проводом коммутаторы и, возможно, поменять сетевые адреса некоторых компьютеров, чтобы исключить совпадения; в результате вместо двух разных локальных сетей наши персонажи обнаружили бы себя в одной большой сети. Но тут в дело вступают вполне

[7]Если у вас на этом месте возникли какие-нибудь странные мысли, связанные с так называемым авторским правом, вам стоит обратить внимание, что (во всяком случае, на момент написания этого текста) в России создание копий в цифровой форме для личного использования является абсолютно законным, если только копируемые файлы не являются программами для ЭВМ; см. ст. 1273 ГК РФ. Кроме того, настоятельно рекомендуем вернуться к первому тому нашей книги и перечитать самое первое («философское») предисловие.

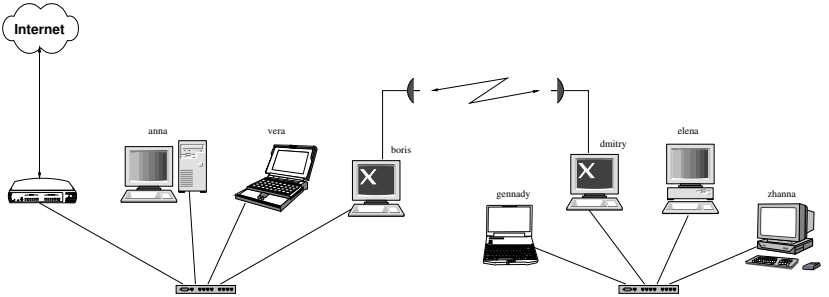

Рис. 6.4. Пример компьютерной сети

ожидаемые технические и организационные соображения. С технической точки зрения кабель витой пары по улице лучше не прокладывать: при первой же грозе есть риск лишиться компьютеров, да и чисто механически протянуть провод не всегда возможно — например, если дома расположены не рядом, а через улицу или между ними находятся чужие участки. Объединению сетей могут помешать и сугубо организационные причины: например, Елена когда-то успела в дым рассориться с Анной и заявляет, что не желает позволять «всей её семейке» даже потенциально иметь какой бы то ни было доступ к своему компьютеру.

Взвесив все имеющиеся соображения, Борис и Дмитрий решают купить пару узконаправленных антенн, установить сетевые интерфейсы для WiFi и создать между своими компьютерами радиоканал, а сами компьютеры настроить как шлюзы: компьютер Бориса будет передавать на компьютер Дмитрия через радиоканал пакеты, адресованные компьютерам самого Дмитрия, а также Геннадия, Елены и Жанны, тогда как компьютер Дмитрия будет передавать по радиоканалу на компьютер Бориса пакеты, предзназначенные для компьютеров Анны, Веры и самого Бориса.

После этого наши друзья приступают к настройке компьютеров своих домочадцев. На компьютерах Анны и Веры указывается, что пакеты для компьютеров Геннадия, Дмитрия, Елены и Жанны нужно передавать через компьютер Бориса; компьютеры Геннадия и Жанны настраиваются на передачу пакетов для компьютеров Анны, Бориса и Веры через компьютер Дмитрия. Елена от аналогичной настройки наотрез отказывается; следуя её настойчивым требованиям, Дмитрий на своём компьютере устанавливает специальное правило для шлюза, чтобы пакеты, пришедшие извне (через сетевой интерфейс радиоканала) и адресованные компьютеру Елены, сбрасывались, никуда не передаваясь. После этого Геннадий у себя на компьютере запускает сервер игры Battle For Wesnoth, и наши персонажи — все, кроме Елены — проводят вечер, разыгрывая сражения между эльфами и орками; Елена могла бы принять участие в игре, поскольку сервер находится в одной

сети с её компьютером, но она играть не хочет и вообще недовольна происходящим.

До сих пор мы никак не упоминали связь с Интернетом; предположим, до сего момента компьютеры наших друзей вообще не имели выхода в Интернет — хотя, конечно, такое предположение в наши дни выглядит несколько странно, но в качестве иллюстрации наш пример более интересен, если всю инфраструктуру приходится строить именно «с нуля». Пусть теперь Борис заключает договор с провайдером и к нему в дом вводят канал для связи с Интернетом, который, как водится, заканчивается маршрутизатором. Здесь можно не изобретать лишних сущностей, а просто подключить этот маршрутизатор к уже имеющемуся в доме коммутатору, сделав его ещё одним (четвёртым) компьютером в локальной сети. На компьютерах Анны, Бориса и Веры добавляется ещё одно маршрутное указание: пакеты, предназначенные для «всех остальных» компьютеров (читай — для всех компьютеров Интернета, за исключением тех, про которые есть другие указания) направлять на новый маршрутизатор. Такое правило маршрутизации называется «шлюз по умолчанию». К примеру, на компьютерах Анны и Веры теперь по три правила маршрутизации: пакеты для других компьютеров локальной сети отдавать в локальное соединение (без указания шлюза), пакеты для локальной сети соседей отдавать на компьютер Бориса, а пакеты для всех остальных компьютеров — на маршрутизатор от провайдера.

Дмитрий предлагает Борису поучаствовать в оплате интернет-канала, после чего наши друзья добавляют несложные настройки: на компьютере Дмитрия шлюзом по умолчанию объявляется компьютер Бориса, на компьютерах Геннадия, Елены и Жанны — компьютер Дмитрия. Кстати, ранее прописанное у Геннадия и Жанны правило насчёт соседской локальной сети теперь можно убрать, ведь для них шлюзом выступает тот же самый компьютер Дмитрия. А вот с компьютером Елены имеется проблема: ограничение, которое по её просьбе ввёл Дмитрий, не позволяет ей выходить в Интернет, ведь пакеты, предназначенные для её компьютера, фильтруются. Подумав, Дмитрий модифицирует фильтр так, чтобы из пакетов, адресованных компьютеру Елены, фильтровались только те, в которых обратным адресом обозначен какой-нибудь из адресов локальной сети соседей.

Для нашего примера возможны и более сложные варианты. Например, Дмитрий тоже мог бы заключить договор с провайдером (желательно — с другим, не тем, что у Бориса) и иметь свой собственный канал в Интернет, но при этом наши друзья могли бы настроить маршрутизацию так, чтобы при выходе из строя одного из каналов обе локальные сети начинали бы использовать то соединение с Интернетом, которое осталось рабочим; кроме того, в какой-то момент обоим «администам-любителям» могло бы прийти в голову, что использовать в

качестве маршрутизаторов личные компьютеры не очень удобно (например, их приходится всегда держать включёнными и нельзя лишний раз перезагрузить), так что в обеих локальных сетях появились бы выделенные маршрутизаторы — либо специализированные, либо в виде обычных компьютеров, на которые в таком случае можно возложить функции серверов, и так далее. Позже мы подробно разберём, как выглядят сетевые адреса компьютеров (точнее, их сетевых интерфейсов) при использовании протоколов TCP/IP и как могли бы выглядеть адреса и маршрутные правила на компьютерах из нашего примера. Узнаем мы и о том, что́ обычно происходит внутри маршрутизатора, соединяющего локальную сеть с Интернетом, почему компьютеры локальной сети из Интернета не видны и как получается, что локальные сетевые адреса не нужно менять при переходе на другой интернет-канал; но всему своё время.

6.2. Сетевые протоколы

6.2.1. Понятие протокола и модель OSI

Под ***протоколом обмена*** (или, для краткости, просто «протоколом») понимается набор соглашений, которым должны следовать участники обмена информацией[8], чтобы понять друг друга. При любом осмысленном взаимодействии по компьютерной сети задействуется сразу несколько протоколов, относящихся к разным *уровням*. Так, сетевая карта, через которую наш компьютер подключён к локальной сети, следует протоколу, фиксирующему правила перевода цифровых данных в аналоговый сигнал, передающийся по проводам, и обратно. Одновременно запущенный нами браузер связывается с сайтом в сети Internet, используя транспортный протокол TCP. Сервер и браузер обмениваются информацией, используя протокол HTTP (*hypertext transfer protocol*).

Понятие протокола почему-то вызывает у значительного числа новичков неожиданные сложности, и мы попробуем ещё раз заострить внимание на том, что́ же такое «протокол». Говоря, что протокол есть набор соглашений, мы имеем в виду *буквально* именно это. Иначе говоря, протокол — это некий *текст*, написанный на естественном языке (в применении к компьютерам язык обычно оказывается английским). Этот текст содержит указания для разработчиков программного обеспечения, а в некоторых случаях — и аппаратуры: *что нужно сделать, чтобы разработанные вами программы или устройства могли взаимодей-*

[8] Это не обязательно должны быть компьютеры. Скажем, азбука Морзе также является своего рода протоколом; более сложный пример некомпьютерного протокола — правила радиообмена между пилотами самолетов и авиадиспетчерами.

ствовать с другими программами/устройствами, декларирующими поддержку данного протокола.

Существует стандартная модель (ISO OSI), предполагающая разделение всех сетевых протоколов на семь уровней. ISO расшифровывается как International Standard Organization (организация, утвердившая соответствующий стандарт), OSI означает Open Systems Interconnection (буквально переводится как «взаимосоединение открытых систем», но обычно при переводе используется слово «взаимодействие»). Модель включает семь уровней:

- **физический** — соглашения об использовании физического соединения между машинами, включая количество проводов в кабеле, частоту и другие характеристики сигнала и т. п.;
- **канальный** (в оригинале *datalink*) — соглашения о том, как будет использоваться физическая среда для передачи данных, включая, например, размеры пакетов и способы коррекции ошибок;
- **сетевой** — соглашения о том, как данные будут передаваться по сети от шлюза к шлюзу; именно на этом уровне определяется, как выглядят сетевые адреса и как настраивается маршрутизация;
- **транспортный**; пакеты, передаваемые по сети с помощью протоколов сетевого уровня, обычно ограничены в размерах и, кроме того, могут доставляться не в том порядке, в котором были отправлены, теряться или, наоборот, дублироваться (приходить в двух и более экземплярах); обычно прикладным программам требуется более высокий уровень сервиса, обеспечивающий надёжность доставки данных и простоту работы, и за это как раз отвечают протоколы транспортного уровня; реализующие их программы сами следят за доставкой пакетов, отправляя и анализируя соответствующие подтверждения, нумеруют пакеты и расставляют их в нужном порядке после получения;
- **сеансовый** — определяет порядок проведения сеанса связи, очерёдность запросов и т. п.;
- **представительный**; на этом уровне определяются правила представления данных, в частности, кодировка, способы представления двоичных данных текстом и т. п.;
- **прикладной**; протоколы этого уровня определяют, как прикладные программы будут использовать сеть для решения конкретных задач конечного пользователя.

Для упрощения запоминания английских названий уровней модели ISO OSI существует мнемоническая фраза «All People Seem To Need Data Processing» («всем людям, похоже, нужна обработка данных»). Первые буквы слов этой фразы соответствуют первым буквам названий уровней: Application, Presentation, Session, Transport, Network, Datalink и Physical. Аналогичной русской фразы автору, к сожалению, не встречалось. В реальной жизни модель ISO OSI не используется;

существовавшие когда-то буквальные её реализации, поддерживавшие ровно семь слоёв, распространения не получили. Для нас эта модель представляет интерес скорее как иллюстрация того, как *может быть* организовано взаимодействие между программами, работающими на разных компьютерах.

Специалисты иногда шутят, что модель OSI получилась семиуровневой, потому что в соответствующем комитете образовалось семь подкомитетов и каждый предложил что-то своё. В каждой шутке есть доля шутки, но всё остальное — чистая правда; модель OSI служит прекрасной иллюстрацией того, сколь «полезны» результаты деятельности комитетов, в особенности когда речь идёт о технической стандартизации. Отметим, что набор протоколов TCP/IP, на котором построена сеть Интернет и который в итоге вытеснил все остальные протоколы из активного употребления, был создан узкой группой людей по принципу ad hoc, то есть чтобы решить задачи, вставшие здесь и сейчас. Никаких комитетов в создании TCP/IP задействовано не было.

6.2.2. Физические и канальные протоколы

Хотя в явном виде семиуровневая модель OSI не используется, можно, проанализировав обычный сеанс повседневной работы с Интернетом, обнаружить задействованные в этом сеансе протоколы, *приблизительно* соответствующие уровням OSI. Начнём мы, естественно, с самого «низа» — физического уровня. Какой из существующих физических протоколов используется в вашем сеансе работы — зависит от способа, которым ваш компьютер[9] подключается к Интернету. Самый простой способ — обычная локальная сеть на основе витой пары; соответствующий протокол определяется спецификацией, известной под именем 100BASE-TX, она же известна как «стандарт IEEE 802.3u». Протокол описывает, в частности, какие кабели, штекеры и разъёмы должны использоваться для соединения, каковы должны быть технические характеристики проводов, какую следует использовать частоту для передачи сигнала по проводам, какое электрическое напряжение, как конкретно будут кодироваться двоичные цифры электрическими сигналами. Надо сказать, что спецификация 100BASE-TX этим не ограничивается, она определяет ещё и то, какой длины будут отдельные передаваемые порции информации («фреймы», или кадры), какие в этих фреймах предусмотрены служебные данные, включая аппаратный идентификатор (mac-адрес) сетевого интерфейса, для которого предназначен конкретный фрейм. В модели OSI это уже следующий, «канальный» уровень.

[9] Каково бы ни было устройство, с помощью которого вы просматриваете странички в Интернете — обычный настольный компьютер, ноутбук, планшет, мобильный телефон (смартфон) или даже «умные часы» — строго говоря, всё это компьютеры; мы не делаем между ними различий, обсуждая работу в Сети.

Если для подключения к сети вы используете WiFi, протокол физического уровня оказывается сложнее, ведь используется радио, а эфир, в отличие от проводов, один на всех. Приходится учитывать, что в одном месте могут работать разные сети WiFi, и они не должны друг другу мешать; в разных странах действуют различные правила использования радиочастот, и те же самые частоты могут использоваться устройствами, не имеющими отношения к WiFi. Спецификация WiFi предполагает использование 14 различных частотных каналов, в каждом из которых производится деление на временны́е срезы (*time slices*); выбор каналов и распределение временных срезов между разными сетевыми соединениями производится автоматически, без участия пользователя — так, чтобы обеспечить, насколько это возможно, успешную работу всех соединений, оказавшихся в зоне действия друг друга (возможно, принадлежащих совершенно разным сетям). Принципы и правила взаимодействия WiFi-устройств между собой описаны в наборе спецификаций, известных под общим названием IEEE 802.11. Эти спецификации, как и Ethernet, полностью покрывают физический уровень модели ISO OSI и часть канального уровня.

Ещё сложнее обстоят дела, если вы выходите в Интернет через сеть сотовой телефонной связи; здесь роль физических и канальных протоколов исполняют спецификации GSM, включающие специальные протоколы для передачи цифровых данных, такие как GPRS и более новые. Следует отметить, что спецификации GSM-сетей затрагивают не только физический и канальный, но и сетевой уровень.

С другой стороны, протоколы физического уровня бывают, наоборот, гораздо проще; например, если вам придёт в голову соединить два компьютера нуль-модемным кабелем через COM-порты, то в роли протокола физического уровня будет выступать RS232, спецификация которого достаточно проста даже для «самопальной» реализации радиолюбителями. К сожалению, из-за низкой скорости передачи данных RS232 в наше время вряд ли можно считать актуальным, когда речь идёт об организации компьютерной сети.

Поднимаясь на следующий уровень, канальный, мы обнаружим, что проблемы, которые по замыслу создателей OSI должны решаться протоколами этого уровня, частично решены спецификациями Ethernet и WiFi, но остаётся ещё кое-что. Сетевые интерфейсы, участвующие в одном многоточечном соединении, опознают друг друга по так называемым mac-адресам — уникальным шестибайтовым идентификаторам, которые приписываются каждому сетевому адаптеру прямо на заводе-изготовителе. Например, на одном из старых компьютеров автора этих строк основная сетевая карта имела mac-адрес `00:1F:C6:65:42:48`, причём первые два байта (`00:1F`) здесь указывают на компанию-производитель (ASUS), а остальные представляют собой серийный номер устройства. Современные операционные системы позво-

ляют отключить функции сетевой карты, отвечающие за формирование и распознавание Ethernet-кадров, содержащих именно такой mac-адрес. Пользователь может вручную задать произвольные шесть байт, которые и будут использоваться в качестве mac-адреса, причём формировать кадры будет не сетевой адаптер, а его драйвер, находящийся в ядре операционной системы. Но в большинстве случаев в этом нет нужды, а mac-адрес, зашитый в адаптер при его изготовлении, нас вполне устраивает.

Так или иначе, адреса, используемые для сетевых интерфейсов при настройке сети — так называемые ip-адреса, по которым, в частности, производится маршрутизация пакетов — с mac-адресами не имеют ничего общего, но при этом сетевые адаптеры должны знать, кто из них отвечает за какой сетевой адрес, чтобы в рамках многоточечного сетевого соединения отправлять пакеты кому следует, а не кому попало. Для автоматического построения таблицы соответствия ip-адресов и mac-адресов служит протокол ARP (*address resolution protocol*). Когда операционной системе требуется отправить пакет через многоточечное сетевое соединение, драйвер сетевого адаптера должен указать mac-адрес получателя, но система может его ещё не знать. В этом случае как раз и задействуется протокол ARP. Через сетевое соединение отправляется широковещательный (то есть предназначенный для всех, кто здесь есть) пакет, содержащий оформленный по правилам ARP запрос вида «у кого здесь такой-то ip-адрес?». Если кто-то из участников соединения опознаёт этот ip-адрес как свой, он формирует (опять-таки в соответствии с соглашениями ARP) ответный кадр, означающий «такой-то ip-адрес у меня, а мой mac-адрес такой-то». Получив этот ответ, участник, задавший вопрос, сохраняет полученную информацию в своих таблицах, так что для передачи следующего пакета на тот же адрес задействовать ARP уже не придётся.

Протокол ARP используется в многоточечных сетевых соединениях, таких как Ethernet и WiFi. В соединениях «точка-точка» он не нужен, но физические протоколы, используемые в таких соединениях, чаще всего не определяют никаких соглашений канального уровня (в отличие от физических протоколов для многоточечных соединений, которые вынужденно фиксируют соглашения о длине и структуре передаваемых кадров — то есть соглашения канального уровня). В роли канального протокола в соединениях «точка-точка» часто выступает протокол PPP (*Point to Point Protocol*). Кроме структуры кадра и способа контроля его целостности (через контрольную сумму), этот протокол содержит правила для установления соединения и проверки состояния линии связи (при отсутствии трафика по линии время от времени передаются специальные пакеты, чтобы удостовериться, что связь всё ещё установлена), для аутентификации участников соединения (это может быть нужно, если к линии связи физически может подключиться посторон-

ний; аутентификационная составляющая PPP активно использовалась в эпоху доступа в Интернет через телефонные модемы).

6.2.3. Протокол IP

Следующий слой в модели OSI — «сетевой». Часто можно встретить утверждение, что в Интернете в роли этого уровня выступает протокол IP[10]; это утверждение, как часто бывает, в принципе верно, но не совсем. С одной стороны, именно IP определяет структуру отдельного пакета, передающегося по сети через шлюзы, систему сетевых адресов и некоторые дополнительные функции; все остальные протоколы в Интернете работают «поверх» протокола IP, то есть «заворачивают» всю свою информацию, как служебную, так и прикладную, в IP-пакеты и с их доставкой полагаются на программы, реализующие IP. С другой стороны, в Интернете используется ещё несколько протоколов, которые, если классифицировать их в соответствии с моделью OSI, приходится тоже отнести к сетевому уровню, хотя они и работают поверх IP, то есть вроде бы должны относиться к более высокому слою. Наиболее важным из этих протоколов следует считать ICMP (*internet control message protocol*), без которого сети на основе IP вообще, скорее всего, не могли бы работать, ведь именно в соответствии с соглашениями ICMP участники сетевых соединений и шлюзы обмениваются между собой сообщениями о всяческих штатных и нештатных ситуациях, таких как обрыв соединения, отсутствие подходящего маршрута и другие случаи невозможности доставки пакета; в некоторых случаях сообщения ICMP содержат рекомендации по использованию альтернативного маршрута. Можно обнаружить и другие протоколы, работающие поверх IP, но очевидным образом относящиеся всё к тому же «сетевому» слою OSI.

Протокол IP изначально разрабатывался для американских военных и был предназначен для создания *гетерогенной* компьютерной сети — такой, которую невозможно вывести из строя, уничтожив какой-нибудь «самый главный узел». Предполагалось, что каналы, связывающие между собой отдельные узлы и подсети, могут иметь какую угодно природу, лишь бы по ним можно было передавать пакеты с данными, а схема соединения этих каналов между собой не обязана иметь никакой чёткой структуры, лишь бы только для любых двух компьютеров, входящих в сеть, существовала цепочка каналов (возможно, не единственная), по которой они могли бы друг с другом связаться. Модель, расчитанная на быстрое задействование резервных каналов при разрушении основных, была придумана для сохранения работоспособности в условиях войны, когда узлы сети и каналы связи могут быть в любой момент

[10] Считается, что аббревиатура IP означает *Internet Protocol*, хотя протокол появился задолго до возникновения слова *Internet* как имени собственного; изначально первая буква в названии протокола означала *internetworking*, то есть буквально «межсетевой».

разрушены. Сеть, получившая название ARPANET, начала работу в 1969 году; протокол IP в его нынешнем виде (IPv4) был опубликован в 1981 году и полностью заменил другие транспортные протоколы в ARPANET 1 января 1983 года; эту дату иногда называют днём рождения Интернета.

На первый взгляд может показаться странным, что именно военная модель и разработанный для неё сетевой протокол оказались самыми подходящими для построения всемирной компьютерной сети — а между тем всё именно так и есть. Явление, которое мы сейчас знаем под именем «Интернет», возникло на основе самого подходящего протокола из многих десятков разнообразных сетевых протоколов, существовавших в те времена. Этому легко найти объяснение. Явление такого масштаба не может возникнуть по указке сверху и тем более на основе каких-либо международных договорённостей, поскольку любые попытки централизованного построения подобной сети совершенно неизбежно утонут в бюрократической волоките. Интернет, каким мы его знаем, возник «снизу» — из частной инициативы, проявляемой разными, никак не связанными друг с другом людьми. Но в таких условиях сложно полагаться на какой-то «порядок», ведь в Интернете свой владелец у каждого канала, у каждой небольшой локальной сетки, у каждого компьютера; Интернет как целое — это некое социальное явление, никому не принадлежащее и никем не контролируемое. Функционирование сети обеспечивает глобальная инфраструктура, элементы которой находятся в частных руках — в десятках тысяч частных рук.

Побочным эффектом этого становится заведомая ненадёжность каждого отдельно взятого элемента глобальной инфраструктуры. Любой канал может перестать работать даже не вследствие аварии или другого ЧП, а просто потому, что его владелец потерял к нему интерес; сложность глобальной инфраструктуры Интернета такова, что каналы связи вводятся в работу и выходят из работы буквально ежеминутно, своей динамичностью сеть напоминает муравейник с его вечной суетой. Аналогию с муравейником можно продолжить: надёжность одного отдельно взятого муравья ничтожна, но их много и они взаимозаменяемы, поэтому надёжность муравейника как целого достаточно высока: уничтожить его можно разве что огнём, и то не сразу.

Очевидно, что такой способ существования сети прекрасно ложится на исходную модель протокола IP. Каналы здесь постоянно выходят из строя именно так, как это предполагалось создателями ARPANET, пусть и не в результате военных действий, а по сугубо мирным причинам; сеть при этом сохраняет работоспособность, поскольку информация сразу же начинает передаваться по другим каналам «в обход» прекратившего работу.

Прежде чем двигаться дальше, введём важное понятие ***сетевого хоста***[11]. Под хостом понимается компьютер, подключённый к компьютерной сети и (что важно) имеющий свой собственный сетевой адрес, что позволяет ему получать разнообразные запросы от других компьютеров. Как мы увидим позже, часто встречается ситуация, когда компьютер подключён к сети, но хостом в ней не является и может лишь направлять запросы к другим компьютерам, но сам получать их не может.

В качестве сетевых адресов в протоколе IP используются так называемые ***ip-адреса***, состоящие из четырёх восьмибитных байтов. По традиции ip-адреса записывают в виде четырёх *десятичных* чисел, разделённых точками: `192.168.215.17`, `203.0.113.171`, `10.10.15.0` и т. п. В двоичном (побитовом) представлении эти адреса будут выглядеть так: «`1100 0000 1010 1000 1101 0111 0001 0001`»; «`1100 1011 0000 0000 0111 0001 1010 1011`»; «`0000 1010 0000 1010 0000 1111 0000 0000`». Отметим, что в компьютерных сетях принят порядок байтов в целых числах, соответствующий архитектурам *big endian* (см. т. 1, §1.6.2), то есть старший байт всегда записывается первым.

Как мы уже видели в примере, приведённом в § 6.1.2, при настройке маршрутов для передачи пакетов через шлюзы постоянно приходится перечислять компьютеры, для которых будет использоваться тот или иной маршрут, но при этом, как правило, все хосты, относящиеся к одному сетевому соединению (или, в более общем случае, к одной сети, что́ бы под сетью ни понималось), достижимы с помощью одного и того же маршрута. Необходимость как-то обозначить разом все хосты той или иной сети возникает не только при настройке маршрутов, но и во многих других случаях; поскольку в одну сеть может входить несколько тысяч хостов, перечислять все их адреса по одному не годится.

При работе с сетями на основе IP принято соглашение, что адреса можно объединять в ***подсети*** (англ. *subnet*). Размер подсети (т. е. количество адресов в ней) всегда представляет собой степень двойки, при этом в неё входят *те и только те адреса*, для которых совпадают сколько-то первых битов. Оставшиеся биты в пределах подсети могут принимать произвольные значения, что как раз и даёт степень двойки. Например, если нам нужна подсеть на восемь адресов, то варьироваться в такой подсети будут три последних бита ($8 = 2^3$); поскольку всего в ip-адресе 32 бита, остальные 29 фиксируются и образуют *адрес подсети*, также называемый *ip-префиксом*. Точно так же, если нам нужна подсеть

[11] Этот термин происходит от английского *network host*; придумать русский перевод никто не потрудился. На всякий случай отметим, что английское слово *host* буквально переводится как «хозяин, принимающий гостей»; в старину этим словом обозначали, например, хозяина постоялого двора или трактира. Читатель наверняка встречался с этим словом и его производными как в области гостиничного бизнеса, так и в области телекоммуникаций (например, в словосочетании «услуги хостинга»).

на 256 адресов, то варьироваться будут 8 младших разрядов адреса, а остальные 24 составят адрес подсети (префикс); для подсети на 1024 адреса варьироваться будут уже 10 разрядов, а длина префикса будет равна 22. Адрес подсети записывают так же, как обычный ip-адрес, четырьмя десятичными числами через точку; варьируемые биты при этом принимают равными нулю. Иначе говоря, адрес подсети записывается так же, как наименьший из входящих в эту подсеть ip-адресов.

Подсеть обычно обозначают в виде её адреса подсети, после которого через дробную черту записывают длину префикса. Например, рассмотрим подсеть `203.0.113.32/28`; длина префикса здесь — 28 бит, так что на варьируемую часть адреса остаётся 4 бита; следовательно, эта подсеть содержит $2^4 = 16$ адресов. Нетрудно сообразить, что это ip-адреса с `203.0.113.32` по `203.0.113.47`. В двоичном виде все адреса этой подсети имеют вид «`11001011 00000000 01110001 0010xxxx`», где `xxxx` — варьируемые биты.

В учебниках часто встречается утверждение, что первый и последний адреса любой подсети нельзя использовать для хостов, поскольку самый первый адрес обозначает всю подсеть целиком, а самый последний считается «широковещательным» и предназначен для пакетов, адресованных всем хостам подсети. На самом деле это не совсем так. При использовании ip-подсети для нумерации хостов, например, в локальной сети на основе Ethernet действительно обычно самый первый адрес не используют (хотя при желании использовать его можно), а самый последний настраивают как широковещательный, но так делают только если адресов достаточно. При нехватке ip-адресов вполне можно задействовать первый и последний адреса подсети для нумерации хостов, нужно только соответствующим образом настроить все хосты данной сети, чтобы, в частности, они не реагировали на последний адрес подсети в качестве широковещательного (практически любая система это позволяет).

Бывает и так, что ip-подсеть обозначает не хосты, включённые в какое-то сетевое соединение, а просто абстрактный набор адресов. В такой ситуации первый и последний адрес тем более не имеют никакого особого смысла.

Часто можно увидеть обозначение вроде `203.0.113.41/32`. На первый взгляд понятие «подсети с префиксом в 32 бита» не имеет смысла, ведь в такой подсети не остаётся ни одного бита на вариативную часть; но *один* адрес в эту подсеть всё же входит — в нашем примере это адрес `203.0.113.41`. Иначе говоря, длина префикса 32 бита используется для обозначения *одного отдельно взятого ip-адреса*.

Очевидно, что для каждого адреса можно указать подсеть *любого размера*, в которую он входит. Например, адрес `198.51.100.115` входит в следующие подсети:

```
198.51.100.115/32   198.51.96.0/21   198.0.0.0/10
198.51.100.114/31   198.51.96.0/20   198.0.0.0/9
198.51.100.112/30   198.51.96.0/19   198.0.0.0/8
198.51.100.112/29   198.51.64.0/18   198.0.0.0/7
198.51.100.112/28   198.51.0.0/17    196.0.0.0/6
198.51.100.96/27    198.51.0.0/16    192.0.0.0/5
198.51.100.64/26    198.50.0.0/15    192.0.0.0/4
198.51.100.0/25     198.48.0.0/14    192.0.0.0/3
198.51.100.0/24     198.48.0.0/13    192.0.0.0/2
198.51.100.0/23     198.48.0.0/12    128.0.0.0/1
198.51.100.0/22     198.32.0.0/11    0.0.0.0/0
```

Последняя «подсеть» обозначает весь Интернет. **Если здесь что-то непонятно, попробуйте представить адреса перечисленных подсетей в двоичном виде**.

Чтобы минимизировать трудозатраты при настройке сетей, системные администраторы всегда стараются выбрать ip-адреса и подсети так, чтобы ip-адреса, используемые в каждой сети, чтó бы ни понималось под «сетью», можно было указать в виде одной ip-подсети. Из этого естественным образом следует, что компьютер, входящий больше чем в одну сеть, должен иметь собственный адрес в *каждой* из сетей, в которые он входит. Поскольку каждое сетевое соединение обычно рассматривается как отдельная сеть (возможно, являющаяся частью большей сети), компьютеру, выполняющему роль шлюза, нужны адреса в каждом из соединений, в которых он участвует. Поэтому **в сетях на основе протокола IP сетевой адрес приписывается не компьютеру, а сетевому интерфейсу**, так что шлюзы обычно имеют больше одного адреса — точнее, столько адресов, сколько у данного шлюза интерфейсов.

Из всего пространства ip-адресов некоторые ip-подсети выделены для особых нужд. В частности, подсеть `0.0.0.0/8` (то есть все ip-адреса, первый байт которых равен нулю) зарезервирована для обозначения хостов в этой же сети; по правде говоря, в такой роли эти адреса нигде не используются и операционные системы такое их использование не поддерживают, но использовать их для чего-то ещё всё равно нельзя. Подсеть `127.0.0.0/8` (все адреса, у которых старший байт равен 127) используется внутри одного хоста для обозначения его же самого, когда нужно заставить программу, обычно работающую через сеть, установить связь с другой программой, запущенной на том же компьютере. Опять-таки, в реальности используется всего один адрес — `127.0.0.1`, этот адрес действительно присутствует в любой системе, поддерживающей TCP/IP (а в наши дни это значит — практически в любой системе мира) и обозначает «сам этот компьютер». Остальные адреса, начинающиеся с байта 127, просто не используются. Точно так же не используются адреса, начинающиеся с байтов 240–255. Кто-то когда-то

отнёс их к «зарезервированным»; отменить этот их статус никто так и не решился.

Адреса, начинающиеся с байтов 224-239, формально говоря, используются — они были выделены для протоколов «группового вещания» (или *мультикастных*, от английского *multicast*). Общая идея «группового вещания» состоит в том, что для одного источника передаваемой информации имеется неопределённое число её получателей, примерно так, как это происходит при передаче теле- и радиопрограмм. Отдельный ip-адрес из подсети `224.0.0.0/4` представляет собой некий аналог *канала*.

Исходно предполагалось, что передающий «мультивещательную» информацию просто начинает передавать в сеть пакеты, адресованные на выбранный ip-адрес; все, кто желает получать информацию, отправленную на этот ip-адрес, сообщают об этом ближайшему маршрутизатору по протоколу IGMP, маршрутизаторы также передают друг другу информацию о собственной заинтересованности в получении данного канала, в результате чего рано или поздно между отправителем и каждым из получателей возникает связная цепочка маршрутизаторов, каждый из которых знает, каким ещё маршрутизаторам следует передавать пакеты, направленные на данный ip-адрес. При этом, по идее, должна достигаться серьёзная экономия на загрузке каналов: например, отправитель посылает ближайшему маршрутизатору только один экземпляр каждого пакета, сколько бы при этом ни было получателей; если на очередном шаге нужно передать «мультикастный» пакет нескольким получателям через одно и то же сетевое соединение, то такой пакет тоже передаётся всего один.

Некоторые из мультикастных ip-адресов зарезервированы для общения между собой маршрутизаторов в крупных локальных сетях, где шлюзы сами динамически выстраивают таблицу маршрутизации; впрочем, изначально ip multicast предназначался для передачи через Интернет аудио- и видеопотоков, аналогичных традиционному радио и телевидению. Проблема в том, что протокол IGMP поддерживается далеко не везде; даже если ближайший к вам маршрутизатор готов принять от вас запрос на «присоединение к группе» (т. е. пожелание принимать мультикастный трафик с заданным адресом), до отправителя пакетов на этот адрес, скорее всего, цепочка так и не выстроится.

Существуют и другие области адресов, зарезервированных под специальные нужды. Для нас могут представлять интерес подсети `10.0.0.0/8`, `172.16.0.0/12` и `192.168.0.0/16`, которые предназначены для использования *в частных сетях*: кто угодно, выстраивая свою компьютерную сеть, может использовать в ней любые из этих адресов по своему усмотрению, но при этом не должен выпускать пакеты с такими адресами за пределы своей сети, то есть компьютеры, имеющие адреса из этих трёх блоков, не видны из Интернета (в нашей терминологии — не являются хостами Интернета), хотя благодаря хитрым преобразованиям адресов сами могут осуществлять доступ к серверам через Интернет.

Так, герои нашего примера, приведённого в §6.1.2, могли бы договориться, что Борис в своём доме будет использовать сеть `192.168.12.0/24` (то есть любые адреса, начинающиеся с `192.168.12.`),

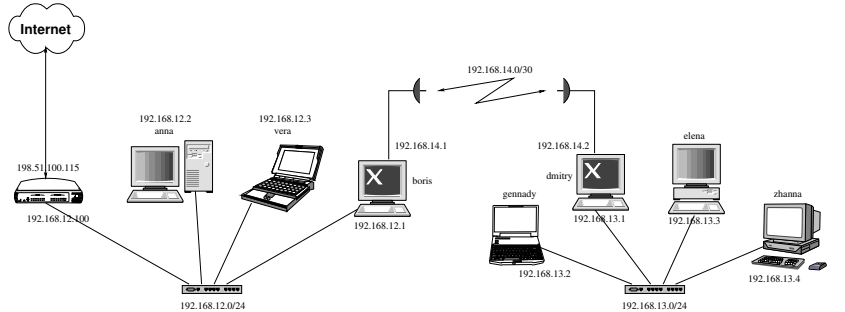

Рис. 6.5. Пример назначения ip-адресов

а Дмитрий — сеть `192.168.13.0/24`. После этого Борис настроил бы на своём собственном компьютере адрес `192.168.12.1`, на компьютере Анны — адрес `192.168.12.2`, на компьютере Веры — `192.168.12.3`; Дмитрий для своего компьютера мог бы взять адрес `192.168.13.1`, для компьютеров Геннадия, Елены и Жанны — адреса `192.168.13.2`, `192.168.13.3` и `192.168.13.4` (см. рис. 6.5). Впрочем, с тем же успехом можно было бы использовать и другие значения младшего байта адреса, например, `192.168.13.17`, `192.168.13.49`, `192.168.13.121` и `192.168.13.197`, лишь бы все они входили в подсеть. Экономить адреса в частных сетях смысла нет, поэтому наши админы-любители могли бы считать адреса `192.168.12.0` и `192.168.13.0` сетевыми, а адреса `192.168.12.255` и `192.168.13.255` — широковещательными.

Радиоканал между компьютерами Бориса и Дмитрия — это отдельное сетевое соединение, для которого тоже можно выделить подсеть, но использовать подсеть на 256 адресов тут несколько глупо, поскольку в соединении «точка-точка» не может быть больше двух участников; поэтому для своего радиоканала Борис и Дмитрий могли бы, например, воспользоваться подсетью `192.168.14.0/30`, установив на радиоканальном интерфейсе компьютера Бориса адрес `192.168.14.1`, а у Дмитрия — `192.168.14.2`; в роли сетевого и широковещательного тут выступали бы адреса `192.168.14.0` и `192.168.14.3`.

После подключения к Интернету Борис мог бы дать маршрутизатору провайдера адрес `192.168.12.100` (или любой другой из своей подсети). Именно этот адрес должен быть объявлен «шлюзом по умолчанию» для компьютеров, находящихся с ним в одной подсети; а для компьютеров из сети Дмитрия в этой роли будет выступать адрес `192.168.13.1` (компьютер Дмитрия) — именно через него Геннадий, Елена и Жанна будут «видеть» весь Интернет.

Прежде чем показать, как будут выглядеть таблицы маршрутизации на компьютерах из нашего примера, отметим ещё один очень важный момент. Правила маршрутизации бывают двух видов: «пакеты для

адресов из данной подсети пересылать через данный сетевой интерфейс» и «пакеты для данной подсети передавать заданному шлюзу для дальнейшей доставки». В обоих случаях требуется указать подсеть (т. е. адрес сети и маску), для которой действует данное правило; после этого в первом варианте указывается обозначение интерфейса, во втором — ip-адрес шлюза. Несложно догадаться, что маршрутизационное правило вида «через интерфейс» для многоточечных соединений может (и должно) использоваться только для указания подсети, используемой для данного соединения, то есть включающей *непосредственно* подключённые к нашему интерфейсу компьютеры; соединения «точка-точка» такого ограничения не имеют. Естественно, для успешной работы правила «через шлюз» нужно, чтобы другое правило определяло, как до этого шлюза добраться.

Допустим, интерфейс для подключения к локальной сети Ethernet на всех компьютерах, участвующих в нашем примере, называется `eth0`, интерфейсы радиоканала — `wlan0`; именно так обозначаются сетевые интерфейсы в ОС Linux. Самыми простыми окажутся маршрутизационные таблицы на компьютерах Геннадия, Елены и Жанны, они будут содержать всего по два правила:

```
192.168.13.0/24   ->   eth0
0.0.0.0/0         ->   192.168.13.1
```

Чуть сложнее будет ситуация на машинах Анны и Веры, поскольку весь Интернет эти машины «видят» через один шлюз, а локальную сеть соседей — через другой. Выглядеть это будет так:

```
192.168.12.0/24   ->   eth0
192.168.13.0/24   ->   192.168.12.1
0.0.0.0/0         ->   192.168.12.100
```

На машинах Бориса и Дмитрия таблица окажется ещё более «заковыристой», ведь каждая из них участвует в двух сетевых соединениях. На машине Бориса таблица маршрутизации будет такой:

```
192.168.12.0/24   ->   eth0
192.168.14.0/30   ->   wlan0
192.168.13.0/24   ->   192.168.14.2
0.0.0.0/0         ->   192.168.12.100
```

Компьютер Дмитрия потребует следующих маршрутных правил:

```
192.168.13.0/24   ->   eth0
192.168.14.0/30   ->   wlan0
0.0.0.0/0         ->   192.168.14.1
```

Сложнее всего будут настройки маршрутизатора, через который в наших сетях осуществляется выход в Интернет. Предположим, что его внешний сетевой интерфейс, обращённый в сеть провайдера, называется wan0 и работает в режиме «точка-точка», а ip-адрес, выделенный нам провайдером — `198.51.100.115`[12]. Непосредственно наш маршрутизатор видит только сеть `192.168.12.0/24`, а сеть `192.168.13.0/24` ему доступна через шлюз, тогда как весь Интернет — через интерфейс `wan0`. Таблица маршрутизации получается такая:

```
192.168.12.0/24   ->   eth0
192.168.13.0/24   ->   192.168.12.1
0.0.0.0/0         ->   wan0
```

Сама по себе эта таблица кажется довольно простой, но на сей раз это лишь начало истории. Дело в том, что адреса, начинающиеся на «`192.168.`», как мы уже отмечали, *частные*. С одной стороны, это удобно, поскольку каждый волен использовать такие адреса в своей сети как ему будет угодно — именно это обстоятельство позволило нашим персонажам организовать свои домашние сети. С другой стороны, очевидное неудобство этих адресов состоит в том, что компьютеры за пределами отдельно взятой локальной сети не знают и не могут знать ничего о том, что в этой локальной сети использованы такие адреса. Иначе говоря, маршрутизатор с его внешним адресом является *хостом* в Интернете, а вот компьютеры Бориса, Дмитрия и их домочадцев — нет.

Тем не менее благодаря небольшой хитрости все эти компьютеры могут получить доступ ко всему, что есть в Интернете, практически наравне с полноценными хостами. Это потребует простой дополнительной настройки маршрутизатора, которая включит так называемую ***трансляцию сетевых адресов*** (англ. *network address translation*, NAT). В данном конкретном случае можно заметить, что все адреса, используемые в нашей примерной сети, «накрываются» подсетью `192.168.12.0/22` — эта подсеть, как нетрудно видеть, включает все адреса с `192.168.12.0` по `192.168.15.255`; остаётся лишь сообщить маршрутизатору, что все пакеты, исходящие из этой подсети и адресованные хостам за её пределами, подлежат трансляции через внешний адрес маршрутизатора — адрес `198.51.100.115`. После этого маршрутизатор, видя такой пакет, подменяет в нём обратный адрес на свой собственный и в таком виде отправляет в Интернет — как будто от своего имени. При получении *ответного* пакета маршрутизатор производит обратную замену и отправляет полученный пакет в локальную

[12]На самом деле такой адрес нам не мог бы выделить ни один провайдер в мире, поскольку подсеть `198.51.100.0/24` зарезервирована для примеров; именно в этом качестве — для примера — мы её и используем. В реальной жизни провайдер должен дать нам адрес из тех, которые выделены ему из общего адресного пространства и больше нигде в мире не используются.

сеть. Адрес 198.51.100.115 в такой ситуации называется *внешним*, а адреса из подсети 192.168.12.0/22 — *внутренними*.

Хосты в Интернете, к которым обращаются машины из нашей локальной сети, воспринимают это так, как если бы к ним приходили запросы только от внешнего адреса, про существование внутренних адресов они могут не догадываться. Подчеркнём ещё раз, что машины, работающие на внутренних адресах, вообще не видны из Интернета (не являются хостами Интернета), то есть они могут посылать запросы другим машинам и получать ответы, но сами принимать запросы не могут.

На самом деле, конечно, всё далеко не так просто, ведь с одного и того же «внутреннего» адреса в сеть отправляются пакеты, относящиеся к самым разным соединениям и прочим сеансам работы. Для корректного выполнения трансляции адресов маршрутизатор должен знать, как устроены протоколы транспортного уровня, а в ряде случаев (когда на транспортном уровне отсутствует соединение как таковое, например, при использовании UDP) — учитывать также и протоколы более высоких уровней, вплоть до прикладного. Кроме прочего, маршрутизатор должен помнить, какие соединения от чьего имени в настоящий момент находятся в работе, чтобы не перепутать, какой ответ кому отдать. NAT как явление появился во второй половине 1990-х годов, когда начала ощущаться нехватка ip-адресов, то есть гораздо позже, чем большинство используемых в Интернете протоколов; исходно TCP/IP и протоколы прикладного уровня не были рассчитаны на такой режим работы. Поэтому, например, протокол FTP, предназначенный для передачи файлов, сначала вообще не мог работать из локальных сетей, скрытых за NAT'ом, а позже этот протокол модифицировали, добавив специальный «пассивный» режим.

Возвращаясь к настройкам маршрутизации в нашем примере, мы можем заметить, что адреса из подсети 192.168.14.0/30, используемые на радиоканале между двумя локальными сетями, не упоминаются в настройках нигде, кроме самих компьютеров Бориса и Дмитрия. В принципе ничто не мешает указать эти адреса в таблицах маршрутизации на всех компьютерах; например, на компьютерах Анны и Веры таблица маршрутизации станет такой:

```
192.168.12.0/24   ->   eth0
192.168.13.0/24   ->   192.168.12.1
192.168.14.0/30   ->   192.168.12.1
0.0.0.0/0         ->   192.168.12.100
```

После этого компьютеры Бориса и Дмитрия будут доступны как по своим исходно установленным адресам (192.168.12.1 и 192.168.13.1), так и по адресам интерфейсов радиоканала — 192.168.14.1 и 192.168.14.2. В большинстве случаев так и поступают, но есть ещё один вариант, который, как ни странно, часто упускают из вида даже опытные системные администраторы: радиоканал (и вообще любое соединение вида «точка-точка») можно вообще не снабжать ip-адресами, ведь, как мы

уже отмечали, такие соединения позволяют указывать маршрутные правила «через интерфейс» для произвольных подсетей, а не только непосредственно подключенных, как в случае многоточечных соединений.

Если избавиться от отдельной подсети на радиоканале, то на машине Бориса таблица маршрутизации станет такой:

```
192.168.12.0/24   ->   eth0
192.168.13.0/24   ->   wlan0
0.0.0.0/0         ->   192.168.12.100
```

а на компьютере Дмитрия — такой:

```
192.168.13.0/24   ->   eth0
0.0.0.0/0         ->   wlan0
```

Как можно заметить, маршрутизация при этом становится даже проще.

Есть ещё один важный факт, который следует помнить при работе с таблицами маршрутизации. **Если ip-адрес подходит под несколько разных правил маршрутизации, то маршрутизатор всегда использует то правило, в котором указана подсеть наименьшего размера**. Этот принцип по-английски звучит как *use most specific rule*. Вспомним, например, приведённую чуть выше таблицу маршрутизации для компьютеров Анны и Веры (при условии использования отдельной подсети для радиоканала); применив most-specific rule, её можно несколько упростить:

```
192.168.12.0/24   ->   eth0
192.168.12.0/22   ->   192.168.12.1
0.0.0.0/0         ->   192.168.12.100
```

При таких настройках все пакеты, адресованные в подсеть `192.168.12.0/22`, будут передаваться через шлюз `192.168.12.1`, *за исключением* пакетов, адресованных в подсеть `192.168.12.0/24`, ведь эта подсеть вчетверо меньше по размеру и для неё указано отдельное правило — передавать пакеты через интерфейс `eth0`, используя непосредственно подключённое соединение.

В нашем примере подсети в 256 адресов используются для сетевых соединений на три и четыре компьютера. Полезно иметь в виду, что адреса можно использовать гораздо более экономно. Конечно, экономить частные адреса, в том числе адреса `192.168.*.*`, в большинстве случаев нет никакого смысла, но иногда приходится иметь дело с глобально-уникальными адресами, которых вечно не хватает; встречаются, хотя и редко, ситуации, когда экономить приходится и частные адреса тоже — например, когда наша сеть должна быть частью бóльшей сети, использующей частные адреса.

Для примера допустим, что нашим друзьям зачем-то потребовалось, с одной стороны, обойтись по возможности ме́ньшей подсетью, а с другой стороны — обязательно использовать ip-адреса для интерфейсов радиоканала, плюс к тому во всех подсетях иметь как широковещательные, так и сетевые адреса, не задействуя их для конкретных компьютеров. Ясно, что для каждой из двух локальных сетей можно воспользоваться ip-подсетью на восемь адресов (`/29`), ведь такая подсеть позволяет обслуживать до шести компьютеров, а вот подсеть следующего размера — `/30` — содержит всего четыре адреса, из которых лишь два могут быть приписаны компьютерам. Но ведь есть ещё и радиоканал, для которого нужна как раз подсеть на четыре адреса, и плюс к тому стоит задуматься о размере подсети, которая «накроет» все используемые адреса разом. Если использовать две ip-подсети по восемь адресов и одну на четыре адреса, то их общая ёмкость составит 20 адресов. Общую подсеть придётся сделать на 32 адреса (`/27`), ведь в подсеть меньшего размера 20 адресов уже не поместятся. Но при этом пропадёт целых 12 адресов, и этим можно воспользоваться, чтобы расширить одну из используемых подсетей. Расширять подсеть, выделенную для радиоканала, нет никакого смысла, поскольку это соединение «точка-точка» и в нём больше двух интерфейсов быть не может, что называется, по определению; иное дело — подсети, используемые для локальных домашних сетей. Сеть в доме Дмитрия больше (пусть и всего на один компьютер), так что логично было бы расширить именно её, воспользовавшись следующим возможным размером ip-подсети — 16 адресов (`/28`).

Итак, для начала выбираем произвольную ip-подсеть на 32 адреса; пусть это будет `192.168.45.64/27`, то есть диапазон от `192.168.45.64` до `192.168.45.95`. Первые восемь адресов, то есть подсеть `192.168.45.64/29`, отведём под локальную сеть Бориса, назначив его компьютеру адрес `192.168.45.65`, а компьютерам Анны и Веры — соответственно `192.168.45.66` и `192.168.45.67`. Адрес `192.168.45.71` в этой сети будет широковещательным; маршрутизатору, соединяющему нас с Интернетом, выделим адрес `192.168.45.70`, а адреса 68 и 69 оставим «на вырост».

Поскольку мы договорились, что для локальной сети Дмитрия задействуем подсеть на 16 адресов, для выделения этой сети у нас не остаётся никаких иных вариантов, кроме как использовать вторую половину нашей «общей» ip-подсети. Читатель, несомненно, догадался, что это будет подсеть `192.168.45.80/28`. В ней можно выделить адрес `192.168.45.81` для Дмитрия, адреса `192.168.45.82`, `192.168.45.83` и `192.168.45.84` — для Геннадия, Елены и Жанны; широковещательным здесь будет адрес `192.168.45.95`, а адреса с 85 по 94 останутся в резерве.

Остаётся ещё радиоканал. Здесь у нас имеется два варианта: подсети `192.168.45.72/30` и `192.168.45.76/30`. Мы выберем второй

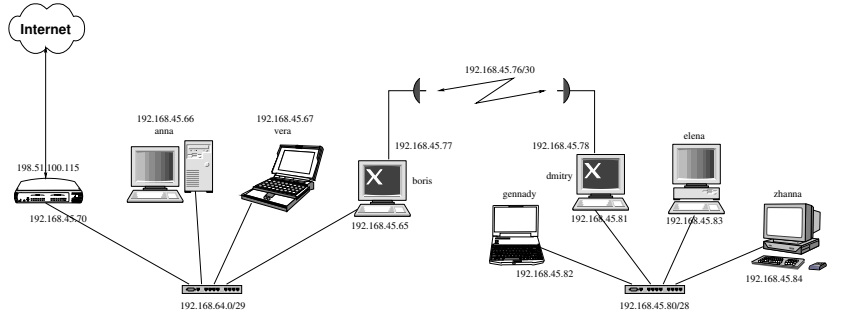

Рис. 6.6. Более компактное использование ip-адресов

вариант, установив на радиоинтерфейсе компьютера Бориса адрес `192.168.45.77`, а на компьютере Дмитрия — адрес `192.168.45.78`. Адреса с `192.168.45.72` по `192.168.45.75` так и останутся не входящими ни в одну используемую подсеть, но теперь их всего четыре, а не 12. Новое распределение адресов в сети Бориса и Дмитрия показано на рис. 6.6; написать таблицы маршрутизации для этого варианта предоставим читателю в качестве упражнения.

Во всех наших примерах мы задавали размер ip-подсети, указывая через дробную черту длину её префикса. Достаточно часто можно столкнуться с совершенно иным обозначением — в виде так называемой ***маски подсети***. Такая маска представляет собой 32-битное число, в котором сначала идёт N единичных разрядов (где N — длина префикса соответствующей подсети), а остальные разряды сброшены в ноль. Записывается маска точно так же, как и ip-адрес — в виде четырёх десятичных чисел, разделённых точками, каждое число соответствует одному байту. Пусть, например, нам нужна маска для подсети `/27`. В двоичном виде это будут байты `11111111 11111111 11111111 11100000`, первые три из них представляют собой двоичную запись числа 255, последний — числа 224. Таким образом, подсети `/27` соответствует маска `255.255.255.224`. Маски для всех возможных размеров подсетей приведены в таблице 6.1.

Кроме адресации и принципов передачи пакетов, протокол IP предусматривает для шлюзов некоторые дополнительные обязанности. В частности, каждый пакет имеет *ограниченный срок жизни* (англ. *time to live*, TTL), устанавливаемый при его создании; шлюз при передаче пакета должен уменьшить это значение, и когда оно становится равно нулю, пакет уничтожается. Это делается, чтобы избежать засорения каналов «бесхозными» пакетами при случайных зацикливаниях маршрутных правил. Разные системы устанавливают различные значения TTL по умолчанию: большинство версий Linux использует значение `64`, Windows — `128`, встречаются системы, позволяющие своим пакетам

Таблица 6.1. Маски для ip-подсетей разных размеров

префикс	маска	префикс	маска
/32	255.255.255.255	/16	255.255.0.0
/31	255.255.255.254	/15	255.254.0.0
/30	255.255.255.252	/14	255.252.0.0
/29	255.255.255.248	/13	255.248.0.0
/28	255.255.255.240	/12	255.240.0.0
/27	255.255.255.224	/11	255.224.0.0
/26	255.255.255.192	/10	255.192.0.0
/25	255.255.255.128	/9	255.128.0.0
/24	255.255.255.0	/8	255.0.0.0
/23	255.255.254.0	/7	254.0.0.0
/22	255.255.252.0	/6	252.0.0.0
/21	255.255.248.0	/5	248.0.0.0
/20	255.255.240.0	/4	240.0.0.0
/19	255.255.224.0	/3	224.0.0.0
/18	255.255.192.0	/2	192.0.0.0
/17	255.255.128.0	/1	128.0.0.0
		/0	0.0.0.0

«пережить» всего 30 переходов через шлюзы (что характерно, этого почти всегда хватает).

Кроме того, обычно каждое сетевое соединение определяет некий максимальный размер пакета, который можно через него передать (*maximal transfer unit*, MTU; чаще всего 1500 байт), и в соответствии с протоколом IP, если пакет не может быть передан через очередное соединение из-за превышения размера, шлюз должен этот пакет разбить на части и передать каждую часть в виде отдельного пакета; обратная сборка пакета в этом случае производится получателем. IP предусматривает контроль целостности заголовка пакета (но не всего пакета) через подсчёт контрольной суммы.

Поскольку ip-адрес представляет собой 32-разрядное целое число, всего их существует 2^{32}, то есть чуть больше 4 миллиардов, но, как уже говорилось, в силу тех или иных причин многие из этих адресов не могут использоваться. Кроме того, на начальных этапах развития сети Интернет некоторым организациям в пользование выдавались адресные блоки гигантского размера, например, с префиксом /8 или /16, а подсети размера /24 (так называемые «сети класса C») одно время считались вообще минимально возможными для предоставления отдельно взятой организации.

Уже в начале 1990-х годов, когда широкая публика ещё не успела толком узнать о существовании Интернета, стало понятно, что ip-адресов скоро перестанет хватать. Раздача адресов направо и налево, характерная для 1980-х, сменилась более бережным отношением к пространству ip-адресов, а затем, ближе к рубежу веков, и вовсе превратилась в режим жёсткой экономии, когда получение уникальных ip-адресов для конечного пользователя стало проблемой.

С 1992 года начали появляться различные предложения о замене имеющегося протокола IP новой версией; к 1996 году создание нового протокола, получившего название «IPv6», было более-менее завершено.

В IPv6 адрес по-прежнему имеет фиксированный размер, но этот размер составляет 128 бит, так что пространство адресов IPv6 включает 2^{128} адресов. Чтобы понять, насколько это много, достаточно вспомнить легенды о числе 2^{64} (одна из них связана с изобретением шахмат, вторая — с «ханойскими башнями»), а затем прикинуть, что 2^{128} — это, ни много ни мало, 2^{64} *в квадрате*. Записываются такие адреса в шестнадцатеричной системе счисления, но всё равно получается довольно громоздко, примерно так: `2001:0db8:21b1:a742:0000:ff00:0042:8326`. В настоящее время при выдаче блоков адресов IPv6 обычно провайдеры получают подсети с префиксом /32, то есть блоки, содержащие $2^{128-32} = 2^{96}$, или примерно $6,5 \cdot 10^{27}$ адресов, а конечным пользователям, запросившим адреса IPv6, выделяется блок /64, содержащий 2^{64} уникальных адресов (отметим, что это число можно получить, если *возвести в квадрат* общий размер адресного пространства протокола IPv4 — и всю эту прорву адресов предоставляют одному пользователю в единоличное распоряжение).

С повсеместным внедрением IPv6, что вполне закономерно, возникли сложности. Дело в том, что, в отличие от 1983 года, когда переход к использованию протокола IPv4 произошёл одномоментно в соответствии с указаниями администрации ARPANET, у Интернета никакой администрации нет, а если бы она и была, требованию «с такого-то момента всем перейти на новый протокол» никто бы не подчинился, Интернет для этого слишком большой и сложный. Поддержка IPv6 сейчас есть практически во всех используемых операционных системах и на всех устройствах, вообще способных работать в Интернете, но не помогает даже это. Дело в том, что, если разместить некий общедоступный ресурс (например, web-сайт) на адресе IPv6, не дав ему адреса IPv4, то для пользователей, не имеющих поддержки IPv6, этот ресурс будет недоступен. Для владельцев сайтов это неприемлемо, поскольку ведёт к потери большей части аудитории (а если называть вещи своими именами — то аудитория теряется вообще вся, ведь пока что с IPv6 никто реально не работает, несмотря на декларируемую готовность); поэтому все создаваемые сайты по-прежнему доступны (и, судя по всему, будут доступны в сколько-нибудь обозримом будущем) через адреса IPv4. Что касается пользователей Интернета, не предоставляющих от своего имени никаких доступных кому-то ещё ресурсов (а таких, к сожалению, подавляющее большинство), то им вполне нормально живётся вообще без ip-адресов — точнее говоря, на «приватных» ip-адресах, невидимых из Интернета, но позволяющих получить доступ ко всем имеющимся в Интернете сайтам и прочим серверам.

Энтузиасты предсказывали массовый переход на IPv6, когда «старые» ip-адреса кончатся; но когда в 2011 году между пятью региональными регистратурами ip-адресов были распределены последние пять блоков /8, на IPv6 так никто и не перешёл. Можно было встретить утверждение, что переход на IPv6 пойдёт быстрее, когда эти пять блоков подойдут к концу, но и этого не произошло. Региональные регистратуры, достигнув определённого предела исчерпания свободного ip-пространства, окончательно урезали возможности получения ip-адресов для новых провайдеров и автономных сетей: теперь, создав новый провайдер доступа в Интернет или хостинговый оператор, можно

получить для него подсеть /22, содержащую всего 1024 адреса, больше никто не даст. Конечным пользователям практически нереально получить в своё распоряжение больше *одного* адреса, да и один адрес дают не во всех провайдерах; такое положение жесточайшей экономии адресов существует уже несколько лет и (предположительно) при нынешней скорости раздачи адресов IPv4 может протянуть ещё не одно десятилетие. Формально организации, ответственные за распределение ip-адресов, называют нынешнюю эпоху «периодом перехода к IPv6», но можно усомниться, что всеобщий переход на IPv6 когда-нибудь в действительности произойдёт. К примеру, если всё-таки начать использование адресов с первым байтом 240-255 (а технические сложности, которые при этом возникнут, не идут ни в какое сравнение со сложностями перехода на IPv6), то это даст ещё 16 блоков размера /8, которых при нынешней черепашьей скорости раздачи адресов может хватить лет на сто. Кроме того, большое количество уже выданных адресов сейчас не используется, поскольку когда-то они были выделены организациям, не нуждающимся в таком количестве адресов; просто «взять и отобрать» у них адреса вряд ли получится, но, вполне возможно, когда-то ip-адреса станут обычным предметом купли-продажи. Ничего хорошего это не принесёт, возникнет огромное количество проблем как технического (взрывной рост размера мировой таблицы маршрутизации из-за дробления подсетей), так и правового характера (в самом деле, как может быть объектом купли-продажи *32-битное целое число*?!), но проблема исчерпания адресного пространства при этом, скорее всего, исчезнет надолго.

Конечно, если не рассматривать сценарии мировой катастрофы, то рано или поздно адреса протокола IPv4 всё же закончатся, и закончатся совсем, полностью, когда никакая экономия уже не поможет, поскольку экономить будет нечего; но к тому времени, скорее всего, появится что-то более удачное, нежели протокол IPv6, который даже сейчас, спустя всего двадцать лет после своего появления, выглядит довольно странно. Может так случиться, что этот протокол будет признан морально устаревшим, так и не дождавшись внедрения.

6.2.4. Дейтаграммы и потоки

Следующий слой модели ISO, как мы помним, называется *транспортным*. Чтобы понять, в чём состоит его роль, нужно для начала заметить, что пакеты, передаваемые на уровне IP — это инструмент, с которым достаточно тяжело работать. Как мы уже отмечали выше, протоколы сетевого уровня, в том числе IP, не гарантируют доставку пакета. Отправленный пакет может потеряться или, наоборот, прийти в двух экземплярах, а пакеты, отправленные раньше других, могут прийти к получателю позже. Содержимое такого пакета обычно называют ***дейтаграммой***. С точки зрения процесса, который пытается обменяться информацией с партнёром через компьютерную сеть, дейтаграммное взаимодействие выглядит так: берём некую порцию данных (например, содержимое какого-то массива или структуры, либо часть такого содержимого), надписываем адрес получателя и просим операционную систему позаботиться о пересылке. Поскольку пакет может не

дойти, адресат в ответ на полученный пакет обязательно должен отправить подтверждение; в свою очередь, отправитель должен некоторое время ожидать получения подтверждения, а если такового не поступит, снова отправить ту же самую дейтаграмму, предполагая, что первая потерялась. Для экономии пропускной способности сети подтверждения можно отправлять не на каждый пакет, а сразу на некоторую их последовательность, причём в подтверждении можно указать, какие пакеты были получены, а какие потерялись. Всю эту механику приходится реализовывать прямо в программах, которые будут взаимодействовать друг с другом через сеть; всё, что предоставляет нам операционная система (содержащая реализацию протокола передачи дейтаграмм) — это возможность послать дейтаграмму, указав адрес её получателя.

Ситуация существенно осложняется, когда дейтаграммы на самом деле представляют собой фрагменты одного информационного объекта — например, файла. Поскольку при доставке дейтаграмм может нарушиться их порядок, нужно, чтобы отправитель в каждом пакете указывал его номер или как-то иначе обозначал место этой дейтаграммы в составе целого, а получатель на своей стороне должен восстановить мозаику фрагментов, при необходимости запросив у отправителя повторную отправку тех пакетов, которые потерялись по дороге.

Если программистов устраивает такой режим работы, то от протокола транспортного уровня им нужно совсем немного. Когда в качестве сетевого протокола используется IP, как мы уже знаем, контрольная сумма, содержащаяся в заголовке его пакета, позволяет проверить целостность заголовка, но не всего пакета; для проверки правильности передачи «полезной нагрузки» (в данном случае — дейтаграммы) нужно где-то хранить контрольную сумму для дейтаграммы. Кроме того, на одном компьютере может работать одновременно много разных программ, желающих отправлять и принимать дейтаграммы, и их нужно как-то между собой различать; сам компьютер можно идентифицировать по сетевому адресу.

Соглашения о том, как решить эти вопросы, содержатся в протоколе UDP (*user datagram protocol*). Это один из самых простых протоколов в Интернете, поскольку почти всю работу уже сделал за него протокол IP. UDP предусматривает, что участники сетевого взаимодействия, работающие на одном компьютере, будут для идентификации использовать двухбайтные беззнаковые целые числа, называемые ***номерами портов*** или просто ***портами***. К данным, которые нужно передать через сеть, протокол UDP предписывает добавить заголовок из 8 байт, содержащий номер порта отправителя, номер порта получателя, длину дейтаграммы (данные вместе с заголовком) и контрольную сумму; полученная структура (дейтаграмма) «заворачивается» в пакет в соответствии с соглашениями протокола IP. Сетевые адреса отправителя и получателя указываются уже в заголовке этого пакета.

Как видим, дейтаграммное взаимодействие оказывается очень простым в реализации, но довольно сложным для использования. Со вторым основным видом транспортного взаимодействия — *потоковым* — дела обстоят прямо противоположным образом. С точки зрения программ, взаимодействующих таким способом, сетевое соединение выглядит как обычный поток ввода-вывода, к которому можно применять хорошо знакомые нам системные вызовы `read` и `write`. Информацию, записанную в один конец сетевого соединения, можно после доставки её через сеть прочитать из другого конца, и обратно, то есть поток в данном случае оказывается двухсторонним. Операционная система гарантирует, что все байты, записанные в поток, будут затем доступны для чтения на другом конце потока, причём их порядок будет сохранён; при невозможности соблюдения этой гарантии соединение окажется разорвано, о чём узнают оба партнёра — очередные системные вызовы вернут им ошибки.

Очевидно, что обмениваться информацией через такое соединение гораздо проще и удобнее, чем с помощью дейтаграмм. Решение всех проблем, связанных с потерями пакетов, восстановлением данных из отдельных фрагментов и т. д. берёт на себя реализация транспортного протокола, которая, конечно же, оказывается гораздо сложнее реализации протокола дейтаграммного. Соответствующий транспортный протокол, используемый в Интернете, называется TCP (*transmission control protocol*, т. е. «протокол управления передачей [данных]»). Точно так же, как и UDP, протокол TCP использует *номера портов* для идентификации участников взаимодействия, работающих на одном компьютере, но сходство на этом заканчивается; достаточно сказать, что заголовок пакета для TCP имеет переменную длину и может занимать от 16 до 56 байт. В отличие от работы с дейтаграммами, при потоковой работе необходимо сначала *установить соединение*, для чего один из будущих партнёров должен находиться в состоянии ожидания запроса на соединение, а второй должен этот запрос послать; естественно, партнёры должны уметь подтверждать получение отдельных пакетов, а при отсутствии такого подтверждения — посылать недостающие пакеты повторно; кроме того, в зависимости от целого ряда условий партнёры по соединению выбирают *размер окна передачи*, то есть количество данных, передаваемых в каждом пакете, и этот размер может изменяться в ходе работы. Протокол TCP предусматривает, кроме перечисленного, ещё целый ряд нетривиальных соглашений, пояснение которых заняло бы слишком много места. К счастью для прикладных программистов, всю реализацию соглашений протокола TCP берёт на себя операционная система.

6.2.5. Протоколы прикладного слоя

Оставшиеся три уровня модели OSI — сеансовый, представительный и прикладной — вычленить из реально существующих протоколов будет несколько сложнее.

Интереснее всего обстоят дела с сеансовым уровнем. Наиболее популярно мнение, что в Интернете протоколы этого уровня попросту отсутствуют; это звучит несколько неожиданно, поскольку сеансы работы как таковые, вне всякого сомнения, не только присутствуют, но и постоянно используются. Проблема лишь в том, что организуются они либо средствами протоколов прикладного уровня, либо соглашениями, имеющими *ещё более высокий уровень*. В модели OSI не предусмотрено слоя, находящегося выше прикладного; ну а в Интернете невозможно найти такие протоколы, которые занимались бы организацией сеансов и при этом находились *между* транспортным слоем (TCP/UDP) и слоем прикладным.

Хрестоматийным примером сеанса, организованного поверх протоколов прикладного уровня, можно считать сеанс работы с веб-сайтом, предусматривающим аутентификацию пользователя. Протоколы HTTP и HTTPS, используемые для взаимодействия браузера и сервера, предусматривают загрузку каждой новой версии страницы, а также дополнительных элементов — изображений, стилевых файлов и т. п. — с использованием новых соединений; одно соединение может использоваться, а может и не использоваться для обслуживания нескольких запросов, идущих подряд, но, так или иначе, сеанс работы с информационной системой через web-интерфейс заведомо состоит более чем из одного соединения, предусмотренного протоколами. Логическое объединение этих соединений в один сеанс работы с пользователем достигается путём сохранения идентификатора сеанса либо в файле cookie на стороне браузера, либо в виде параметра в адресной строке; очевидно, что эти механизмы работают *поверх* протоколов HTTP/HTTPS, то есть находятся на уровне более высоком, а не менее высоком, как того требует модель OSI.

Поскольку протокол есть набор соглашений, мы можем отнести к числу протоколов кодировки текста (ASCII, koi8r, utf-8 и другие), а также форматы данных и правила их интерпретации — например, спецификации разнообразных версий HTML/XHTML, графических форматов JPEG, GIF, PNG и прочее в таком духе (ведь всё это тоже наборы соглашений). Кроме того, стоит упомянуть используемый в Интернете набор спецификаций под общим названием MIME, который исходно предназначался для электронной почты, но со временем проник и в другие коммуникационные среды. MIME позволяет представлять в виде ASCII-текста файлы различных типов и наборы файлов, задаёт классификацию форматов и правила их идентификации; когда мы отправляем по электронной почте письма с вложенными файлами, почтовые программы используют для этого как раз соглашения MIME. Если считать соглашения о форматах данных протоколами, то они будут,

очевидно, относиться к представительному уровню, хотя создатели модели OSI имели в виду не совсем это.

Наконец, с прикладным уровнем всё оказывается достаточно просто: на этом уровне организуется взаимодействие программ, непосредственно обеспечивающих решение информационных задач для конечного пользователя, в том числе, например, таких хорошо известных систем, как электронная почта, «всемирная паутина» (World Wide Web, или WWW), службы передачи моментальных сообщений и т. п.

Большинство сервисов Интернета построено в соответствии с так называемой ***клиент-серверной моделью***, что, по-видимому, нуждается в дополнительных пояснениях. **Под *сервером* понимается программа (или, в широком смысле слова, любое «действующее лицо» — набор взаимосвязанных программ, компьютер, система в целом и т. п.), ожидающая запросов и производящая какие-либо действия исключительно в ответ на запросы, а при отсутствии запросов не делающая ничего**. Как несложно догадаться, **под *клиентом* понимается программа (или другой участник взаимодействия), обращающаяся с запросом к серверу**. Вообще говоря, одна и та же программа может выполнять функции сервера по отношению к одним программам и клиента — по отношению к другим, если для удовлетворения запроса клиента серверу необходимо воспользоваться услугами другого сервера.

Многие протоколы прикладного уровня (хотя, конечно, не все) основываются на обмене *текстовыми* репликами через двунаправленное потоковое соединение. Именно таковы, в частности, протоколы SMTP (*simple mail transfer protocol*, протокол передачи электронной почты) и HTTP (*hypertext transfer protocol* — протокол, по которому браузер связывается с web-сайтами). Текстовый формат этих протоколов позволяет при желании связаться с сервером «вручную» и «поговорить» с ним, используя любую программу, позволяющую установить потоковое соединение и передавать туда-обратно простой текст; например, для этого вполне подойдёт программа `telnet`, входящая в стандартную комплектацию практически любой Unix-системы, а до недавних пор (вплоть до Windows XP) входившая также и в состав систем линейки Windows.

Приведём пример сеанса отправки электронного письма от имени пользователя `test@stolyarov.info` получателю, имеющему адрес `test123@unicontrollers.com`:

```
=   bash-3.1$ telnet 195.42.170.131 25
=   Trying 195.42.170.131...
=   Connected to 195.42.170.131.
=   Escape character is '^]'.
    220 pluto.unicontrollers.net ESMTP Postfix
>   EHLO reptile.croco.net
```

```
    250-pluto.unicontrollers.net
    250-PIPELINING
    250-SIZE 10240000
    250-ETRN
    250-ENHANCEDSTATUSCODES
    250-8BITMIME
    250 DSN
>   MAIL FROM: <test@stolyarov.info>
    250 2.1.0 Ok
>   RCPT TO: <test123@unicontrollers.com>
    250 2.1.5 Ok
>   DATA
    354 End data with <CR><LF>.<CR><LF>
>   From: Andrey Stolyarov <test@stolyarov.info>
>   To: Test Account <test123@unicontrollers.com>
>   Subject: test message
>
>   This is a test
>   .
    250 2.0.0 Ok: queued as 1549E481E12
>   QUIT
    221 2.0.0 Bye
=   Connection closed by foreign host.
=   bash-3.1$
```

Для наглядности мы пометили знаком «=» строки, напечатанные локальными программами — интерпретатором командной строки и программой `telnet`; эти строки не являются частью сеанса и не имеют никакого отношения к протоколу. Строки, переданные серверу (напечатанные на клавиатуре в ходе сеанса), помечены знаком «>»; все остальные строки — это ответы сервера. Попробуем разобрать, что здесь к чему.

Сразу после установления соединения сервер «представился» нам, прислав строку

```
220 pluto.unicontrollers.net ESMTP Postfix
```

Здесь `pluto.unicontrollers.net` — доменное имя компьютера в сети, аббревиатура `ESMTP` указывает на то, что сервер поддерживает протокол ESMTP (расширенную версию SMTP, буква «E» означает слово *extended*), а слово `Postfix` — это название используемой «по ту сторону» серверной программы. Некоторые программы выдают в этой строке ещё и номер своей версии, так может поступить и Postfix, если его по-другому настроить. Число `220` — это так называемый *код ответа* (*response code*); к этому моменту мы ещё вернёмся.

Первая команда, которую мы дали серверу — это команда `EHLO` с параметром, который означает доменное имя нашей (передающей) машины, сейчас выступающей в роли клиента. В ранних версиях протокола команда называлась `HELO` и была образована от английского слова

Hello; используя команду `EHLO` вместо `HELO`, мы извещаем сервер, что готовы к использованию поздних расширений протокола. Сервер ответил на эту команду, перечислив кодовые имена расширений, которые он поддерживает; если бы мы использовали `HELO`, ответ сервера состоял бы из одной строки:

```
250 pluto.unicontrollers.net
```

поскольку именно такой ответ предусмотрен ранними спецификациями SMTP. Впрочем, если говорить совсем строго, то после кода `250` и пробела протокол допускает произвольную строку, предназначенную скорее для человека, нежели для компьютерных программ.

Так или иначе, сейчас мы используем расширенную версию протокола, и многострочный ответ сервера позволяет клиенту (в роли которого обычно выступает другая программа, а не человек, как в нашем случае) узнать, какие из расширений протокола сервер способен обработать; от этого зависит, какие команды стоит посылать, а какие нет. Впрочем, мы расширениями протокола не воспользовались; оставшиеся команды, использованные в нашем сеансе, присутствовали в SMTP с самого начала.

Сразу после обмена приветствиями с сервером мы послали ему команду `MAIL FROM:`, которая указывает, что мы сейчас намерены передать электронное письмо, отправителем которого является владелец адреса `test@stolyarov.info`:

```
MAIL FROM: <test@stolyarov.info>
```

Ответ сервера, как видим, весьма лаконичен:

```
250 2.1.0 Ok
```

Большинство программ, работающих с протоколом SMTP, из всей этой строки использует только код `250`, означающий, что последняя команда принята сервером без каких-либо замечаний. Несколько реже используется так называемый *усовершенствованный код* — цифры `2.1.0`; это уже особенность расширенной версии SMTP, точнее говоря — одного конкретного расширения, которое сервер поддерживает, о чём он заявил в ответ на наше `EHLO` (идентификатор этого расширения — `ENHANCEDSTATUSCODES`). Что касается слова `Ok`, то это уже сугубо декоративный комментарий; некоторые сервера могут в таких случаях быть более многословны, выдать, например, что-нибудь вроде «`Sender address test@stolyarov.info is welcome`»; Postfix, как видим, предпочитает краткость. Надо сказать, что команда `MAIL FROM:` допускается только одна, то есть мы не имеем права (согласно протоколу) давать вторую такую команду, пока не закончим с передачей одного письма.

Следующая команда — `RCPT TO:` — задаёт адрес *получателя*. Здесь интересны сразу два момента. Во-первых, этих команд мы можем дать сколько угодно; именно так и происходит, если наше письмо адресовано сразу нескольким получателям. Во-вторых, если команду `MAIL FROM:` сервер чаще всего пропускает (лишь некоторые сервера производят различные проверки адреса отправителя и, например, отказываются принимать почту, если домен отправителя не существует), то команда `RCPT TO:` будет успешной лишь при соблюдении довольно жёстких условий. Почтовые сервера в наше время настраиваются так, чтобы принимать почту от любого пользователя для своего пользователя либо от своего пользователя (то есть, например, пришедшего из локальной сети, обслуживаемой этим сервером) для кого угодно. Сервер `pluto.unicontrollers.net`, с которым мы установили соединение, настроен на обслуживание нескольких почтовых доменов, в том числе домена `unicontrollers.com`, и знает, что адрес `test123@unicontrollers.com` в этом домене действительно существует (на самом деле, он был создан специально для нашего эксперимента).

Примерно до 1994 года практически все почтовые сервера сети Интернет были настроены гораздо либеральнее — они были готовы от кого угодно принять почту для доставки кому угодно, а затем уже сами определяли, на какой сервер следует отправить письмо, чтобы оно достигло своего адресата. К сожалению, этим начали активно пользоваться спамеры — любители засорять чужие почтовые ящики рекламным мусором. Сообществу системных администраторов сети Интернет пришлось поменять своё отношение к настройкам серверов. Примерно к концу 1995 года в Интернете почти не осталось серверов, готовых принимать и передавать почту для кого угодно — так называемых *открытых релеев*. В наше время открытые релеи изредка появляются, но тут же оказываются внесены в разнообразные блокирующие списки, в результате чего от них отказывается принимать почту большая часть Интернета. Надо сказать, что попасть в такой «чёрный список» обычно гораздо проще, чем потом из него выбраться; если вам когда-нибудь доведётся настраивать почтовый сервер, обратите особое внимание на запрет открытой ретрансляции.

Конечно, сервер, с которым мы экспериментировали, не допускает открытой ретрансляции. Если бы в команде `RCPT TO:` мы указали адрес в домене, не входящем в число обслуживаемых этим сервером, мы бы получили отказ:

```
>  RCPT TO: <test17@croco.net>
   554 5.7.1 <test17@croco.net>: Relay access denied
```

Отказ мы бы получили также и в случае, если бы указанный нами адрес находился в «правильном» домене, но при этом сервер обнаружил, что такого адреса на самом деле нет:

```
> RCPT TO: <abrakadabra@unicontrollers.com>
  550 5.1.1 <abrakadabra@unicontrollers.com>: Recipient address
  rejected: User unknown in virtual alias table
```

(на самом деле весь ответ, начиная с кода 550, исходно был записан в одну строчку, но нам пришлось его разбить).

Сообщив серверу адрес отправителя, адреса всех получателей и получив положительные ответы на все команды, мы можем, наконец, приступить к отправке письма; делается это командой DATA. Сервер нам на неё ответил немного хитро:

```
354 End data with <CR><LF>.<CR><LF>
```

Здесь для клиентской программы интересен только код 354, означающий, что следует продолжить начатое, а комментарий предназначается для человека вроде нас и переводится так: «закончите данные последовательностью <CR><LF>.<CR><LF>». Поясним, что под <LF> понимается хорошо знакомый нам символ перевода строки (код 10), а под <CR> — символ *возврата каретки* (код 13). Как видим, протокол SMTP предполагает, что строки в текстовом потоке разделяются двумя байтами примерно так, как в текстовых файлах в системах линейки Windows; впрочем, автору не попадалось ни одной программы, которая не принимала бы поток со строками, разделёнными одним только переводом строки. Всё это означает, что текст письма следует начать сразу после команды DATA, а когда он закончится, сообщить об этом серверу, передав строку, состоящую из одного символа точки. Именно так мы и поступили; нужно только заметить, что текст электронного письма состоит из *заголовка* и *тела*, разделённых пустой строкой. Заголовок мы сформировали из трёх строк:

```
From: Andrey Stolyarov <test@stolyarov.info>
To: Test Account <test123@unicontrollers.com>
Subject: test message
```

а в качестве тела передали одну короткую фразу «This is a test», отделив её пустой строкой от заголовка. После этого мы сообщили серверу, что письмо окончено, отправив строку, состоящую из одной точки. Сервер ответил нам, что письмо принято для дальнейшей обработки:

```
250 2.0.0 Ok: queued as 1549E481E12
```

(поясним, что 1549E481E12 — это присвоенный нашему письму идентификатор в локальной очереди сообщений, ожидающих дальнейшей доставки). Интересно, что в ходе дальнейшей обработки сервер добавил к нашему письму ещё несколько строк заголовка. Вот во что превратилось наше письмо, когда оно дошло наконец до почтового ящика:

```
From test@stolyarov.info  Wed Mar 29 21:35:15 2017
Return-Path: <test@stolyarov.info>
Delivered-To: test123@unicontrollers.com
Received: from reptile.croco.net (reptile.croco.net [195.42.160.101])
        by pluto.unicontrollers.net (Postfix) with ESMTP id 1549E481E12
        for <test123@unicontrollers.com>; Wed, 29 Mar 2017 14:11:29 +0000
        (UTC)
From: Andrey Stolyarov <test@stolyarov.info>
To: Test Account <test123@unicontrollers.com>
Subject: test message
Message-Id: <20170329141142.1549E481E12@pluto.unicontrollers.net>
Date: Wed, 29 Mar 2017 14:11:29 +0000 (UTC)

This is a test
```

Вы и сами можете провести аналогичный эксперимент, отправив «вручную» письмо самому себе; нужно только узнать, какая машина обслуживает ваш домен. Для этого достаточно обратиться к глобальной распределённой базе данных DNS с помощью, например, команды `host`, которая, скорее всего, уже имеется в вашей системе. Так, машину, обрабатывающую почту для `unicontrollers.com`, можно установить с помощью следующей команды:

```
avst@host:~$ host -t MX unicontrollers.com
unicontrollers.com mail is handled by 5 pluto.unicontrollers.net.
```

С протколом HTTP всё обстоит ещё проще. Для примера запросим главную страницу сайта `www.stolyarov.info`:

```
=  avst@host:~$ telnet www.stolyarov.info 80
=  Trying 195.42.170.129...
=  Connected to varan103.croco.net.
=  Escape character is '^]'.
>  GET / HTTP/1.1
>  Host: www.stolyarov.info
>
```

В первой строчке здесь мы видим приглашение командной строки на машине, на которой мы работаем, и команду `telnet`. Следующие три строчки нам выдала сама программа `telnet`, после чего мы ввели *заголовок запроса*, состоящий в нашем случае из двух строк:

```
GET / HTTP/1.1
Host: www.stolyarov.info
```

Чтобы показать, что заголовок запроса окончен, мы передали серверу пустую строку. Ответ сервера при этом довольно многословен:

```
HTTP/1.1 200 OK
Date: Wed, 29 Mar 2017 18:07:52 GMT
```

и так далее; весь его мы приводить не будем. Сначала в ответе идёт заголовок, содержащий разнообразную служебную информацию, затем пустая строка, а после неё — собственно HTML-код главной страницы сайта. В отличие от SMTP, где сервер присылает «приветственную» строку сразу после установления соединения, протокол HTTP предполагает, что обмен по установленному каналу начнёт клиент; именно это мы и сделали, послав сначала команду `GET` с указанием локального пути к нужной странице и версии протокола, а потом дополнительную строку `Host`, позволяющую серверу понять, к какому из обслуживаемых сайтов мы хотим обратиться. Получив заголовок целиком, сервер ответил на запрос, прислав нам сначала служебную информацию, а затем и текст нужного нам документа — главной страницы сайта.

У двух рассмотренных нами протоколов совершенно неожиданно обнаруживается общая (точнее, похожая) система трёхзначных *кодов результата*. Поясним, что коды, начинающиеся с двойки, означают успешное выполнение запроса; коды, начинающиеся с четвёрки, обозначают «устранимые» ошибки (запрос невозможно выполнить прямо сейчас, но, возможно, через некоторое время ситуация станет более благоприятной); а коды, начинающиеся с пятёрки, сигнализируют о фатальных ошибках, таких, которые точно никуда не денутся, во всяком случае без вмешательства администраторов сервера. Коды, начинающиеся с единицы, используются только в HTTP и означают, что сервер предлагает клиенту продолжать начатое; коды, начинающиеся с тройки, в HTTP означают, что запрошенную информацию следует искать где-то в другом месте; в SMTP используется только один код, начинающийся с тройки (`354`), он выдаётся в ответ на команду `DATA` и означает, что сервер теперь ожидает получения тела электронного письма.

6.2.6. Доменные имена

Ранее мы обсуждали ip-адреса, записываемые цифрами, которые используются для идентификации хостов в Интернете. Несомненно, читатель знает из своего опыта, что для конечного пользователя Интернета необходимости работы с такими адресами практически никогда не возникает, вместо них используются ***доменные имена***, такие как `www.stolyarov.info`, которые существенно проще запомнить. Такие же имена мы использовали в примерах работы по протоколам SMTP и HTTP в предыдущем параграфе.

Использование доменных имён становится возможно благодаря функционированию в Интернете так называемой ***Системы доменных имён*** (*domain name system*, DNS). Часто можно услышать, что её функция состоит в преобразовании символических доменных имён в ip-адреса, что в принципе верно (можно даже считать, что это *основ-*

ная функция DNS), но реальность, как водится, устроена несколько сложнее.

Система доменных имён представляет собой глобальную распределённую базу данных, в которой иерархия доменных имён, записываемых через точку, используется в качестве ключа для поиска; информация, связанная с конкретным доменным именем, совершенно не обязана ограничиваться ip-адресом, структура базы данных позволяет хранить записи различных типов, в том числе простые текстовые сообщения.

Само по себе слово «домен» (*domain*) переводится как «множество» или «область», но в достаточно специфических контекстах. Например, область определения и область значений для математической функции по-английски называются *domain* и *codomain*. Биологи для систематизации живых организмов используют самую верхнюю ступень классификации — делят все живые организмы на три *домена*: археи, бактерии и эукариоты; к последним относятся все организмы, клетки которых имеют клеточное ядро, в том числе растения, животные и грибы (это уже так называемые *царства*). Слово *домен* в данном случае прямо заимствовано из английского, где эта ступень классификации называется точно так же — *domain*.

В системе доменных имён собственно *доменное имя* — это последовательность разделённых точками *токенов*, каждый из которых представляет собой непустую последовательность латинских[13] букв, цифр и символа дефиса, причём начинаться и заканчиваться дефисом токен не может. Например, доменное имя `www.stolyarov.info` состоит из трёх токенов: `www`, `stolyarov` и `info`. В качестве других примеров токенов перечислим `zx19`, `25ab`, `25-a-b`, `a`, `zzzzzz`, `777` и т. п. Количество токенов в доменном имени называется *уровнем* этого имени: так, `www.stolyarov.info` — это имя третьего уровня, `stolyarov.info` — второго, а `info` — первого.

С каждым доменным именем DNS позволяет связать произвольное количество *записей*, каждая из которых имеет служебный параметр *time to live* (количество секунд, в течение которого с момента получения запись можно продолжать считать актуальной), а также *класс*, *тип* и собственно содержимое. Класс записей в DNS реально используется всего один — `IN` (от слова *Internet*), хотя в спецификациях упоминается ещё по меньшей мере два других класса и два служебных значения, которые классами не являются, но якобы должны использоваться вместо класса. С типами записей ситуация интереснее. Самым очевидным типом DNS-записи можно считать тип `A` (от слова *address*): такая запись содержит адрес протокола IPv4, который должен быть поставлен в соответствие данному доменному имени. Для ip-адресов протокола IPv6

[13] Читатель может заметить, что в наше время домены бывают и кириллическими. Это не совсем верно: русскоязычные домены в «зоне .рф» на самом деле тоже представляют собой последовательности латинских букв, цифр и дефиса, получаемые из русских слов по хитрым правилам.

используются записи типа `AAAA`, для указания серверов электронной почты — записи `MX` (*mail exchanger*), что, кстати, позволяет создавать доменные имена, используемые только для почты. Для указания местонахождения серверов некоторых других протоколов можно использовать записи типа `SRV`, для указания обычных текстовых данных — записи типа `TXT`. Есть и другие типы записей; с некоторыми из них мы столкнёмся позже.

Суффиксами доменного имени называют доменные имена, полученные из исходного отбрасыванием нуля или более токенов слева. Так, доменное имя `www.stolyarov.info` имеет три суффикса: `www.stolyarov.info`, `stolyarov.info` и просто `info`. **Доменные имена, имеющие общий заданный суффикс, как раз и образуют домен**; иначе говоря, домен — это (бесконечное) множество доменных имён, имеющих некий заданный общий суффикс. Например, в домен `stolyarov.info` входят реально используемые имена `stolyarov.info` и `www.stolyarov.info`, но могут также входить и другие имена, например, `test.stolyarov.info`, `mars.stolyarov.info`, `z12.class.stolyarov.info` и т. п. Следует отметить, что домены, в полном соответствии с приведённым определением, *вкладываются друг в друга*: например, имя `z12.class.stolyarov.info` входит как в домен `class.stolyarov.info`, так и в домен `stolyarov.info`, и в домен первого уровня `info`. Можно сказать, что домен `class.stolyarov.info` является подмножеством (*поддоменом*) домена `stolyarov.info`, при этом оба они представляют собой поддомены домена `info`, а он, в свою очередь, считается поддоменом ***корневого домена***. Корневой домен обозначается точкой «`.`» и объединяет все существующие домены и доменные имена.

Иерархичность доменов — это как раз то свойство, которое позволяет глобальной базе данных DNS быть полностью распределённой, обслуживаться сотнями тысяч DNS-серверов, находящихся в совершенно разных руках, и при этом избегать появления «бутылочных горлышек» — таких мест, через которые проходило бы слишком много запросов и которые было бы тяжело «расширить». Обслуживание конкретного домена может быть поручено двум или более[14] серверам; такие сервера называются *авторитативными* для данного домена. Обычно один из авторитативных серверов выполняет функции «главного» (*master*), а остальные — функции «подчинённых» или «ведомых» (*slaves*). Время от времени ведомые запрашивают у главного актуальную версию информации, относящейся к обслуживаемому домену. В случаях, когда информация изменилась на главном сервере, он сам оповещает об этом подчинённых, чтобы они не ждали следующего планового момента обновления, но

[14] Теоретически можно использовать и один сервер, но из соображений надёжности так делать не рекомендуется, ведь любой компьютер хотя бы иногда приходится перезагружать.

```
@       IN      SOA     ns2.croco.net. netop.croco.net. (
                        4
                        28800
                        7200
                        604800
                        86400   )
        IN      NS      ns2.croco.net.
        IN      NS      ns3.croco.net.
        IN      NS      ns.intelib.org.
@       IN      MX 5    reptile.croco.net.
@       IN      A       195.42.170.129
www     IN      CNAME   varan103.croco.net.
```

Рис. 6.7. Файл доменной зоны `stolyarov.info`

даже если оповещение (представляющее собой одну дейтаграмму) до адресата не дойдёт — это не так страшно, просто информация будет обновлена чуть позже.

Администратор домена может принять решение *делегировать* обслуживание поддомена своего домена другой группе серверов. В этом случае за обслуживание запросов по доменным именам, входящим в такой поддомен, будет отвечать уже новая группа серверов, а не та, которая обслуживает исходный домен. Домен за вычетом поддоменов, делегированных на другие сервера, называется ***доменной зоной***. Правильнее говорить, что авторитативные DNS-сервера обслуживают не домен как таковой, а *доменную зону*.

Доменная зона обычно формируется в виде текстового файла, содержащего *записи*. Из этих записей состоит база данных DNS, а функционирование DNS как системы состоит в том, чтобы по заданному доменному имени найти нужный авторитативный сервер и получить от него запись нужного типа либо несколько таких записей — большинство типов допускают множественность. Пример файла доменной зоны приведён на рис. 6.7. Поясним, что символом «`@`» обозначается сам домен, для которого написан файл зоны, а все прочие доменные имена в файле зоны считаются относительными (то есть к ним добавляется имя текущего домена), если только в конце не поставить точку. Собственно говоря, в рассматриваемом файле точка стоит в конце всех имён, кроме «`www`» (поскольку здесь имеется в виду `www.stolyarov.info`). Загадочное `IN` в каждой строке указывает на *класс* записей (напомним, что класс `IN` — единственный, который реально используется в DNS).

Запись `SOA` (*start of authority*) означает, что для данного имени имеется отдельный файл зоны (в нашем случае это так и есть). Записи `NS` (*name server*) указывают доменные сервера, обслуживающие данный домен; на самом деле набор серверов, обслуживающих домен, задаётся записями в вышестоящей зоне (в данном случае — в зоне `info`), и технически используются именно записи `NS` из вышестоящей

зоны, но считается хорошим тоном поддерживать в своей собственной зоне тот же набор NS-записей. Запись `MX` задаёт машину, обрабатывающую электронную почту для данного доменного имени, запись `A` (для имени `@`, что означает `stolyarov.info`) указывает ip-адрес, который соответствует доменному имени `stolyarov.info`, а запись `CNAME` для имени `www` (то есть `www.stolyarov.info`) предписывает рассматривать это имя как *синоним* другого имени (на самом деле это другое имя — `varan103.croco.net` — тоже преобразуется в тот же самый ip-адрес).

Упомянем ещё несколько интересных возможностей файлов доменных зон. Вместо младшего токена в доменном имени можно указать звёздочку «`*`»; это будет означать, что указанную запись следует отдавать в ответ на запросы, в которых вместо звёздочки может стоять что угодно. Так, если бы мы внесли в наш примерный файл зоны запись вроде

```
*.camp      IN    A    198.51.100.72
```

то ip-адрес `198.51.100.72` система DNS стала бы отдавать в ответ на запросы `www.camp.stolyarov.info`, `vasya.camp.stolyarov.info`, `abrakadabra.camp.stolyarov.info` и т. п.

Для делегирования поддомена достаточно занести в зону исходного домена соответствующие записи. Например, чтобы делегировать домен третьего уровня `example.stolyarov.info` серверам `ns1.example.com` и `ns2.example.com`, достаточно в доменную зону `stolyarov.info` внести такие строки:

```
example     IN    NS   ns1.example.com
example     IN    NS   ns2.example.com
```

С делегированием поддоменов связана довольно забавная проблема, неизменно вызывающая трудности у начинающих. Если доменное имя (домен $N+1$-го уровня) делегируется на DNS-сервера, имена которых сами располагаются в делегируемом поддомене, то для этих серверов необходимо наличие записей типа `A` в вышестоящей зоне. Так, домен `stolyarov.info` делегирован на DNS-сервера, имена которых расположены за его пределами; если бы его нужно было делегировать на сервера, имена которых сами находятся в этом домене — например, на сервера с именами `ns1.stolyarov.info` и `ns2.stolyarov.info` — то для этих имён пришлось бы в доменной зоне `info` расположить записи типа `A`. Объясняется это очень просто: *чтобы обратиться к DNS-серверам, необходимо знать, на каких ip-адресах они находятся*; если бы в вышестоящей доменной зоне не было соответствующих записей, обращение к авторитативным серверам для делегированного поддомена оказалось бы невозможным. Такие записи называются *склеивающими* (*glue records*).

Регистрируя доменные имена, мы чаще всего имеем дело с доменами *второго уровня*. Домены первого уровня, такие как `info`, `com`, `net`,

`ru` и т. п, *тоже делегируются*; их делегирование выполняется на так называемых *корневых серверах*, обслуживающих *корневой домен* (точнее, *корневую зону*). Текущий список корневых серверов — это некая общедоступная информация, необходимая для первичной настройки любого DNS-сервера. К счастью, изменения в список корневых серверов вносятся достаточно редко, причём если один из серверов меняет свой адрес, остальные ещё несколько лет остаются работать на прежних адресах, так что даже если администратор DNS-сервера не уследил за новостями, его сервер долгое время не утратит работоспособности.

Сейчас корневой домен обслуживают 13 «логических» серверов: `a.root-servers.net`, `b.root-servers.net`, ..., `m.root-servers.net`. Каждому из этих имён приписан как адрес протокола IPv4, так и адрес протокола IPv6. При этом все эти сервера, за исключением одного (под буквой `b`), на самом деле представляют собой целое множество «реплик» — физических компьютеров, размещённых в самых разных точках мира. Так, на момент написания этого текста в Москве были размещены реплики корневых серверов «`F`», «`J`», «`K`» и «`L`», в Санкт-Петербурге — реплики «`I`», «`J`» и «`K`», а больше всего реплик по всему миру — 158 — поддерживалось для сервера «`L`».

Конечно, если бы каждый DNS-запрос проходил через корневые сервера, Интернет не спасли бы никакие ухищрения — с такой нагрузкой не справились бы и десятки тысяч машин. К счастью, запросы, обращённые к корневым серверам — на самом деле не такое уж частое событие. Дело в том, что каждый DNS-сервер, обслуживающий пользователей, *запоминает* все записи, которые через него проходят, и помнит их, пока для них не истечёт установленное время жизни *time-to-live*. Если DNS-серверу потребовалась информация в каком-нибудь домене второго уровня, кончающемся на `com`, то ему придётся обратиться к корневым серверам, чтобы узнать, какие сервера обслуживают домен `com`; время жизни для этих записей установлено в 48 часов, так что эта информация останется в памяти сервера надолго: если не перезапускать его, то следующее обращение к корневым серверам за списком серверов домена `com` раньше чем через двое суток не произойдёт. То же самое можно сказать и про другие домены первого уровня, ну а самих этих доменов не так много — всего несколько сотен, и обычно пользователи одного отдельно взятого DNS-сервера к большинству доменов первого уровня не обращаются.

6.3. Система сокетов в ОС Unix

Две предыдущие главы позволили нам составить общее впечатление о компьютерных сетях; теперь настало время вернуться к интерфейсу

системных вызовов ОС Unix и научиться писать программы, использующие сеть для взаимодействия друг с другом и с другими программами.

Системы семейства Unix (а в современных условиях — вообще все существующие операционные системы) предоставляют пользовательским задачам доступ к возможностям сети через так называемые ***сокеты*** или, говоря строже, через системные вызовы, спецификация которых использует понятие сокета. Изучением таких системных вызовов мы и займёмся в этой главе.

6.3.1. Семейства адресации и типы взаимодействия

Дать строгое определение сокета достаточно сложно. Ограничимся замечанием, что сокет (англ. *socket*) — это объект ядра операционной системы, через который происходит сетевое взаимодействие[15]. На сокет как абстрактный объект имеются две совершенно разные точки зрения. Процесс, создавший (или использующий) сокет для связи с другими процессами, видит его как нечто, «живущее» в ядре ОС и определённым образом реагирующее на системные вызовы. С другой стороны, все остальные участники сетевого взаимодействия воспринимают сокет как этакую (естественно, сугубо абстрактную) «амбразуру», видимую извне системы, на которой написан определённый адрес и через которую можно взаимодействовать с кем-то, работающим внутри системы.

В ОС Unix с сокетом связывается файловый дескриптор; в частности, работа с сокетом может быть завершена обычным вызовом `close`. Для идентификации сокетов (или, точнее, абонентов связи) в сетях используются ***адреса***. В зависимости от используемых протоколов адреса могут выглядеть совершенно по-разному. Так, обсуждая в §6.2.4 идентификацию абонентов сетевого взаимодействия при использовании протоколов UDP и TCP, мы ввели понятие *сетевого порта*, который позволяет идентифицировать участников взаимодействия внутри одной системы, тогда как сама система идентифицируется ip-адресом. Эта пара — ip-адрес плюс номер порта — как раз и составляет сетевой адрес для сокета при использовании протоколов семейства IP.

Очевидно, что если произойдёт чудо и Интернет всё же перейдёт на протокол IPv6, то для представления сетевых адресов потребуются совершенно иные структуры данных: в самом деле, для ip-адреса нужно будет уже не 4 байта, а 16. С другой стороны, в сетях, построенных по технологии компании Novell и вышедших из употребления сравнительно недавно (поддержка этих протоколов для Linux и BSD существует до сих пор), использовался стек протоколов IPX/SPX. В рамках этих протоколов адрес сокета состоит из трех частей: 4-байтного номера

[15] Это нельзя считать определением сокета хотя бы по той причине, что основанное на сокетах взаимодействие не обязательно происходит по сети, а взаимодействие по сети бывает основано не только на сокетах — во всяком случае, раньше существовали и другие программные интерфейсы для работы с сетью, такие как STREAMS.

сети, 6-байтного номера машины (хоста) и 2-байтного номера сокета. Существуют и другие семейства протоколов. Кроме того, отдельный специальный вид сокетов предназначен для связи процессов в рамках одной машины; в качестве адресов такие сокеты используют имена специальных файлов в файловой системе.

Несмотря на такие различия, между разными видами взаимодействия очень много общего. В любом случае было бы категорически неприемлемо модифицировать интерфейс ядра операционной системы и, соответственно, переделывать всё прикладное программное обеспечение при добавлении поддержки очередного семейства протоколов. Именно поэтому введена подсистема сокетов — своего рода общий знаменатель для всех возможных видов сетевого взаимодействия процессов. При создании сокета указывается, к какому ***семейству адресации*** (англ. *address family*)[16] новый сокет будет принадлежать. Набор поддерживаемых семейств может быть расширен добавлением соответствующих модулей в ядро и написанием прикладных программ, работающих с новыми адресами; при этом системные вызовы останутся прежними, а значит, не придётся переделывать системные библиотеки и программы, не использующие новые протоколы.

Кроме используемого семейства адресации, при создании сокета нужно указать ***тип взаимодействия***. Обсуждая в §6.2.4 протоколы транспортного уровня, мы упомянули два основных типа взаимодействия — *дейтаграммный* и *потоковый*; их мы и будем рассматривать. В принципе ядра операционных систем (в том числе Linux и FreeBSD) обычно поддерживают сокеты с другими типами взаимодействия, но некоторые из них недоступны в семействе протоколов TCP/IP (то есть в большинстве современных компьютерных сетей), другие оказываются особенностью конкретного ядра (например, *raw sockets* в ОС Linux), а их рассмотрение потребовало бы рассказа о достаточно специфических свойствах системы.

Напомним, что при *дейтаграммном взаимодействии* на сокете доступны две основные операции: передача и приём пакета данных (*дейтаграммы*), причём размер пакета, вообще говоря, ограничен; в частности, для UDP ограничение максимальной длины дейтаграммы обусловлено максимальной длиной пакета, передаваемого через сетевые соединения (так называемый *maximal transfer unit*, MTU), которое в большинстве случаев составляет 1500 байт, что, за вычетом длины заголовков IP и UDP, оставляет на саму дейтаграмму 1432 байта. С другой стороны, никто не гарантирует, что MTU на всех сетевых соединениях, которые встретятся на пути вашего пакета, будет составлять именно 1500 байт. В RFC791, определяющем протокол IPv4, содержится рекомендация для всех сетевых хостов принимать пакеты длиной хотя бы 576 байт

[16] Часто используется также термин ***семейство протоколов*** (англ. *protocol family*). Это синонимы.

(«октетов», т. е. групп по восемь бит), что оставляет на дейтаграмму 508 байт; дейтаграммы бóльшего размера рекомендовано отправлять только при наличии веских оснований для предположения о достаточности размера MTU по всему пути следования пакета. Переданный пакет может быть потерян или, наоборот, случайно сдублирован, то есть получено будет два или более одинаковых пакета; два переданных пакета могут прийти получателю в обратном порядке. Как уже говорилось, при таком режиме работы обеспечение надёжности ложится на пользовательскую программу (приложение).

Потоковый тип взаимодействия предоставляет прикладному программисту иллюзию надёжного двунаправленного канала передачи данных. Данные могут быть записаны в канал порциями любого размера; гарантируется, что на другом конце данные либо будут получены без потерь и в том же порядке, либо не будут получены вообще: соединение в этом случае будет разорвано с фиксацией ошибки. При использовании потокового взаимодействия заботу о передаче подтверждений, о расстановке пакетов в исходном порядке, о повторной передаче потерянных пакетов и т. п. берёт на себя операционная система.

Сокет в ОС Unix создаётся с помощью системного вызова, который так и называется `socket`:

```
int socket(int family, int type, int protocol);
```

Параметр `family` задаёт используемое семейство адресации. Мы будем рассматривать два из них: `AF_INET` для взаимодействия по сети посредством протоколов TCP/IP (адрес сокета в этом случае представляет собой пару ip-адрес/порт) и `AF_UNIX` для взаимодействия в рамках одной машины (в этом случае адрес сокета представляет собой имя файла). Параметр `type` задаёт тип взаимодействия; константа `SOCK_STREAM` соответствует потоковому взаимодействию, константа `SOCK_DGRAM` — дейтаграммному. Наконец, последний параметр задаёт конкретный используемый протокол. Для рассматриваемых нами двух семейств адресации и двух типов взаимодействия протокол однозначно определяется значениями первых двух параметров, так что в качестве этого параметра всегда можно указать число `0`. Можно, впрочем, указать и значение конкретного протокола: `IPPROTO_TCP` для TCP и `IPPROTO_UDP` для UDP.

Вызов возвращает `-1` в случае ошибки; в случае успеха возвращается номер файлового дескриптора, связанного с созданным сокетом.

6.3.2. Сокет и его сетевой адрес

Для того, чтобы с сокетом мог установить соединение (или отправить ему дейтаграмму) другой участник сетевого взаимодействия, нужно, очевидно, некое средство идентификации — ***адрес сокета***, который,

как уже говорилось в предыдущем параграфе, в зависимости от используемого семейства адресации может иметь существенно разный вид. Снабдить сокет адресом можно с помощью системного вызова `bind`:

```
int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

где `sockfd` — дескриптор сокета, полученный в результате выполнения вызова `socket`; `addr` — указатель на структуру, содержащую адрес; наконец, `addrlen` — размер структуры адреса в байтах.

Вызов `bind` возвращает `0` в случае успеха, `-1` в случае ошибки. Учтите, что существует множество ситуаций, в которых вызов `bind` может не пройти; например, ошибка произойдет при попытке использования привилегированного номера порта (от 1 до 1023) или порта, который на данной машине уже кем-то занят (возможно, другой вашей программой). Поэтому обработка ошибок при вызове `bind` особенно важна.

Реально в качестве параметра `addr` используется не структура типа `sockaddr`, а структура другого типа, который зависит от используемого семейства адресации. В семействе `AF_INET` используется структура `struct sockaddr_in`, умеющая хранить пару «ip-адрес + порт». Эта структура имеет следующие поля:

- `sin_family` — обозначает семейство адресации;
- `sin_port` — задаёт номер порта в *сетевом порядке байтов*, который может отличаться от порядка байтов, используемого на данной машине;
- `sin_addr` — задаёт ip-адрес; поле `sin_addr` само является структурой, имеющей лишь одно поле с именем `s_addr`, которое хранит ip-адрес в виде беззнакового четырёхбайтного целого, причём, опять же, в сетевом порядке байтов.

Попробуем понять, что тут к чему. Первое поле, которое в структуре `sockaddr_in` называется `sin_family` — это единственное общее поле для всех существующих структур, используемых для задания адреса сокета. Называется оно, впрочем, по-разному в разных структурах; это отголоски далёкой эпохи, когда в языке Си в разных структурах нельзя было использовать одинаковые имена полей. Так или иначе, в это поле всегда заносится идентификатор используемого семейства адресации; в данном случае это `AF_INET`.

Предназначение оставшихся двух полей как будто бы достаточно очевидно, ведь мы уже знаем, что при работе в сети Интернет участник сетевого соединения идентифицируется ip-адресом и ***портом***(см. §§6.2.3, 6.2.4); мы даже знаем, что ip-адрес обычно приписывается *сетевому интерфейсу*, так что некоторые компьютеры обладают более чем одним ip-адресом (хотя большинство компьютеров — именно что одним), но в любом случае количество процессов, участвующих в сетевых взаимодействиях, в общем случае может превышать количество ip-адресов,

наличествующих на данном конкретном компьютере; отсюда потребность в использовании портов — абстрактных беззнаковых двухбайтных чисел, дополняющих ip-адреса. Но что за загадочный «сетевой порядок байтов»?

Как мы знаем из вводной части (см. т. 1, §1.6.2), порядок байтов в представлении целых чисел в памяти может варьироваться от одной архитектуры к другой. Архитектуры, в которых старший байт числа имеет наименьший адрес (иначе говоря, байты расположены в *прямом* порядке), в англоязычной литературе обозначаются термином *big-endian*, а архитектуры, в которых наименьший адрес имеет младший байт, то есть порядок байтов *обратный*, — *little-endian*. Чтобы сделать возможным взаимодействие по сети между машинами, имеющими разные архитектуры, принято соглашение, что передача целых чисел по сети всегда производится в прямом (*big-endian*) порядке байтов, т. е. старший байт передается первым. Чтобы обеспечить переносимость программ на уровне исходного кода, в операционных системах семейства Unix введены стандартные библиотечные функции для преобразования целых чисел из формата данной машины (*host byte order*) в сетевой формат (*network byte order*). На машинах, архитектура которых предполагает порядок байтов, совпадающий с сетевым, эти функции просто возвращают свой аргумент, в ином случае они производят нужные преобразования. Вот эти функции:

```
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Как можно догадаться, буква `n` в названиях функций означает network (т. е. сетевой порядок байтов), буква `h` — host (порядок байтов данной машины). Наконец, `s` обозначает короткие целые, а `l` — длинные целые числа. Например, функция `ntohl` используется для преобразования длинного целого из сетевого порядка байтов в порядок байтов, используемый на данной машине. Номер порта представляет собой двухбайтное целое — такое целое, которое большинство компиляторов Си (если не все) называют «коротким целым» (`short int` или просто `short`). Итак, для заполнения поля `sin_port` нам потребуется функция `htons`; например, если мы решили запустить сервер, использующий порт `7654`, то выглядеть это будет так:

```
struct sockaddr_in addr;
/* ... */
addr.sin_port = htons(7654);
```

Отметим ещё один немаловажный момент, который нужно будет учесть, если вы решите поэкспериментировать с серверными сокетами. Как

для протокола TCP, так и для протокола UDP **номера портов от 1 до 1023 включительно считаются привилегированными**; такой порт может забрать себе только процесс, имеющий полномочия администратора (`euid == 0`). Если обычный процесс попытается задать сокету адрес с привилегированным номером порта, вызов `bind` вернёт `-1`, а `errno` получит значение `EACCES`.

Что касается ip-адреса, то, имея его текстовое представление (например, строку `"192.168.10.12"`), можно воспользоваться функцией `inet_aton` для формирования структуры типа `struct in_addr` (поле `sin_addr` структуры `sockaddr_in` имеет именно этот тип):

```
int inet_aton(const char *cp, struct in_addr *inp);
```

Здесь `cp` — строка, содержащая текстовое представление ip-адреса, а `inp` указывает на структуру, подлежащую заполнению. Функция возвращает ненулевое значение, если заданная строка является допустимой текстовой записью ip-адреса, и `0` в противном случае. Допустим, нужный нам ip-адрес содержится в строке `char *serv_ip`, а порт — в переменной `port` в виде целого числа. Тогда заполнение структуры `sockaddr_in` может выглядеть так:

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
if(!inet_aton(serv_ip, &(addr.sin_addr))) {
    /* Ошибка - невалидный ip-адрес */
}
```

Можно также воспользоваться функцией `inet_addr`, которая принимает один аргумент — строковое представление адреса, а возвращает четырёхбайтное целое число, которое можно занести в поле `sin_addr.s_addr`:

```
unsigned int inet_addr(const char *cp);
```

Отметим, что эта функция возвращает адрес уже в сетевом порядке байтов, так что никаких дополнительных преобразований не нужно.

В большинстве случаев задавать созданному нами сокету конкретный ip-адрес не обязательно, да это и не так легко сделать: как ни странно, в системах семейства Unix нет простого и переносимого способа узнать, какие ip-адреса имеются в системе. Поэтому, когда нам нужен серверный сокет, гораздо проще проинструктировать систему принимать соединения (или дейтаграммы) *на заданный порт на любом из имеющихся в системе ip-адресов*, а при отправке исходящих дейтаграмм или запросов на установление соединения использовать ip-адрес того сетевого интерфейса, через который происходит работа. Для этого поле `sin_addr` следует заполнить специальным значением `INADDR_ANY`:

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

Отметим, что на самом деле INADDR_ANY — это просто арифметический ноль, или ip-адрес 0.0.0.0. При подготовке к вызову bind в большинстве случаев ip-адрес, т. е. значение поля sin_addr, как раз и заполняют с помощью константы INADDR_ANY; функции inet_addr и inet_aton чаще используют при формировании адреса *получателя* дейтаграммы или запроса. Вообще-то применение функции htonl к арифметическому нулю не обязательно, поскольку все байты этого числа нулевые и совершенно всё равно, как их переставлять; но системные заголовочные файлы предоставляют целый набор констант с префиксом INADDR_, и многие из этих констант зависят от порядка байтов, так что лучше выработать привычку всегда применять к ним htonl.

Полезно знать, что структура sockaddr_in **описана в заголовочном файле** netinet/in.h. Как ни странно, эту информацию довольно тяжело отыскать в тексте страниц man'а.

Поскольку тип структуры struct sockaddr_in формально отличается от типа второго параметра вызова bind, при обращении к этому вызову приходится прибегать к явному приведению типа, например:

```
if(-1 == bind(sd, (struct sockaddr*)&addr, sizeof(addr))) {
    /* обработка ошибки */
}
```

В семействе AF_UNIX используется структура struct sockaddr_un, в которой можно хранить имя файла. Эта структура состоит из двух полей:

- sun_family — обозначает семейство адресации (в данном случае значение этого поля должно быть установлено в AF_UNIX);
- sun_path — массив на 108 символов, в который непосредственно записывается строка имени файла.

Эта структура описана в заголовочном файле sys/un.h. Файл, имя которого используется в качестве адреса сокета, должен иметь специальный тип «сокет»; это отдельный тип файла наряду с обычными файлами, директориями, символическими ссылками, символьными и блочными устройствами и именованными каналами. Если файла с заданным именем не существует, вызов bind попытается его создать; при этом у вашего процесса может не хватить полномочий для создания файла (например, у вас может не быть прав на запись в директорию, в которой вы пытаетесь создать сокет), что приведёт к ошибке. Следует также учитывать, что файл сокета сам по себе не исчезает при завершении работы с сокетом; его нужно удалить явным образом с помощью системного вызова unlink.

Ещё один интересный момент, связанный с сокетами, касается прав доступа к ним. В ОС Linux права доступа к сокету используются примерно так же, как и права для других типов файла; для того, чтобы связаться с сокетом, процесс должен иметь полномочия на запись в него. При этом некоторые другие системы, в том числе BSD, игнорируют права доступа, установленные на файле сокета.

Для других семейств адресации системные заголовочные файлы предоставляют свои структуры, имена которых тоже имеют вид `sockaddr_XX`. Структуры этих типов нужны не только для вызова `bind`, они используются везде, где происходит работа с адресом сокета. С некоторыми такими случаями мы познакомимся позже.

6.3.3. Приём и передача дейтаграмм

Рассмотрим работу с сокетами дейтаграммного типа. Сокет создаётся вызовом `socket` с указанием константы `SOCK_DGRAM` в качестве второго параметра. Желательно связать сокет с конкретным адресом с помощью `bind`, при этом ip-адрес можно заменить константой `INADDR_ANY`, но порт всё же задать. Если этого не сделать, то при отправке первой дейтаграммы система сама выберет один из своих адресов и портов в качестве адреса отправителя. Это не проблема, если мы собираемся сначала отправить кому-то дейтаграмму (например, в качестве запроса), а потом уже пытаться получить дейтаграмму, которая будет послана нам в качестве ответа; но если мы предполагаем, что кто-то другой может послать нам дейтаграмму по своей инициативе, а не в ответ на наш запрос, то, очевидно, наш сокет должен быть связан с конкретным адресом, и этот адрес должен быть известен нашим партнёрам.

Для передачи и приёма данных предназначены системные вызовы `sendto` и `recvfrom`:

```
int sendto(int s, const void *buf, int len, int flags,
           const struct sockaddr *to, socklen_t tolen);
int recvfrom(int s, void *buf, int len, int flags,
           struct sockaddr *from, socklen_t *fromlen);
```

В обоих вызовах параметр `s` задаёт дескриптор сокета, `buf` указывает на буфер, содержащий данные для передачи либо предназначенный для размещения принятых данных, `len` устанавливает размер этого буфера (соответственно, количество данных, подлежащих приему или передаче). Параметр `flags` используется для указания дополнительных опций; для нормальной работы обычно достаточно указать значение 0.

В вызове `sendto` параметр `to` указывает на структуру, содержащую адрес сокета, на который нужно отправить данные (то есть адрес получателя сообщения). Ясно, что используется при этом структура типа, соответствующего избранному семейству адресации: `sockaddr_in` для

`AF_INET` и `sockaddr_un` для `AF_UNIX` и т. д. Параметр `tolen` должен быть равен размеру этой структуры. Тип `socklen_t` обычно описан в системных заголовочных файлах как синоним типа `int`; при использовании `sendto` можно не обращать внимания на идентификатор `socklen_t`, но вызов `recvfrom` требует адреса переменной такого типа, и хотя это тот же самый `int`, указательные типы `socklen_t*` и `int*` компилятор формально считает различными. Во избежание лишних проблем проще описать нужную переменную с использованием типа `socklen_t`.

В вызове `recvfrom` параметр `from` указывает на структуру, в которую вызову следует записать адрес отправителя полученного пакета, т. е. этот параметр позволяет узнать, откуда пакет пришел. Параметр `fromlen` представляет собой указатель на переменную типа `socklen_t`, причём перед вызовом `recvfrom` в эту переменную следует занести размер адресной структуры, на которую указывает предыдущий параметр. После возврата из `recvfrom` переменная будет содержать количество байтов, которые вызов в итоге в эту структуру записал.

Как уже отмечалось, при использовании протокола UDP передан может быть только пакет ограниченного размера. Конкретный предельный размер дейтаграммы, вообще говоря, может оказаться различным для различных адресов получателей; если особенности решаемой задачи не позволяют удовлетвориться «гарантированным» размером дейтаграммы в 508 байт, придётся применить достаточно сложные процедуры динамического определения допустимого размера пакета; описание этих процедур выходит за рамки нашей книги. При желании читатель может самостоятельно изучить их, обратившись, например, к книге [3].

Есть ещё один достаточно важный момент, который следует учитывать при работе с дейтаграммами. Достаточно часто встречается сценарий работы, при котором наша программа отправляет кому-то дейтаграмму и затем ожидает ответа (тоже в виде дейтаграммы). «Наивная» реализация этого довольно очевидна: создать сокет, сформировать дейтаграмму, отправить её с помощью `sendto`, после чего вызвать `recvfrom`. К сожалению, такая реализация никуда не годится. Дело в том, что дейтаграммы, как уже неоднократно отмечалось, ненадёжны. Наша исходная дейтаграмма может потеряться, так и не дойдя до нашего партнёра, так что он не будет знать, что мы ждём от него ответа; либо может потеряться ответ. В обоих случаях наша программа так и останется в вызове `recvfrom`; внешне это будет выглядеть так, будто она зависла.

Правильным здесь будет более сложное решение, подразумевающее установку (тем или иным способом) ограничения на время ожидания ответной дейтаграммы (тайм-аута). Самый простой способ введения такого ограничения нам уже известен: мы можем предусмотреть в программе обработчик сигнала `SIGALRM`, после чего попросить операци-

онную систему прислать нам этот сигнал, скажем, через пять секунд (см. §5.5.2), и лишь после этого обращаться к `recvfrom`.

Существуют и другие способы ограничить время ожидания ответной дейтаграммы. В §6.4.3 мы будем рассматривать вызов `select`, который позволяет дождаться одного из нескольких предопределённых событий; в нашем случае в качестве таких событий могут выступать получение дейтаграммы на заданном сокете и истечение установленного тайм-аута.

Кроме того, возможность установки тайм-аута для `recvfrom` предусмотрена в самой подсистеме сокетов. Для этого нужно, используя системный вызов `setsockopt`, установить на уровне `SOL_SOCKET` опцию `SO_RCVTIMEO`, например, так:

```
struct timeval tv;
tv.tv_sec = 3;            /* полные секунды */
tv.tv_usec = 500000;      /* микросекунды */
setsockopt(sd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
```

Здесь мы установили для сокета `sd` тайм-аут на чтение (получение дейтаграммы) в 3,5 секунды. Подробности об этом способе можно найти, например, дав команду «`man 7 socket`». С вызовом `setsockopt` мы снова встретимся в §6.3.5.

6.3.4. Потоковые сокеты. Клиент-серверная модель

При взаимодействии с помощью потоковых сокетов нужно перед началом взаимодействия установить ***соединение***. Ясно, что если речь идет о взаимодействии процессов не только не родственных, но и, возможно, находящихся на разных машинах, один из участников взаимодействия должен быть инициатором соединения, а второй — принять соединение (согласиться на его установление). Здесь мы вновь сталкиваемся с уже знакомыми нам понятиями ***клиента*** и ***сервера***. При установлении соединения между потоковыми сокетами один процесс ожидает запроса на установление соединения, а другой инициирует такой запрос. Эти процессы с точки зрения установления соединения и называются сервером и клиентом; в частности, при работе по сети Интернет для организации взаимодействия потоковых сокетов используется протокол TCP, а соответствующие программы называются TCP-сервером и TCP-клиентом.

Чтобы начать ожидание запросов на соединение, сервер создаёт сокет соответствующего типа, связывает его с адресом и переводит в специальное состояние, называемое ***слушающим*** (англ. *listening*). На сокете, находящемся в слушающем состоянии, может быть выполнена только одна операция — ***приём соединения***. При установлении соединения ядро операционной системы, которая обслуживает программу-сервер, создаёт еще один сокет, который и будет использоваться для передачи данных по только что установленному соединению (рис. 6.8).

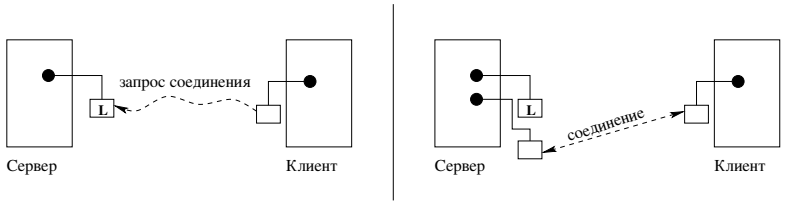

Рис. 6.8. Установление соединения между потоковыми сокетами

Итак, на стороне сервера сокет нужно создать вызовом `socket` с соответствующими параметрами и связать его вызовом `bind` с конкретным адресом, на котором будут приниматься соединения. Затем сокет следует перевести в слушающий режим (на рис. 6.8 слушающий сокет обозначен буквой `L`) с помощью вызова `listen`:

```
int listen(int sd, int qlen);
```

Параметр `sd` — связанный с сокетом файловый дескриптор. Параметр `qlen` задаёт размер очереди непринятых запросов на соединение. Поясним это. Допустим, мы перевели сокет в слушающий режим, и несколько клиентов уже отправили нам запросы на соединение, но в силу тех или иных причин мы некоторое время не принимаем эти запросы, то есть не выполняем соответствующую операцию со слушающим сокетом. Возможности системы по хранению необработанных запросов ограничены. Первые `qlen` запросов будут ожидать принятия соединения, если же система получит ещё запросы, они будут отклонены. Следует особо подчеркнуть, что параметр `qlen` не имеет никакого отношения к общему количеству соединений с клиентами.

Сказанное, впрочем, не позволяет дать ответ на очень простой и весьма актуальный вопрос: так *какое же* число следует передать вторым параметром вызова `listen`? В примерах часто встречается значение `5`, имеющее исторические причины: ядра некоторых операционных систем не позволяли создавать очередь бóльшего размера. В современных условиях такое значение может оказаться недостаточным. К счастью, если использовать «слишком большое» число, ничего страшного не произойдёт: система «молча» заменит его на наибольшее поддерживаемое. Но слишком усердствовать с этим не стоит. Многие современные системы, включая Linux, поддерживают максимально возможное значение `128`; если вашему серверу этого стало не хватать, то дальнейшее увеличение очереди вас уже не спасёт, ваш сервер работает слишком медленно, так что нужны более серьёзные изменения, нежели подкручивание размеров очереди. Всё сказанное останется верно и для существенно меньших значений: так, если вы укажете в качестве `qlen` число `16` и его в какой-то момент не хватит, это должно стать скорее поводом для поиска путей возможной оптимизации, а не для увеличения очереди.

Принятие соединения производится вызовом `accept`:

```
int accept(int sd, struct sockaddr *addr, socklen_t *addrlen);
```

Параметр `sd` задаёт дескриптор слушающего сокета. Параметр `addr` указывает на структуру для записи адреса сокета, с которым установлено соединение (иначе говоря, сюда будет записан адрес другого конца соединения). Параметр `addrlen` представляет собой указатель на переменную типа `socklen_t`, причём перед вызовом `accept` в эту переменную следует занести размер адресной структуры, на которую указывает предыдущий параметр; после возврата из `accept` переменная будет содержать количество байт, которые вызов в итоге в эту структуру записал. Это аналогично параметрам `from` и `fromlen` в вызове `recvfrom`.

Вызов `accept` возвращает файловый дескриптор нового сокета, созданного специально для обслуживания вновь установленного соединения (либо `-1` в случае ошибки). Если на момент выполнения `accept` запросов на соединение ещё не поступило, вызов блокирует вызвавший процесс и ожидает поступления запроса на соединение, возвращая управление только после того, как такой запрос поступит и соединение будет установлено. Следует отметить, что с момента принятия первого соединения в программе-сервере имеется два дескриптора, требующих обработки: дескриптор слушающего сокета, на котором можно принимать новые запросы на соединения, и сокет, соответствующий принятому соединению, с которого требуется читать пришедшие от клиента данные (например, текст запроса). Это создаёт ***проблему очерёдности*** дальнейших действий, которой будет посвящена следующая глава.

Клиентская программа должна, как и сервер, создать сокет вызовом `socket`. Связывать сокет с конкретным адресом в этом случае не обязательно, хотя и возможно; если этого не сделать, система выберет адрес автоматически. Запрос на соединение формируется вызовом `connect`:

```
int connect(int sd, struct sockaddr *addr, int addrlen);
```

Параметр `sd` — связанный с сокетом файловый дескриптор. Параметр `addr` указывает на структуру, содержащую адрес сервера, т. е. адрес слушающего сокета, с которым мы хотим установить соединение. Естественно, используется при этом структура типа, соответствующего избранному семейству адресации: `sockaddr_in` для `AF_INET`, `sockaddr_un` для `AF_UNIX`. Параметр `addrlen` должен быть равен размеру этой структуры. Вызов возвращает `0` в случае успеха, `-1` в случае ошибки.

После успешного установления соединения для передачи по нему данных можно использовать уже известные нам вызовы `read` и `write`, считая дескрипторы соединённых сокетов обычными файловыми дескрипторами; на стороне сервера это дескриптор, возвращённый вызовом `accept`, на стороне клиента — дескриптор сокета, к которому

применялся вызов `connect`. Для более гибкого управления обменом данными существуют также вызовы `recv` и `send`, отличающиеся от `read` и `write` только наличием дополнительного параметра `flags`. При желании читатель может ознакомиться с этими вызовами самостоятельно, воспользовавшись командой `man` или книгами [1] и [3].

Вызов `connect` можно применять не только к потоковым, но и к дейтаграммным сокетам. Настоящего соединения при этом не устанавливается, вызов влияет только на сам объект сокета в ядре; иначе говоря, в результате такого вызова `connect` никакие пакеты по сети не передаются и никакие удалённые машины никакой информации о возникшем «соединении» не получают. Зато поведение нашего собственного сокета меняется при этом весьма заметно. Сокет намертво «привязывается» к единственному (указанному в вызове `connect`) адресу партнёра по коммуникации. При чтении из сокета мы теперь получаем только дейтаграммы, обратным адресом в которых указан наш партнёр по «соединению»; все прочие полученные дейтаграммы ядро молча сбрасывает. Отправлять дейтаграммы мы теперь тоже можем только на один адрес. Обычно для приёма и передачи дейтаграмм через «соединённые» сокеты используются вызовы `read` и `write` или `send`/`recv`, а не `sendto`/`recvfrom`; их, впрочем, тоже можно использовать, но это довольно бессмысленно.

Завершить работу с дескриптором сокета можно, как уже говорилось, с помощью хорошо знакомого нам вызова `close`. Следует понимать, что в общем случае закрытие дескриптора не означает закрытия соединения и вообще прекращения использования сокета, ведь с одним и тем же сокетом (объектом ядра) может быть связано больше одного дескриптора, причём как в одном процессе, так и в разных. В разных процессах дескрипторы, связанные с одним и тем же потоком ввода/вывода (объектом ядра), появляются при создании нового процесса с помощью `fork`, а в одном процессе такие дескрипторы могут появиться в результате манипуляций с вызовами `dup` и `dup2` (см. §§5.4.3, 5.4.9). В обеих ситуациях в роли потоков ввода/вывода могут, естественно, оказаться сокеты; если с помощью `close` мы закрываем лишь один дескриптор из нескольких связанных с данным сокетом, то никакого влияния на сокет и его соединение наш вызов `close` не окажет, просто на него будет ссылаться на один дескриптор меньше.

Конечно, если мы закроем *все* дескрипторы, связанные с сокетом (или, что бывает гораздо чаще, закроем *единственный* такой дескриптор), системе всё-таки придётся закрыть соединение и уничтожить объект сокета. Точнее говоря, вызов `close` вернёт управление сразу же, но сокет может исчезнуть не сразу: система сначала передаст «на тот конец» соединения все данные, которые ещё не переданы, и лишь затем, отправив партнёру извещение о закрытии соединения, окончательно ликвидирует сокет как объект ядра. **Когда наш партнёр по соединению закрыл свой сокет, очередной вызов чтения (`read` или `recv`) вернёт нам значение 0, соответствующее ситуации «конец файла»**.

Несколько иной подход к завершению обмена информацией через установленное соединение реализует системный вызов `shutdown`:

```
int shutdown(int sd, int how);
```

Параметр `sd` задаёт дескриптор сокета, `how` — что́ именно следует прекратить. Для параметра `how` есть три возможных значения: `SHUT_RD` прекращает приём данных через сокет, `SHUT_WR` — передачу данных, `SHUT_RDWR` закрывает обмен данными в обоих направлениях. Полезно знать, что эти три константы были введены сравнительно недавно, а в более старых программах можно встретить обращения к `shutdown`, в которых параметр `how` задан явным образом в виде целого числа (соответственно `0`, `1` и `2`).

В отличие от `close` **вызов** `shutdown` **относится к сокету как объекту ядра, а не к отдельному дескриптору**. Это значит, что, если уж мы прекратили передачу информации через сокет в одном или в обоих направлениях, то передавать информацию в соответствующих направлениях не удастся ни через один из дескрипторов, связанных с данным сокетом, сколько бы их ни было. Есть и ещё один крайне важный момент: **вызов** `shutdown` **не уничтожает ни объект сокета, ни связанные с ним дескрипторы**, так что применение этого вызова не отменяет необходимости последующего закрытия сокета с помощью `close`. Помните, что количество одновременно открытых файловых дескрипторов в системе ограничено.

Когда с помощью `shutdown` прекращается *передача* данных через сокет, система отправляет «на тот конец» все данные, которые были ей переданы до вызова `shutdown`, после чего извещает партнёра о прекращении передачи данных. «На том конце» результатом такого прекращения становится ситуация «конец файла». При этом мы сохраняем возможность получать данные и, по идее, должны их получать до тех пор, пока «конец файла» не возникнет уже на нашем конце. Это позволяет реализовать типичный сценарий обмена информацией: после установления соединения клиент отправляет на сервер некий запрос, после чего делает `shutdown`; сервер получает «конец файла», из чего делает вывод, что запрос кончился, и только после этого анализирует полученный запрос и отвечает на него.

Прекращение *приёма* данных через сокет означает, что в дальнейшем получении данных мы не заинтересованы. Данные, полученные нашей системой «с того конца» до того, как мы обратились к `shutdown`, но которые мы ещё не прочитали, будут при этом сброшены; если система получит новые данные уже после запрета на их получение, то все такие данные будут «тихо проигнорированы».

Обычно при закрытии сокета вызовом `close` система возвращает управление немедленно; при этом, возможно, в буферной памяти ядра всё ещё находятся

данные для отправки. По умолчанию в этом случае система передаёт все оставшиеся данные и лишь затем закрывает соединение. Если в это время произойдёт ошибка, программа об этом уже не узнает.

Такое поведение вызова `close` в отношении потоковых сокетов можно изменить, модифицировав с помощью системного вызова `setsockopt` параметр `SO_LINGER`. Помимо варианта «по умолчанию» система предоставляет ещё две возможности: либо при вызове `close` немедленно сбросить все неотправленные данные и разорвать соединение, либо, напротив, заблокировать вызов `close` до тех пор, пока все данные, предназначенные к отправке, не окажутся отправлены и их доставка адресату не будет подтверждена. Для такого варианта система требует указания временно́го периода, в течение которого вызов `close` может оставаться заблокированным; если за это время система не успеет передать все данные и получить подтверждение об их доставке, соединение будет разорвано, а `close` вернёт ошибку.

Технические подробности работы с параметром `SO_LINGER` можно найти в книге [3] и в технической документации — в частности, дав команду «`man 7 socket`».

В статье [11] приведён придирчивый анализ текстов спецификации протокола TCP, на основании чего автор статьи делает вывод, что использования `SO_LINGER` недостаточно для достижения стопроцентной надёжности (либо все данные благополучно переданы и получены «на том конце», либо система сообщает об ошибке). Там же описаны некоторые трюки для повышения степени уверенности в корректной работе TCP. Текст статьи довольно сложен, но при должном приложении усилий понять его вполне возможно.

6.3.5. О «залипании» TCP-порта

Часто при работе с сервером можно заметить, что после завершения программы-сервера её некоторое время невозможно запустить с тем же значением номера порта. Это происходит обычно при некорректном завершении программы-сервера либо если программа-сервер завершается при активных клиентских соединениях. В этих случаях ядро операционной системы некоторое время продолжает считать адрес занятым (находящимся в так называемом состоянии `TIME_WAIT`). Делается это для того, чтобы все пакеты, которые (теоретически) могут быть в пути от вашего сокета к партнёру по соединению или обратно, достигли своей цели или исчезли, так и не дойдя по назначению; считается, что в противном случае такие пакеты, придя на наш компьютер в неподходящий момент, могут помешать работе соединений, которые установит новый процесс, использующий тот же самый адрес сокета.

На практике это выглядит довольно неприятно: в течение ощутимого времени после завершения старого процесса-сервера при попытке запустить его снова вызов `bind` выдаёт ошибку, не позволяя получить порт. Период `TIME_WAIT` может составлять от 30 до 120 секунд; если вы обнаружили ошибку в вашей серверной программе, то за это время вы можете успеть найти ошибочный фрагмент в исходном тексте,

исправить его, перекомпилировать программу и попытаться запустить исправленную версию — но не тут-то было; система всё ещё считает порт занятым. Кстати, если вы пренебрежёте проверкой значения, возвращаемого вызовом `bind`, то происходящее с вашей программой будет выглядеть как совершеннейшая мистика (хотя в действительности всё очень просто: начиная с `bind`, *все* вызовы работы с сокетами будут «ошибаться» и, как следствие, ничего не делать).

Эффект «залипшего» порта может превратить отладку серверных программ в натуральную пытку. К счастью, система сокетов позволяет изменить поведение ядра в отношении адресов, находящихся в состоянии `TIME_WAIT`. Для этого нужно перед вызовом `bind` выставить на будущем слушающем сокете опцию `SO_REUSEADDR`. Это делается с помощью системного вызова `setsockopt`:

```
int setsockopt(int sd, int level, int optname,
               const void *optval, int optlen);
```

Параметр `sd` задаёт дескриптор сокета, `level` обозначает уровень (слой) стека протоколов, к которому имеет отношение устанавливаемая опция (в данном случае это уровень сокетов, обозначаемый константой `SOL_SOCKET`). Параметр `optname` задаёт «имя» (на самом деле, конечно, это номер, или числовой идентификатор) устанавливаемой опции; в данном случае нам нужна опция `SO_REUSEADDR`. Поскольку информация, связанная с нужной опцией, может иметь произвольную сложность, вызов принимает нетипизированный указатель на значение опции и длину опции (параметры `optval` и `optlen` соответственно). Значением опции в данном случае будет целое число `1`, так что следует завести переменную типа int, присвоить ей значение `1` и передать в качестве `optval` адрес этой переменной, а в качестве `optlen` — выражение `sizeof(int)`. В итоге наш вызов будет выглядеть так:

```
int opt = 1;
setsockopt(ls, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

6.3.6. Сокеты для связи родственных процессов

В отличие от неименованных каналов, потоковые сокеты подразумевают двунаправленную связь, что делает их в некоторых случаях более удобным средством даже при взаимодействии родственных процессов, но процедура установления соединения, рассмотренная выше, представляется для такой ситуации слишком сложной. В связи с этим в ОС Unix предусмотрен вызов `socketpair`, создающий сразу два сокета, между которыми уже установлено соединение. Вот профиль этого вызова:

```
int socketpair(int af, int type, int protocol, int sv[2]);
```

Параметры `af`, `type` и `protocol` задают соответственно семейство адресации, тип и протокол для создаваемых сокетов. Следует отметить, что многие реализации допускают лишь одну комбинацию этих параметров: `AF_UNIX`, `SOCK_STREAM` и `0`. Параметр `sv` должен указывать на массив из двух элементов типа `int`, в которые вызов занесёт файловые дескрипторы двух созданных сокетов. Сокеты создаются уже связанными друг с другом, причём оба конца открыты как на чтение, так и на запись. Вызов возвращает `0` в случае успеха, `-1` в случае ошибки. От классического `pipe` результат работы `socketpair` отличается тем, что создаваемое соединение оказывается двухсторонним, в остальном принципы работы с ним совершенно такие же: предполагается, что после создания такой пары сокетов процесс породит одного или более потомков, а передача информации через созданное соединение будет производиться либо между потомком и предком, либо между потомками одного предка (процесса, обратившегося к `socketpair`).

6.4. Проблема очерёдности действий и её решения

6.4.1. Суть проблемы

Ранее мы упоминали, что при работе с сокетами потокового типа после принятия первого соединения в программе-сервере возникает проблема очередности действий. Поясним, в чём она заключается. После принятия первого же соединения у нашего сервера имеется два дескриптора, а после установления соединения с новыми клиентами их число возрастёт ещё больше. На один из дескрипторов может в любой момент прийти запрос на соединение от нового клиента, что потребует вызова `accept`. Но если мы сделаем `accept`, а никакого запроса на соединение не было, наша программа так и останется внутри вызова `accept`, причём, может быть, надолго: никто ведь не гарантирует, что кто-либо захочет установить с нами новое соединение. В это время на любой из клиентских сокетов, то есть сокетов, созданных предыдущими обращениями к `accept` и отвечающих за соединение с каждым из клиентов, могут прийти данные, требующие обработки; ясно, что пока мы висим внутри вызова `accept`, никакой обработки данных не произойдёт, т. к. мы их даже не прочитаем.

С другой стороны, если попытаться произвести чтение с того или иного клиентского сокета, есть риск, что клиент по каким-либо причинам долго не будет нам ничего присылать. Всё это время наша программа будет находиться внутри вызова `read` или `recv`, не выполняя никакой работы, в том числе не принимая новых соединений и не обрабатывая данные, поступившие от других (уже присоединённых) клиентов.

В ходе активной работы сервера к нежелательной блокировке может привести не только попытка приёма данных (или запросов), но и попытка отправки данных. В самом деле, пропускная способность сети ограничена; следовательно, заблокировать наш процесс могут не только вызовы `read` и `accept`, но и `write`. К примеру, если один из клиентов прислал нам запрос, в ответ на который мы должны передать ему файл размером в сотню мегабайт, у нас есть вполне реальная возможность растерять всех остальных клиентов за то время, пока этот файл будет передаваться, особенно если пытаться записывать данные в сокет большими порциями.

Проиллюстрировать проблему можно на примере из совсем некомпьютерной области. Представим себе ресторан, в котором каждый столик размещён в отдельной кабинке, так что в каком бы месте мы ни стояли, нельзя одновременно увидеть все столики и входные двери. Представим себе теперь, что на весь ресторан есть только один официант, исполняющий еще и обязанности метрдотеля. В нашей аналогии официант будет играть роль процесса. Сразу после открытия ресторана официант, естественно, подойдет к входным дверям и будет поджидать новых клиентов (вызов `accept`), чтобы встретить их и проводить к столику. Однако как только первые клиенты займут столик и примутся изучать меню, официант окажется перед проблемой: следует ли ему стоять около столика в ожидании, пока клиенты определятся с заказом, или же пойти к дверям проверить, не пришёл ли ещё кто-нибудь. Когда же клиентов за столиками станет много, официанту и вовсе придется тяжело. Ведь каждому клиенту за любым из столиков в любой момент может что-то понадобиться — он может решить заказать ещё что-нибудь, может уронить вилку и попросить другую, может, наконец, решить, что ему пора идти, и потребовать счёт. С другой стороны, в любой момент могут прийти и новые клиенты, и если их не встретить у дверей, они могут уйти, а ресторан недополучит денег. Напомним, что по условиям нашей аналогии в ресторане нет такого места, откуда было бы видно и столики, и входные двери; точнее говоря, из любого места видно либо один столик (но не больше), либо двери, либо вообще ни то, ни другое.

Самое простое решение, приходящее в голову — это бегать по всему ресторану, подбегая по очереди к каждому из столиков, а также и к дверям, узнавать, что внимание официанта там не требуется и идти, вернее, бежать на следующий круг. Аналогичное решение возможно и для нашего процесса. Можно с помощью вызова `fcntl` перевести все сокеты (и слушающий, и клиентские) в ***неблокирующий режим***; напомним, как это делается (здесь `sd` — дескриптор сокета; с переводом в неблокирующий режим мы уже встречались в §5.3.3, где делали это для обычных дескрипторов):

```
flags = fcntl(sd, F_GETFL);
fcntl(sd, F_SETFL, flags | O_NONBLOCK);
```

Для сокетов в неблокирующем режиме вызовы `read` и `accept` всегда возвращают управление немедленно, ничего не ожидая; если не было данных или входящего соединения, возвращается ошибка. Вызов `write` на неблокирующем сокете также возвращает управление немедленно; если ему не удалось записать ни одного байта, он завершается с ошибкой (т. е. возвращает `-1`), если же хотя бы один байт был записан, вызов считается прошедшим успешно; возвращает он при этом, как всегда, количество записанных байтов. Когда все сокеты настроены как неблокирующие, можно их опрашивать по очереди в бесконечном цикле. Это называется ***активным ожиданием***.

Следует отметить, что такой вариант считается совершенно неприемлемым в многозадачных системах. Официант из нашей аналогии будет понапрасну уставать, нарезая круги по ресторану (вполне возможно, что за несколько десятков таких кругов от него ничего никому так и не понадобится) и в итоге попросту упадёт без сил. Процесс, бесконечно опрашивающий набор сокетов с помощью системных вызовов, тоже будет вхолостую тратить процессорное время. Конечно, в отличие от официанта процесс не устанет, но ведь процессорное время, расходуемое без пользы, могло бы пригодиться другим задачам. Если в системе установлено ограничение на потребляемое процессом время, сервер может исчерпать свой лимит и будет уничтожен ядром системы.

Проблема действительно серьёзна. Существуют серверные программы, которые сутками, а иногда и месяцами находятся в бездействии, и было бы очень странно, если бы всё это время программа, которая совершенно ничего не делает, занимала бы процессор; но даже если сервер активно работает, это нельзя считать поводом для создания бесполезной нагрузки на центральный процессор.

Другое решение (для ресторана) состоит в том, чтобы нанять в ресторан адекватное количество персонала. Во-первых, разумеется, у дверей должен стоять метрдотель, встречая приходящих клиентов и проводя их к свободным столикам. Во-вторых, дорогой ресторан вполне может себе позволить прикрепить к каждому столику своего официанта. Аналогичным образом при написании серверной программы мы можем *создать отдельный процесс* для обслуживания каждого пришедшего клиента.

Если же этот вариант неприемлем — например, если бóльшую часть времени весь персонал все равно простаивает, — можно сделать и иначе. Официант может, нарезав несколько кругов, сообразить, что так дело не пойдёт и, например, подвесить колокольчик к входным дверям, а каждый столик снабдить кнопкой звонка. После этого можно будет спокойно усесться в укромный угол и спать или, например, разгадывать кроссворд, пока один из колокольчиков или звонков не зазвонит. Поскольку ресторан имеет свойство в некий момент закрываться, в дополнение к этим звонкам следует, видимо, завести ещё и будильник

на определённое время. Аналогичный вариант в программировании называется ***мультиплексированием ввода-вывода***.

Рассмотрим два последних варианта подробнее.

6.4.2. Решение на основе обслуживающих процессов

В этом варианте наш главный процесс выполняет обязанности метрдотеля, находясь бóльшую часть времени в вызове `accept`. Приняв очередное соединение, он порождает процесс для обслуживания этого соединения (аналог официанта, прикреплённого к столику). После этого родительский процесс закрывает сокет клиентского соединения, а порождённый процесс закрывает слушающий сокет. Все обязанности по обслуживанию пришедшего клиента возлагаются на порождённый процесс; после завершения сеанса связи с клиентом этот процесс завершается. Всё это время родительский процесс беспрепятственно продолжает исполнять обязанности «метрдотеля», вызывая `accept`. Соответствующий код будет выглядеть примерно так:

```
int ls;
struct sockaddr_in addr;
ls = socket(AF_INET, SOCK_STREAM, 0);
if(ls == -1) {
    /* ... ошибка ... */
}
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
if(-1 == bind(ls, &addr, sizeof(addr)) {
    /* ... ошибка ... */
}
for(;;) {
    socklen_t slen = sizeof(addr);
    int cls = accept(ls, &addr, &slen);
    if(fork() == 0) { /* обслуживающий процесс */
        close(ls);
        /* ...
           работаем с клиентом через сокет cls
           Клиент пришёл с адреса, хранящегося
           теперь в структуре addr
           ...
         */
        exit(0);
    }
    /* родительский процесс */
    close(cls);
    /* проверить, не завершились ли какие-либо
       процессы-потомки (убрать зомби) */
```

```
        while(wait4(-1, NULL, WNOHANG, NULL)>0)
            {}       /* тело цикла пустое */
    }
```

6.4.3. Событийно-управляемое программирование

Рассмотрим теперь случай, когда порождение отдельного процесса на каждое клиентское соединение неприемлемо. Так может получиться, если сервер достаточно серьёзно загружен: операция порождения процесса сравнительно дорога, так что при загрузках порядка тысяч соединений в секунду затраты на порождение процессов могут оказаться чрезмерными. Кроме того, между сеансами обслуживания разных клиентов может происходить активное взаимодействие. Например, сервер может поддерживать сетевую компьютерную игру, где действия каждого игрока влияют на ситуацию в одном игровом пространстве и сказываются на других игроках. В этом случае разделение серверной программы на отдельные процессы потребует высоких накладных расходов на взаимодействие между этими процессами; к тому же проблема очерёдности действий встанет снова, только уже в каждом из процессов: действительно, следует ли процессу в конкретный момент времени анализировать изменения в игре или же принимать данные с клиентского сокета?

В обоих случаях возникает необходимость обслуживания всех клиентов силами одного процесса, причём активное ожидание, естественно, приемлемым не считается. На проблему можно взглянуть шире. Есть некоторое количество типов ***событий***, каждый из которых требует своей обработки. Некоторые системные вызовы, предназначенные для обработки событий, являются *блокирующими*, то есть будучи вызваны раньше наступления события, они этого события ожидают, блокируя вызвавший процесс и делая невозможной обработку других событий. Может случиться и так, что ни одно из событий в течение долгого периода времени не произойдёт, и тогда нужно исключить холостой расход процессорного времени. Идеальной была бы ситуация, когда мы, каким-то образом узнав о наступлении события, реагируем на него соответствующим системным вызовом, точно зная, что он нас не заблокирует и при этом сделает что-то полезное (примет или передаст данные, примет запрос на соединение и т. п.). Этим полезным действием наша «идеальная» ситуация отличается от активного ожидания с циклическим опросом неблокирующих сокетов: не гарантировав никакого полезного действия, мы обрекаем большинство наших обращений к ядру на холостой расход процессорного времени.

Способ построения программ, при котором программа имеет главный цикл, одна итерация которого соответствует наступлению некоторого события из определённого множества,

а все действия программы построены как реакция на событие, называется *событийно-управляемым программированием* (англ. *event-driven programming*). Кроме серверных программ, рассчитанных на одновременное обслуживание неопределённого количества клиентов, существуют и другие области применения этого подхода. В частности, на принципе обработки событий обычно основываются программы, снабжённые графическим интерфейсом пользователя; в роли событий в них выступают действия пользователя — нажатия на кнопки и «горячие» клавиши, а также перемещения и «щелчки» мыши.

Итерация главного цикла в событийно-ориентированной программе делится на две фазы: *выборка события* и *обработка события*. Чаще всего непосредственная обработка событий выносится в отдельные функции, которые вызываются из главного цикла при наступлении соответствующего события. Эти функции так и называются *обработчиками событий*.

Одна из основных особенностей событийно-управляемого программирования состоит в том, что в обработчике события нельзя оставаться надолго. Что такое «надолго» — зависит от задачи; в программе с графическим интерфейсом «надолго» обычно означает «так, что пользователь заметит паузу», то есть десятую долю секунды в обработчике провисеть ещё можно, а вот полсекунды — уже нельзя, поскольку программа, то и дело «подвисающая» на полсекунды, среднестатистического пользователя может буквально взбесить. В случае TCP-сервера всё, как правило, не так критично, в крайнем случае даже пауза на две-три секунды может остаться незамеченной, нужно только не допускать, чтобы клиент «отвалился по тайм-ауту» (конкретная величина тайм-аута может оказаться разной, но довольно редко бывает меньше 20–30 секунд); впрочем, действовать по принципу «авось прокатит» — тоже идея не слишком удачная.

Так или иначе, можно определённо назвать одну вещь, которая в обработчиках событий заведомо недопустима: это обращения к блокирующим системным вызовам, когда имеется вероятность, что блокировка реально произойдёт. Длительность блокировки в большинстве случаев непредсказуема; допустив блокировку, мы уже не можем гарантировать своевременный возврат управления в главный цикл, то есть получается, что мы можем застрять в обработчике события на сколько угодно — в том числе «надолго».

Недопустимость блокировок в обработчиках сигналов в некоторых случаях вынуждает нас отказаться от очевидного и привычного в пользу таких методов программирования, которые, как это ни странно, некоторым программистам оказываются неподвластны. Когда логика задачи предполагает ту или иную *последовательность действий*, обычно мы без раздумий пишем в программе *последовательность операторов*, выполняющих эти действия. Но в событийно-ориентированной программе

такая последовательность операторов окажется недопустимой, если хотя бы один из них может заблокировать наш процесс. К этому вопросу мы вернёмся в §6.4.5.

6.4.4. Выборка событий в ОС Unix: вызов `select`

Для построения серверной программы в виде событийно-ориентированного приложения, очевидно, требуется поддержка со стороны операционной системы; в самом деле, нам нужно *дождаться* наступления события — иначе говоря, *заблокироваться* до момента, когда событие произойдёт, но ведь блокировать процессы умеет только операционная система.

В большинстве своём блокирующие системные вызовы (включая все, которые мы до сих пор обсуждали) ориентированы на ожидание одного конкретного события, но нам в данном случае нужна возможность отдать управление операционной системе, отказавшись от выполнения (т. е. от квантов времени) до тех пор, пока не произойдет одно из *многих* интересующих нас событий. Наступление события операционная система должна отследить сама и вернуть управление процессу, при этом, желательно, сообщив ему, какое именно событие наступило: пришли ли данные по одному из клиентских соединений, получен ли очередной запрос на создание нового соединения, освободилось ли место в буфере для передачи данных в сеть, прошло ли некое заданное время, пришёл ли сигнал. Получив информацию о событии, мы сможем среагировать на него соответствующим образом, то есть принять данные или запрос соединения, отправить очередную порцию данных и т. п., при этом, что важно, мы сможем избежать как холостого опроса источников событий, так и блокировок, мешающих обрабатывать новые события. Когда в роли событий рассматривается готовность нескольких потоков ввода-вывода к очередным операциям, а сами операции ввода-вывода становятся реакцией на эти события, говорят о ***мультиплексировании ввода-вывода***.

В системах семейства Unix выборку событий можно организовать с помощью системных вызовов `select` и `poll`, которые, вообще говоря, предназначены для одних и тех же действий; `select` несколько проще в работе, `poll` несколько более универсален. В некоторых системах ядро реализует только один вариант интерфейса, при этом второй эмулируется через него в виде библиотечной функции. Так, в системах линейки Solaris присутствовал системный вызов `poll`, а `select` был библиотечной функцией. Кроме того, в некоторых современных системах, в частности в FreeBSD, присутствует также вызов `kqueue`, реализующий альтернативный подход к выборке события. В Linux тоже имеется свой вариант событийного интерфейса — набор вызовов под общим названием `epoll`. Мы будем рассматривать вызов `select` как более простой;

изучить `poll`, а также `kqueue` и `epoll` читатель при желании может самостоятельно, прибегнув к технической документации.

Вызов `select` позволяет обрабатывать события трёх типов:

- изменение состояния файлового дескриптора (появление данных, доступных на чтение, или входящего запроса на соединение; освобождение места в буфере исходящей информации; исключительная ситуация);
- истечение заданного количества времени с момента входа в вызов;
- получение процессом неигнорируемого сигнала.

Профиль вызова выглядит так:

```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exfds,
           struct timeval *timeout);
```

Параметры `readfds`, `writefds` и `exfds` обозначают множества файловых дескрипторов, для которых нас интересует соответственно возможность немедленного чтения, возможность немедленной записи и наличие исключительной ситуации. Параметр `n` указывает количество значащих элементов в этих множествах. Этот параметр нужно установить равным `max_d+1`, где `max_d` — максимальный номер дескриптора среди подлежащих обработке. Наконец, параметр `timeout` задаёт промежуток времени, спустя который следует вернуть управление, даже если никаких событий, связанных с дескрипторами, не произошло.

Объект «множество дескрипторов» задаётся переменной типа `fd_set`. Внутренняя реализация переменных этого типа для различных систем может оказаться разной, но проще всего её представлять себе как битовую строку, где каждому дескриптору соответствует один бит. Для работы с переменными этого типа система предоставляет в наше распоряжение следующие макросы:

```
FD_ZERO(fd_set *set);                    /* очистить множество */
FD_CLR(int fd, fd_set *set);   /* убрать дескриптор из мн-ва */
FD_SET(int fd, fd_set *set);  /* добавить дескриптор к мн-ву */
FD_ISSET(int fd, fd_set *set); /* входит ли дескр-р в мн-во? */
```

Структура `timeval`, служащая для задания последнего параметра, имеет два поля типа long. Поле `tv_sec` задаёт количество секунд, поле `tv_usec` — количество микросекунд, т. е. миллионных долей секунды, и его значение не должно превышать `999999`. Например, задать тайм-аут в 5.3 секунды можно следующим образом:

```
struct timeval t;
t.tv_sec = 5;
t.tv_usec = 300000;
```

Отметим, что в разных системах вызов `select` по-разному ведёт себя по отношению к этому параметру. Одни реализации оставляют структуру

`*timeout` нетронутой, другие записывают в неё время, оставшееся до истечения заданного тайм-аута. Такая неопределённость приводит к тому, что мы вообще вынуждены не строить никаких предположений относительно содержимого этой структуры после выхода из `select`: использовать её для отслеживания остатков временны́х интервалов мы не можем, поскольку не все системы этот остаток туда записывают, но и предполагать её неизменность мы тоже не можем, ведь другие системы её изменяют. Приходится считать, что вызов `select` просто *портит* параметр `timeout`.

В качестве любого из параметров, кроме первого (количества дескрипторов), можно передать нулевой указатель, если задание этого параметра нам не требуется. Так, если нужно просто некоторое время подождать, можно указать `NULL` вместо всех трёх множеств дескрипторов; аналогичным образом можно обойтись и без последнего параметра, задающего тайм-аут. Можно заметить, что вызов `select(0,NULL,NULL,NULL,NULL)` эквивалентен вызову `pause()`; если же оставить нулевыми все параметры, кроме последнего, то можно сымитировать `sleep` и `usleep`.

Вызов `select` возвращает управление в следующих случаях:

- произошла ошибка (в частности, в одном из множеств дескрипторов оказалось число, не соответствующее ни одному из открытых дескрипторов); в этом случае вызов возвращает `-1`;
- наш процесс получил обрабатываемый сигнал; в этом случае также возвращается `-1`, а отличить эту ситуацию от ошибочной можно по значению глобальной переменной `errno`, которая будет равна константе `EINTR`;
- истёк тайм-аут, то есть с момента входа в вызов прошло больше времени, чем указано в параметре `timeout` (если, конечно, этот параметр не был нулевым указателем); в этом случае `select` возвращает `0`;
- на один из дескрипторов, входящих в множество `readfds`, пришли данные, которые можно прочитать, или запрос, который можно принять, либо возникла ситуация «конец файла»;
- один из дескрипторов, входящих в `writefds`, готов к немедленной записи, то есть если применить к нему вызов `write`, `send` или ещё какой-то подобный, то он сможет *немедленно* записать *хоть сколько-нибудь*[17] данных;
- с одним из потоков данных, чьи дескрипторы включены во множества `readfds` и `writefds`, случилась неприятность (произошла ошибка);

[17] В действительности это «хоть сколько-нибудь» имеет минимальное допустимое значение, которое вдобавок может быть изменено; по умолчанию это 2048 байт, то есть готовность на запись не наступит, пока ядро не будет готово принять прямо сейчас 2048 байт данных.

- на одном из дескрипторов, входящих во множество `exfds`, возникла исключительная ситуация.

В последних трёх случаях вызов `select` возвращает количество дескрипторов, на которых произошли события, причём если дескриптор входит в разные множества и на нём одновременно наступили разные события, он может быть посчитан больше одного раза. Все множества дескрипторов, переданные вызову `select`, при этом перезаписываются: в них остаются только те дескрипторы, на которых что-то произошло (готовность, ошибка и т. д.). Проверив с помощью макроса `FD_ISSET` интересующие нас дескрипторы, мы можем узнать, на каком из них требуется выполнить операцию чтения, принятия соединения, записи и т. п.

Сразу же отметим, что исключительные ситуации бывают только на сетевых сокетах и только при использовании механизма *внеполосных данных* (англ. *out-of-band*, OOB), а он используется сравнительно редко, так что и сам параметр `exfds` используется редко; в большинстве случаев в качестве четвёртого параметра `select` указывается `NULL`. Рассматривать внеполосные данные мы в нашей книге не будем.

Готовность на чтение для обычных потоков, т. е. файловых дескрипторов, допускающих чтение вызовом `read`, означает, что во входном буфере есть хотя бы один байт данных, так что вызов `read` (**один**) не заблокируется. Следует обратить внимание, что **ситуация «конец файла» также истолковывается как готовность сокета на чтение**, поскольку в этой ситуации вызов `read` не блокируется — он немедленно возвращает `0`.

Во множество `readfds` можно включать не только дескрипторы, доступные на чтение `read`'ом; так, ни к слушающим сокетам, ни к дейтаграммным сокетам, работающим без соединения, очевидно, `read` применять нельзя, но в `readfds` их можно включить с совершенно аналогичной целью: чтобы узнать, когда с ними пора работать. **Для слушающего сокета в роли «пришедших данных» выступают запросы на соединение**, то есть если `select` заявил о готовности слушающего сокета на «чтение», это гарантирует, что вызов `accept` не заблокируется. В случае сокетов без соединения гарантируется, что не будет заблокирован соответствующий вызов `recvfrom` (пришла дейтаграмма), и т. п.

С готовностью к записи дела обстоят несколько сложнее. При обсуждении системного вызова `write` в §5.3.3 мы отметили, что в ситуации, когда все данные не могут быть записаны немедленно, реализация `write` в большинстве систем предпочитает заблокироваться до тех пор, пока все данные не будут переданы, и лишь после этого вернуть управление. Готовность к записи, о которой нам сообщает `select`, означает, что *сколько-то* данных в сокет (или другой поток) можно записать немедленно, но само по себе это никоим образом не гарантирует, что `write`

пройдёт без блокировки: мы ведь не знаем, *сколько* места свободно в буфере ядра, который связан с нашим потоком, и можем потребовать от `write` записи бóльшего количества данных. Вызов примется передавать всё, что мы ему дали, и не вернёт управление нашему процессу, пока не справится с поставленной задачей полностью. К сожалению, это может затянуться, то есть произойдёт та самая блокировка, которая нам совершенно не годится, ведь у нас могут быть другие клиенты со своими запросами.

Часто в силу специфики задачи эту проблему попросту игнорируют. В самом деле, если порции данных, которые вы отправляете своим клиентам, не превышают одного-двух килобайт, вы можете считать, что блокировки на `write` у вас никогда не случится. Даже если она произойдёт, с такими объёмами данных её длительность будет пренебрежимо мала и никак не повлияет на всю вашу программу. В этой ситуации можно вообще не задействовать в вызове `select` параметр `writefds`.

Если же объёмы данных, отправляемых клиентам, могут быть существенными, то игнорировать проблему блокировки при записи уже не получится. Единственный *правильный* выход здесь — перевести ваши сокеты в неблокирующий режим (см. стр. 217), в противном случае никто не может гарантировать вам, что `write` не заблокируется. Значение, возвращаемое вызовом `write`, нужно будет анализировать всегда, предполагая, что оно может оказаться меньше, чем вы ожидали. Данные, подлежащие отправке, вам придётся хранить в своих собственных буферах до тех пор, когда сначала `select` сообщит вам, что можно (осмысленно) попытаться записать данные в сокет, а затем уже `write` подтвердит, что данные отправлены. Когда `write` отправляет только часть данных, эту часть следует изъять из своего «исходящего» буфера, а остальное оставить в буфере до следующего раза, когда `select` снова покажет готовность сокета к записи.

Отметим, что **большинство дескрипторов, открытых на запись, к записи готовы в любой момент**; дескриптор потока вывода может быть не готов к записи только если буфер исходящих данных, который ядро поддерживает для данного потока, заполнен до отказа. Если случайно внести какой-то из дескрипторов, исходящий буфер которых не переполнен, в множество `writefds`, вызов `select` вернёт управление немедленно; если при этом у вас не найдётся данных, которые в этот дескриптор можно записать, фактически ваша программа, несмотря на использование `select`, будет работать в режиме активного ожидания. Поэтому в множество `writefds` следует вносить только те сокеты, для которых прямо сейчас есть данные, требующие отправки.

Работу с вызовом `select` можно построить по нижеприведённой схеме. Предполагается, что номер слушающего сокета хранится в переменной `ls`; организовать хранение дескрипторов клиентских сокетов можно самыми разными способами, в зависимости от задачи: в списке,

в массиве, в дереве и т. п. Кроме того, предполагается, что out-of-band data не используется, так что параметр `exfds` оставлен нулевым, как и параметр `timeout`.

```
for(;;) { /* главный цикл */
    int fd;
    fd_set readfds, writefds;
    int max_d = ls;
     /* изначально полагаем, что максимальным является
        номер слушающего сокета */
    FD_ZERO(&readfds); /* очищаем множество */
    FD_ZERO(&writefds);
    FD_SET(ls, &readfds); /* вводим в множество
                дескриптор слушающего сокета */
         /* организуем цикл по сокетам клиентов */
    for(fd=/*дескриптор первого клиента*/ ;
        /*клиенты ещё не исчерпаны?*/     ;
        fd=/*дескриптор следующего клиента*/)
    {
         /* здесь fd - очередной клиентский дескриптор */
         /* вносим его в множества */
        FD_SET(fd, &readfds);
        if(для клиента есть данные к отправке?)
            FD_SET(fd, &writefds);
         /* проверяем, не больше ли дескриптор,
            нежели текущий максимум */
        if(fd > max_d)
            max_d = fd;
    }
    timeout.tv_sec = /*    заполняем    */;
    timeout.tv_usec = /*    тайм-аут     */;

    int res = select(max_d+1, &readfds, NULL, NULL, NULL);
    if(res < 1) {
        if(errno != EINTR) {
            /* обработка ошибки, происшедшей в select'е */
        } else {
            /* обработка события "пришедший сигнал" */
        }
        continue; /* дескрипторы проверять бесполезно */
    }
    if(res == 0) {
        /* обработка события "тайм-аут" */
        continue; /* дескрипторы проверять бесполезно */
    }
    if(FD_ISSET(ls, &readfds)) {
        /* пришёл новый запрос на соединение */
        /* здесь его надо принять вызовом accept
```

```
                и запомнить дескриптор нового клиента;
                кроме того, не забудьте про перевод сокета
                в неблокирующий режим с помощью fcntl,
                если используете writefds */
        }
          /* теперь перебираем все клиентские дескрипторы */
        for(fd=/*дескриптор первого клиента*/ ;
            /*клиенты ещё не исчерпаны?*/     ;
            fd=/*дескриптор следующего клиента*/)
        {
            if(FD_ISSET(fd, &readfds)) {
                /* пришли данные от клиента с сокетом fd */
                /* читаем их вызовом read или
                   recv и обрабатываем */
                /* не забываем отдельно обработать
                   ситуацию "конец файла"! */
            }
            if(FD_ISSET(fd, &writefds)) {
                /* готовность на запись: пробуем отправить
                   очередную порцию данных */
            }
        }
        /* конец главного цикла */
    }
```

Как отмечалось в предыдущем параграфе, в событийно-ориентированных программах тело главного цикла делится на две фазы — *выборку события* и *обработку события*. В нашей схематической программе с фаза выборки заканчивается вызовом `select`, а всё остальное до конца цикла — это обработка полученного события.

6.4.5. Сеанс работы как конечный автомат

Как уже отмечалось в §6.4.3, недопустимость действий, способных привести к блокировке процесса, вынуждает программистов отказываться от привычных методов работы, заменяя их неочевидными конструкциями: так, реализация *простой последовательности действий*, если хотя бы одно из этих действий может привести к блокировке, в событийно-ориентированной программе не будет иметь ничего общего с привычной *последовательностью операторов*.

В частности, системные вызовы `read` или `recv` могут отдать вам лишь часть запроса, отправленного клиентом; в иной ситуации вы могли бы просто «дочитать» остаток запроса, повторно вызвав `read`/`recv`, но в событийно-ориентированной программе так делать нельзя, ведь гарантия отсутствия блокировки, которую вам дал вызов `select`, распространяется только на *один* вызов чтения. Поэтому, коль скоро запрос

прочитан не полностью, приходится запомнить ту его часть, которая была считана, после чего вернуть управление в главный цикл. Вполне возможно, что ваша программа успеет пообщаться с другими клиентами, принять новые соединения и т. п., прежде чем вы наконец получите остаток недочитанного запроса; возможно, для этого потребуется больше одного чтения, и каждый раз придётся возвращать управление в главный цикл, а после каждого чтения «склеивать» новую порцию данных с принятыми от того же клиента ранее и проверять, получен ли, наконец, весь запрос или придётся подождать следующей порции.

Из этого следует, что **при событийно-ориентированном построении TCP-сервера совершенно необходимо предусмотреть для каждого из клиентов отдельный накопительный буфер**, в котором прочитанная часть запроса будет храниться до тех пор, пока запрос не окажется принят целиком. Следует также учитывать, что если ваш протокол общения с клиентом позволяет клиенту направить вам (серверу) следующий запрос, не дожидаясь ответа на предыдущий, то в вашем буфере в какой-то момент может оказаться запрос целиком, а следом за ним — начало следующего запроса. Если запросы не слишком велики, то вы вполне можете обнаружить в буфере несколько запросов сразу. Всё это приходится учитывать при анализе содержимого вашего накопительного буфера.

Склеиванием запроса из отдельных кусочков, прочитанных в разных вызовах обработчика события, проблемы не ограничиваются. Логика взаимодействия сервера с клиентом в большинстве случаев предполагает некие цепочки, составленные из отдельных обменов данными; такие цепочки как раз и образуют протоколы прикладного уровня. Если бы клиент был один, всю логику прикладного протокола можно было бы напрямую, практически один в один, перевести в операторы языка программирования, но в событийно-ориентированных программах такой подход не годится.

Для примера вспомним школьную задачу по информатике — программу, которая «знакомится» с пользователем, задавая ему один за другим несколько вопросов, например, «как вас зовут», «сколько вам лет», «из какого вы города» и «назовите вашую любимую музыкальную группу». Школьник, изучающий Паскаль, ни на секунду не задумываясь, напишет для этого примерно такой фрагмент:

```
write('What is your name? ');
readln(name);
write('How old are you? ');
readln(age);
write('What city/town are you from? ');
readln(town);
write('What is your favorite band? ');
readln(band);
```

Если подобный диалог потребуется организовать с пользователем, находящимся «по ту сторону» сетевого соединения, мы могли бы написать аналогичную последовательность действий на Си, заменив паскалевский оператор печати на системный вызов `write`, отправляющий вопросы в сетевой сокет, а оператор `readln` — на цикл обращений к вызову `read`, работающий, пока не будет прочитана строка целиком. Но поступить так мы можем лишь когда клиент у нас один и нам не нужно отвлекаться на обслуживание других сеансов связи — например, если мы применяем подход с обслуживающими процессами.

В событийно-ориентированной программе ни о чём подобном не может идти речи, ведь каждый `read` — это источник блокировки; стоит только пользователю отвлечься от общения с нашим сервером, оставив какой-то из вопросов без немедленного ответа, и все остальные пользователи, подключившиеся к серверу, обнаружат, что сервер перестал реагировать на передаваемую ими информацию, а новые клиенты и вовсе не смогут подключиться; все будут вынуждены ждать, когда же единственный «рассеянный» пользователь вспомнит о своём диалоге с сервером. Сервер, построенный подобным образом, ни на что не годится. В конце концов, кто-нибудь может *намеренно* подключиться, начать диалог и не завершить его, оставив соединение открытым, чтобы нарушить работу нашего сервера — из вредности или же, возможно, с целью устранения конкурента в нашем лице.

Для построения последовательного диалога с пользователем в событийно-ориентированной программе приходится *для каждого подключённого клиента* помнить, на какой стадии диалога мы с ним находимся (сакраментальное «на чём мы остановились»). После каждого выполненного чтения из сокета мы вне всякой зависимости от достигнутых результатов обязаны немедленно вернуть управление в главный цикл. Если очередная строка, отправленная пользователем, пришла полностью, то есть в накопительном буфере, связанном с данным клиентом, нашёлся символ перевода строки, мы изымаем строку из буфера, обрабатываем полученный ответ (например, сохраняем информацию в переменных, тоже связанных с конкретным клиентом, но это уже зависит от задачи), после чего запоминаем, что эту стадию диалога с данным клиентом мы уже прошли, отправляем ему очередной вопрос и после этого обработку события заканчиваем, то есть возвращаемся в главный цикл. Если строка после очередного обращения к `read` пока ещё прочитана не полностью — вообще не меняем ничего, кроме накопительного буфера, и немедленно выходим из обработчика события. Продолжить диалог с этим клиентом мы сможем, когда `select` снова сообщит нам о поступлении данных от него — и не раньше.

Итак, в событийно-ориентированной программе нам нужно помнить, на какой стадии коммуникации мы находимся с каждым из клиентов, или, иначе говоря, *помнить* ***состояние*** *сеанса взаимодействия* для

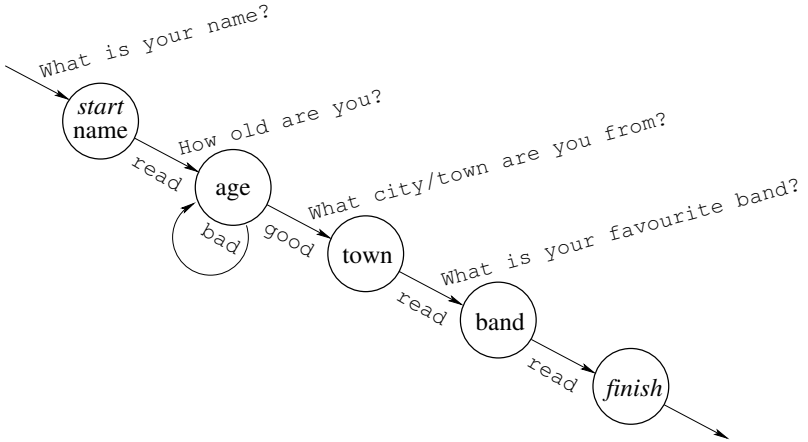

Рис. 6.9. Диаграмма состояний сеанса «знакомства» с пользователем

каждого подключённого клиента. Это состояние меняется при получении от клиента очередного запроса, ответа на наш запрос или другой информации.

Из курса дискретной математики читателю может быть известно понятие ***конечного автомата***[18] — абстрактного устройства, которое может находиться в одном из нескольких предопределённых состояний и переходить из одного состояния в другое при получении символов из входного потока (или, в более общем случае, при наступлении тех или иных событий). То, как приходится описывать сеанс работы с клиентом в событийно-ориентированных программах, весьма напоминает конечные автоматы.

Диаграмма состояний для диалога с пользователем из нашего примера приведена на рис. 6.9; отметим, что на подобных диаграммах, обычно называемых ***диаграммами Мура***, дуги (стрелки), соединяющие разные состояния, подписываются, во-первых, условием, при котором происходит переход из одного состояния в другое, и, во-вторых, (хотя и не всегда) — *действием*, которое автомат должен выполнить в качестве своего рода побочного эффекта. Наш автомат при создании сеанса работы с клиентом (например, сразу после выполнения вызова `accept`) отправляет в сеть свой первый вопрос (`What is your name?`) и принимает начальное состояние; переход в новое состояние происходит, когда получен ответ, под которым мы подразумеваем строку, заканчивающуюся символом перевода строки. При переходе в новое состояние мы отправляем пользователю текст очередного вопроса. Состояние «start»,

[18] Соответствующий английский термин — *finite state machine* (FSM); в англоязычных текстах, посвящённых программированию, эта аббревиатура встречается довольно часто, так что полезно знать, что это такое.

оно же «name», означает, что мы ещё ни одного ответа не получили, в состояние «age» мы переходим, когда пользователь ответит нам на вопрос о его имени, а мы спросим его о возрасте, в состояние «town» — после валидного ответа на вопрос о возрасте и т. д. Состояние «finish» означает, что мы успешно завершили диалог с пользователем и больше никаких данных «с той стороны» не ожидаем.

Обычно вводят ещё ошибочное состояние (например, «error»), в которое автомат переходит, если произошла ошибка, не позволяющая продолжить работу с данным клиентом. К примеру, в такое состояние мы могли бы перейти, если при чтении у нас кончился буфер, отведённый под накопление входящих данных, а строка полностью так и не пришла: корректное восстановление после такой ошибки в принципе возможно, но довольно проблематично, а буфер можно сделать такого размера, чтобы по смыслу нашей задачи он заведомо (и многократно) превосходил размер любой осмысленной строки, какую может прислать клиент. Например, буфер в 2048 байт будет точно больше, чем привычные нам человеческие имена, названия городов, музыкальных групп и тем более — чем десятичная запись чисел, которые могли бы означать возраст. Если принимаемая строка всё-таки не поместится в такой буфер, то в отказе от дальнейшей работы ничего криминального не будет. Но такую ситуацию нужно уметь отличить от ситуации успешного завершения диалога, и в этом нам как раз поможет введение специального состояния для обозначения ошибки.

Наш пример с диалогом — очень простой, в нём из каждого состояния воображаемый конечный автомат может перейти только в одно (следующее) состояние, причём такой переход во всех случаях, кроме одного, происходит без проверки дополнительных условий — достаточно того, что пришла *какая-то* строка. В более сложных случаях возможны различные варианты переходов из одного и того же состояния в зависимости от наступления тех или иных событий; мы могли бы спросить пользователя о каком-нибудь аспекте его взглядов на жизнь и в зависимости от этого скорректировать дальнейшие вопросы (например, принципиальным пешеходам бессмысленно задавать вопрос о любимой марке автомобиля, а вегетарианцам — о предпочтениях в области охотничьего оружия); можно было бы варьировать происходящее также в зависимости от указанного пользователем возраста и т. п. Диаграммы состояний часто оказываются достаточно сложными и перестают помещаться на бумаге и экране; в таких ситуациях приходится применять хорошо знакомые нам методы борьбы со сложностью — объединять состояния в группы и составлять для каждой группы отдельную диаграмму, а ещё одной диаграммой описывать переходы между группами.

Относительно конечных автоматов следует уяснить два момента. Во-первых, **при реализации конечных автоматов в программах состояниям соответствуют значения переменных**. В простейшем

случае можно завести специальный перечислимый тип, имеющий столько же возможных значений, сколько состояний предполагается реализовать в автомате; текущее состояние автомата будет в этом случае храниться в переменной этого типа. Для диалога из нашего примера можно было бы применить следующий перечислимый тип:

```
enum fsm_states {
    fsm_start,
    fsm_name = fsm_start,
    fsm_age,
    fsm_town,
    fsm_band,
    fsm_finish,
    fsm_error
};
```

В более сложных случаях дело может не ограничиться одной переменной. Никто не запрещает делать состояние сколь угодно сложным, включать в него всевозможные счётчики и другие параметры; конечный автомат от этого не перестанет быть конечным автоматом.

Второй момент несколько более сложен для понимания. **Конечный автомат не может управлять поступлением событий**, он над ними не властен. При переходе из одного состояния в другое автомат может выполнять какие-то дополнительные действия (так, в нашем примере он отправлял пользователю текст очередного вопроса), но вот *выборка события* должна оставаться для автомата чем-то внешним и неподконтрольным. Автомат, в частности, не может решать, читать ему очередную строку или не читать: его просто *извещают* о поступлении очередной строки (в общем случае — о наступлении события), ставят перед фактом, а его дело — среагировать, изменив соответствующим образом своё состояние.

Продолжим обсуждение программы, ведущей диалог с пользователем через TCP-соединение. Сеанс работы с клиентом можно описать структурой данных, содержащей дескриптор сокета, состояние автомата, накопительный буфер и поля для размещения сведений, полученных в ходе опроса:

```
#define INBUFSIZE 1024

/* ... */

struct session {
    int fd;                     /* дескриптор сокета */
    char buf[INBUFSIZE];      /* накопительный буфер */
    int buf_used;          /* кол-во данных в буфере */
    enum fsm_states state;       /* состояние сеанса */
```

```
    char *name, *town, *band; /* полученные сведения */
    int age;
};
```

Инициализацию сеанса, то есть действия, выполняемые сразу после вызова `accept`, можно совместить с созданием в динамической памяти экземпляра такой структуры. Кроме выделения памяти под структуру и заполнения её полей начальными значениями, в инициализацию сеанса придётся включить отправку клиенту первого вопроса, поскольку согласно нашему простенькому «протоколу» начать диалог (и в дальнейшем вести его) должен именно сервер. У нас получится что-то вроде следующего:

```
struct session *make_new_session(int fd)
{
    struct session *sess = malloc(sizeof(*sess));
    sess->fd = fd;
    sess->buf_used = 0;
    sess->state = fsm_start;
    sess->name = NULL;
    sess->town = NULL;
    sess->band = NULL;
    sess->age = -1;
    session_send_string(sess, "What is your name?\n");
    return sess;
}
```

Для отправки строки мы здесь воспользовались отдельной функцией `session_send_string`, поскольку это действие нам ещё потребуется, и не раз. Текст этой функции совсем простой:

```
void session_send_string(struct session *sess, const char *str)
{
    write(sess->fd, str, strlen(str));
}
```

Он мог бы быть гораздо сложнее, если бы мы использовали, наряду с буфером для входящих данных, также и буфер для отправки в сочетании с неблокирующим режимом сокета и отслеживанием его готовности к записи; но в задаче, которую мы сейчас решаем, общий объём данных, передаваемых в ходе отдельно взятого сеанса работы, запросто поместится в один пакет, так что возможностью блокировки вызова `write` мы можем пренебречь.

Шаг автомата будет реализован в другой функции, которую вызовут, когда в накопительном буфере окажется полная строка. Вызывающий создаст для нас копию этой строки, а из буфера её уберёт, сдвинув в нём данные в начало и уменьшив значение поля `buf_used`; функция, таким

образом, становится *владельцем* копии строки с момента её вызова. Это удобно в большинстве состояний, поскольку полученный от клиента ответ сохраняется в структуре сеанса, и если бы переданная нам строка не отдавалась нам в единоличное пользование, нам пришлось бы её скопировать ещё раз; получив её «в собственность», мы, в свою очередь, делаем её «владельцем» структуру сеанса работы. Единственное исключение — состояние `fsm_age`, в котором строка переводится в число, а сама она после этого не нужна и её приходится удалять. Отметим, что обработка состояния `fsm_age` несколько сложнее, чем для остальных состояний, так что мы вынесли эту обработку во вспомогательную функцию, чтобы не загромождать код второстепенными деталями. В целом шаг автомата получается таким:

```
void session_fsm_step(struct session *sess, char *line)
{
    switch(sess->state) {
    case fsm_name:
        sess->name = line;
        session_send_string(sess, "How old are you?\n");
        sess->state = fsm_age;
        break;
    case fsm_age:
        session_handle_age(sess, line);
        free(line); /* because we still own it... */
        break;
    case fsm_town:
        sess->town = line;
        session_send_string(sess,
            "What is your favorite band?\n");
        sess->state = fsm_band;
        break;
    case fsm_band:
        sess->band = line;
        session_send_string(sess, "Thank you, bye!\n");
        sess->state = fsm_finish;
        break;
    case fsm_finish:
    case fsm_error:
        free(line);   /* this should never happen */
    }
}
```

Оператор выбора (`switch`), выполняющий разные фрагменты кода в зависимости от текущего состояния — это классическая идиома при реализации конечных автоматов в программах. В вышеприведённом фрагменте наглядно видно, как меняется состояние и какая из веток будет выполнена при следующем вызове функции, то есть на следую-

щем шаге автомата. Вспомогательная функция для обработки ответа о возрасте выглядит так:

```
void session_handle_age(struct session *sess, const char *line)
{
    char *err;
    int age;
    age = strtol(line, &err, 10);
    if(!*line || *err || age < 0 || age > 150) {
        session_send_string(sess, "Please try again!\n");
        session_send_string(sess, "How old are you?\n");
    } else {
        sess->age = age;
        session_send_string(sess,
                            "What city/town are you from?\n");
        sess->state = fsm_town;
    }
}
```

Конечно, все эти функции не предназначены для вызова напрямую из главного цикла. Главный цикл, вообще говоря, не должен ничего знать о том, как устроен сеанс работы; здесь мы продолжаем соблюдать принцип борьбы со сложностью через разделение зон ответственности. Так или иначе, получив от вызова `select` информацию о готовности сокета к чтению, главный цикл должен вызвать (напрямую или как-либо косвенно) функцию, входящую в реализацию сеанса, которая выполнит чтение из сокета в накопительный буфер. Эту функцию мы назовём `session_do_read`. Она будет возвращать значение, которое укажет вызывающему, следует ли продолжать работу с данным сеансом или можно его закрыть: 1 будет означать. что сеанс ещё не завершён и должен продолжаться, 0 — что продолжение не требуется. В свою очередь функция `session_do_read`, выполнив чтение, дальнейшие действия — проверку, пришла ли строка, и вызов шага автомата — возложит на следующую функцию, которая будет называться `session_check_lf` (`lf` образовано от *line feed* — одно из названий символа перевода строки). Заодно (уже после возможного изъятия из буфера очередной строки) функция проверит, не переполнился ли накопительный буфер, а затем — не перешёл ли автомат в заключительное состояние. Код функции получается таким:

```
int session_do_read(struct session *sess)
{
    int rc, bufp = sess->buf_used;
    rc = read(sess->fd, sess->buf + bufp, INBUFSIZE - bufp);
    if(rc <= 0) {
        sess->state = fsm_error;
        return 0;
    }
    sess->buf_used += rc;
    session_check_lf(sess);
    if(sess->buf_used >= INBUFSIZE) {
```

```
            /* we can't read further,
               no room in the buffer, no whole line yet */
            session_send_string(sess, "Line too long! Bye...\n");
            sess->state = fsm_error;
            return 0;
        }
        if(sess->state == fsm_finish)
            return 0;
        return 1;
    }
```

Наконец, функция `session_check_lf` должна связать между собой «системную» обвеску и «прикладной» автомат. Для этого она попытается найти в «занятой» части накопительного буфера символ перевода строки; если его там нет, то ей ничего не останется, кроме как вернуть управление в надежде, что в следующий раз всё будет по-другому. Если же перевод строки в буфере найдётся, функция создаст копию данных из буфера (от начала до перевода строки) в виде отдельной строки в памяти, причём эта строка не будет включать символ перевода строки. Данные в буфере будут сдвинуты к началу, чтобы затереть изъятую строку, а саму эту строку функция передаст уже знакомой нам функции `session_fsm_step`, реализующей шаг автомата. В качестве последнего штриха наша функция проверит, не оканчивается ли полученная строка символом возврата каретки (`'\r'`), и если да — уберёт его из строки, заменив нулевым байтом. Код функции будет выглядеть так:

```
    void session_check_lf(struct session *sess)
    {
        int pos = -1;
        int i;
        char *line;
        for(i = 0; i < sess->buf_used; i++) {
            if(sess->buf[i] == '\n') {
                pos = i;
                break;
            }
        }
        if(pos == -1)
            return;
        line = malloc(pos+1);
        memcpy(line, sess->buf, pos);
        line[pos] = 0;
        memmove(sess->buf, sess->buf+pos, pos+1);
        sess->buf_used -= (pos+1);
        if(line[pos-1] == '\r')
            line[pos-1] = 0;
        session_fsm_step(sess, line); /* we transfer ownership! */
}
```

Как можно заметить, мы привели здесь лишь часть реализации сервера, проводящего опрос пользователей — ту, которая относится к сеансу работы, построенному в виде конечного автомата. Полностью текст программы, организующей с пользователями вышеописанный диалог через TCP-соединения и сохраняющей результаты в текстовый файл, читатель найдёт в архиве примеров к книге; соответствующий файл называется `tcp_questn.c`.

Любопытно заметить, что при переходе от программы, построенной как последовательность действий, к программе, основанной на состояниях, в действительности не возникает ничего нового. Если вернуться к фрагменту на Паскале, приведённому на стр. 6.4.5, то *текущее состояние* диалога с пользователем в этой программе тоже присутствует, просто выражено оно не значением переменной, а *текущей позицией исполнения* самой программы. Иногда говорят, что состояние в обычных императивных (то есть построенных на приказаниях) программах присутствует *неявно* — в отличие от случая, когда состояние *явным образом* определяется значениями переменных. Сам этот стиль написания программ называют ***программированием в терминах явных состояний***.

6.4.6. Неблокирующее установление соединения

Достаточно часто одна и та же программа выступает как в роли сервера, так и в роли клиента. Например, в серверной программе может потребоваться запросить ту или иную информацию у другого сервера, для чего придётся инициировать создание соединения, то есть стать клиентом другого сервера.

Если серверная программа написана в виде событийно-ориентированного приложения, то, как мы неоднократно подчёркивали, в ней нельзя допускать блокировку ни на каких системных вызовах; естественно, к вызову `connect` это тоже относится. Между тем обычно вызов `connect` блокирует вызвавший процесс до тех пор, пока попытка установления соединения не завершится успехом или неудачей. Время такой блокировки может составить (в наихудших случаях) несколько десятков секунд, что для событийно-ориентированных программ совершенно неприемлемо.

Техника установления соединения с помощью вызова `connect`, гарантирующая отсутствие блокировки, довольно нетривиальна, но освоить её вполне возможно. Итак, первое, что нужно сделать — это перевести сокет в неблокирующий режим (см. стр. 217). После этого можно сделать вызов `connect`, как это обсуждалось в §6.3.4; поскольку сокет находится в неблокирующем режиме, вызов вернёт управление немедленно, и здесь возможны три варианта:

- если вызов `connect` вернул `0` — значит, соединение уже благополучно установлено; так бывает сравнительно редко и обычно

только при соединении с сервером, работающим на одной с нами машине, но игнорировать эту возможность ни в коем случае не следует;

- если вызов вернул `-1`, но при этом в переменной `errno` содержится значение `EINPROGRESS` — значит, запрос на соединение отправлен, но результатов пока нет; это самый частый и самый интересный случай;
- если вызов вернул `-1` и переменная `errno` содержит какое-то другое значение, отличное от `EINPROGRESS` — значит, произошла ошибка, то есть соединение заведомо не будет установлено; ждать в этом случае нечего.

С первым и третьим случаями всё понятно; интереснее всего, когда `connect` завершился с «ошибкой» `EINPROGRESS`, которая на самом деле вовсе не ошибка — это значение в `errno` указывает, что установление соединения началось, но пока не завершилось. В этой ситуации следует внести дескриптор сокета в множество дескрипторов, для которых нас интересует *готовность к записи* (в терминах предыдущего параграфа — множество `writefds`); именно готовность к записи (не к чтению!) покажет, что попытка установления соединения так или иначе закончилась — либо успехом, либо неудачей.

Остаётся узнать, удалось установить соединение или нет, и здесь нам придётся упомянуть ещё один системный вызов — `getsockopt`:

```
int getsockopt(int sockfd, int level, int optname,
               void *optval, socklen_t *optlen);
```

Никакой логики в том, что придётся использовать именно этот вызов, вообще-то нет: `getsockopt` изначально предназначен, чтобы узнавать текущее значение параметров сокета, устанавливаемых с помощью уже знакомого нам `setsockopt`; очевидно, что статус «ошибки» к настраиваемым параметрам сокета («опциям») никакого отношения не имеет. Так или иначе, логично это или нет, но узнать, чем кончилась попытка установления соединения, мы можем только так: в качестве «уровня» указываем уже знакомое нам `SOL_SOCKET`, в качестве «имени опции» — `SO_ERROR`; параметр `optval` должен задавать адрес области памяти, куда следует записать значение «опции» — в данном случае эта область памяти представляет собой обычную переменную типа `int`; наконец, параметр `optlen` работает (в общем случае) как на вход, так и на выход — на входе через него передаётся размер области памяти, адрес которой передан через `optval`, а на выходе эта переменная будет содержать количество байтов, которое вызов реально использовал для записи значения «опции». В нашем случае мы точно знаем, что это значение не изменится, но это знание нам, увы, ничем не поможет; придётся честно завести переменную типа `socklen_t`, присвоить ей зна-

чение `sizeof(int)` и передать её адрес последним параметром вызова. Итоговое обращение к `getsockopt` будет выглядеть так:

```
int opt;
socklen_t optlen = sizeof(opt);

getsockopt(sd, SOL_SOCKET, SO_ERROR, &opt, &optlen);
```

После этого значение переменной `opt` укажет нам статус нашего соединения: ноль будет означать, что соединение установлено, а любое другое значение — что произошла ошибка.

6.4.7. (*) Сигналы в роли событий; вызов `pselect`

Материал этого параграфа не имеет прямого отношения к программированию сетевого взаимодействия; но если вы будете писать событийно-ориентированные программы, построенные на основе вызова `select`, рано или поздно вам встретятся *обрабатываемые сигналы* в роли событий, а вместе с ними — утверждение, что для их корректной обработки вызов `select` «не годится» и нужен загадочный вызов `pselect`. Вопросы вроде «как им пользоваться и зачем он вообще нужен» возникают с завидной регулярностью, причём не только у начинающих; в этом параграфе мы попытаемся прояснить, что тут в действительности к чему.

Припомним для начала, что́ нам известно о сигналах (§5.5.2). Сигнал как таковой несёт на себе только номер (число от 1 до 31). Для всех сигналов, кроме двух (`SIGKILL` и `SIGSTOP`), процесс может установить ***диспозицию*** — указание, что́ должно произойти, если сигнал с таким номером когда-нибудь придёт. Есть три варианта диспозиции: обработка по умолчанию (`SIG_DFL`), игнорирование (`SIG_IGN`) и обработка путём выполнения функции вида

```
void handler(int n)
{
    /* ... */
}
```

Главная программа может узнать, что сигнал пришёл и функция-обработчик была вызвана, только через значения глобальных переменных, причём если делать всё в строгом соответствии с требованиями спецификаций, то такие глобальные переменные могут иметь только один тип — `sig_atomic_t` (на самом деле это обычный `int`).

Если процесс находится в состоянии блокировки в результате обращения к блокирующему системному вызову и в это время ему приходит сигнал, для которого установлена в качестве диспозиции функция-обработчик, то в большинстве случаев системный вызов, в котором находился процесс, немедленно возвращает управление с ошибкой `EINTR`. Ядро операционной системы вынуждено так поступить, чтобы обработать сигнал, не дожидаясь окончания блокировки. Из этого правила есть исключения: некоторые блокирующие вызовы при определённых условиях возвращаются в свою блокировку, когда обработчик сигнала завершает работу, но в целом программа всегда должна быть готова к ошибке `EINTR` на

любом блокирующем системном вызове, коль скоро в программе присутствует обработка сигналов.

Вызов `select` относится к числу таких блокирующих вызовов, которые при поступлении обрабатываемого сигнала обязательно возвращают управление с ошибкой `EINTR`, что в принципе позволяет рассматривать сигналы как разновидность *событий* и реагировать на них, как и на другие события, в главном цикле программы. Проблема в том, что главный цикл состоит отнюдь не только из вызова `select`.

Допустим, наша программа обрабатывает сигнал `SIGUSR1`, для чего предусмотрена глобальная переменная `usr1_caught` и функция-обработчик:

```
sig_atomic_t usr1_caught = 0;
void usr1_handler(int n)
{
    signal(n, usr1_handler);
    usr1_caught++;
}
```

При поступлении сигнала нам нужно выполнить некоторые действия, объединённые функцией `actions_on_usr1`, которую надлежит вызвать из главного цикла. Отметим, что в такой функции — в отличие от функции-обработчика сигнала — мы можем себе позволить использование любых средств, лишь бы они не выполнялись «долго»; дело в том, что `actions_on_usr1` вызывается из главной программы и не может, в отличие от «настоящего» обработчика сигнала, оказаться вызвана «в неожиданный момент». Сам главный цикл схематически можно представить следующим образом:

```
for(;;) {
    /* A */
    res = select( /* ... */ );
    /* B */
    while(usr1_caught--)
        actions_on_usr1();
    /* C */
}
```

где `A`, `B` и `C` — некие фрагменты кода; теперь мы уже можем догадаться, какого рода проблема связана с этим решением. Если сигнал придёт во время выполнения `select`, то есть когда наш процесс будет находиться в состоянии блокировки, а равно во время выполнения фрагмента `B` или даже во время выполнения цикла по переменной `usr1_caught`, всё будет в порядке: управление дойдёт до `actions_on_usr1` на текущей итерации главного цикла, причём разнообразные варианты обработки других событий, в том числе, возможно, наступивших на той же итерации, погоды не сделают — мы ведь помним, что в обработке отдельно взятого события нельзя (по условиям событийно-ориентированной парадигмы) задерживаться «надолго».

Картина существенным образом меняется, если наш процесс получит сигнал в ходе выполнения фрагментов `A` или `C`. В этом случае вызов `select` не будет ничего знать о том, что сигнал уже пришёл, и заблокирует наш процесс до

тех пор, пока не наступит *ещё какое-нибудь событие*; лишь после этого, то есть уже на следующей итерации главного цикла, функция `actions_on_usr1` наконец получит управление. Проблема в том, что очередное событие может не происходить очень долго, и именно на этот (в общем случае неопределённый) срок реакция нашей программы на пришедший сигнал окажется «отложена».

Чаще всего на сигналы «вешают» какое-нибудь изменение режима работы программы, например, повторное считывание конфигурационных файлов и т. п.; системный администратор, когда надо, отправляет нашей программе сигнал с помощью команды `kill`. Если ему «не повезёт» и сигнал придётся на неудачную фазу главного цикла, а никаких новых событий происходить не будет, выглядеть это будет так, как будто наша программа вообще не желает откликаться на сигнал.

Начинающие программисты часто не задумываясь применяют «очевидное» решение, вставляя проверку переменной и обработку события-сигнала прямо перед вызовом `select`:

```
for(;;) {
    /* ... */
    while(usr1_caught--)
        actions_on_usr1();
    res = select( /* ... */ );
    /* ... */
}
```

Программа при этом становится «почти правильной», но ключевое слово тут «почти»: по закону подлости сигнал может прийти как раз между циклом `while` и вызовом `select`. Вероятность этого довольно низкая, но всё же не нулевая.

«Совсем правильное» решение основано на системных вызовах `pselect` и `sigprocmask`; чтобы стало понятно, о чём речь, придётся дать довольно пространные пояснения. Начнём с того, что ядро позволяет процессу временно заблокировать доставку некоторых сигналов; как раз для этого предназначен вызов `sigprocmask`. Если процессу послать сигнал, который у него сейчас заблокирован, то доставка этого сигнала (иначе говоря, выполнение действий, предписанных диспозицией сигнала) будет отложена до тех пор, пока процесс не разблокирует соответствующий сигнал. Сразу же оговоримся, что заблокировать сигналы `SIGKILL` и `SIGSTOP` нельзя.

Профиль системного вызова `sigprocmask` выглядит так:

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Второй и третий параметры используют переменные специального типа `sigset_t`; такая переменная представляет *множество сигналов* подобно тому, как используемые в вызове `select` переменные типа `fd_set` представляют множества файловых дескрипторов. Первый параметр (`how`) указывает, какое действие следует произвести с множеством сигналов, заблокированных на текущий момент, и может иметь одно из трёх значений: `SIG_BLOCK`, `SIG_UNBLOCK` и `SIG_SETMASK`; в первом случае сигналы, входящие в множество `set`, блокируются, во втором, наоборот, разблокируются, а в третьем — блокируются все сигналы, входящие в

set, а все остальные разблокируются. Если адрес, переданный третьим параметром (oldset), не нулевой, то в переменную, находящуюся по этому адресу, вызов запишет множество сигналов, которые были заблокированы на момент обращения к sigprocmask.

Вызовом sigprocmask можно также воспользоваться, чтобы просто узнать, какие сигналы сейчас заблокированы. Для этого вторым параметром передают NULL; первый параметр при этом игнорируется (обычно его указывают равным нулю), а в третий, как обычно, записывается множество заблокированных сигналов, но вызов это множество никак не изменяет.

Для манипуляции множествами сигналов предусмотрены специальные функции:

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

Функция sigemptyset очищает заданное множество, т. е. делает его пустым; sigfillset вносит в множество все существующие сигналы; sigaddset добавляет заданный сигнал в заданное множество, sigdelset удаляет из множества заданный сигнал; sigismember позволяет узнать, входит ли данный сигнал в множество — она возвращает 1, если сигнал в множество входит, 0 — если не входит, -1 в случае ошибки. Остальные функции возвращают 0, если всё в порядке, и -1 в случае ошибки. Отметим, что ошибка тут может быть только одна: параметр signum не является номером существующего сигнала; переменная errno получает при этом значение EINVAL.

Вызов pselect отличается от select наличием ещё одного, шестого по счёту параметра, имеющего тип sigset_t; этот параметр указывает, какие сигналы должны быть заблокированы на время работы вызова. Сигналы, не входящие в указанное множество, на время работы pselect разблокируются. При возврате управления pselect восстанавливает множество заблокированных сигналов в том виде, в котором оно было на момент входа в вызов. Ещё одно отличие состоит в том, что pselect, согласно его спецификации, не изменяет содержимое структуры timeout, передаваемой пятым параметром (напомним, что select в некоторых системах оставляет этот параметр нетронутым, но в других — записывает в него время, оставшееся до истечения тайм-аута, так что для обеспечения переносимости программы приходится предполагать, что содержимое структуры просто портится). Профиль pselect выглядит так:

```
int pselect(int n, fd_set *readfds, fd_set *writefds, fd_set *exfds,
            const struct timeval *timeout, const sigset_t *sigmask);
```

Начинающие программисты часто не понимают, как пользоваться вызовом pselect, поскольку его профиль и описание создают (ложное) впечатление, что параметр sigmask предназначен, чтобы *заблокировать* некоторые сигналы на время работы вызова. В действительности роль этого параметра прямо противоположна: он используется, чтобы на время работы pselect некоторые сигналы *разблокировать*, а потом заблокировать обратно.

Общая идея решения проблемы, описанной выше (с сигналами, приходящими «не вовремя»), состоит в том, чтобы сигнал, который надлежит обрабатывать в качестве события, был заблокирован в течение всего времени выполнения, *кроме* времени, которое программа проводит в вызове pselect; тогда сигнал может быть доставлен процессу только во время работы этого вызова и, как следствие, приводит к выходу из него и выполнению итерации главного цикла. Например, если наша программа обрабатывает в качестве события сигнал SIGUSR1, схема главного цикла будет выглядеть так:

```
sigset_t mask, orig_mask;
/* ... */
signal(SIGUSR1, usr1_handler);
sigemptyset(&mask);
sigaddset(&mask, SIGUSR1);
sigprocmask(SIG_BLOCK, &mask, &orig_mask);

/* ... */
for(;;) {
    /* ... */
    res = pselect( /* ... */ , &orig_mask);
    /* ... */
    while(usr1_caught--)
        actions_on_usr1();
    /* ... */
}
```

Как видим, SIGUSR1 остаётся заблокированным всё время работы программы, за исключением времени, проводимого ею в вызове pselect, так что прийти «не вовремя» сигнал не может: доставлен он будет в любом случае только во время работы pselect.

Справедливости ради отметим, что во многих случаях все эти сложности оказываются не нужны. К примеру, если ваша программа, построенная на обычном вызове select, в силу особенностей решаемой задачи «просыпается» через каждую десятую (или какую-то другую) долю секунды по тайм-ауту, то время, которое пройдёт между доставкой сигнала и его обработкой, даже в самом неудачном случае будет разве что чуть-чуть (на время работы других обработчиков) превышать эту долю секунды; скорее всего, это окажется приемлемо. Тем не менее, в любом случае полезно понимать суть проблемы сигналов, приходящих «не вовремя», и принцип предложенного решения на основе pselect и setprocmask; во-первых, это поможет понять другие подобные проблемы и то, как их решать (как мы увидим позже, таких проблем не так уж мало), а во-вторых, понимание описанной в этом параграфе механики и умение рассказать всё это на собеседовании может сильно повысить вашу привлекательность для потенциальных работодателей.

Часть 7

Параллельные программы и разделяемые данные

Эта часть книги будет посвящена *параллельному программированию*, то есть такому подходу к написанию программы, при котором задача разбивается на подзадачи, выполняющиеся *одновременно*. Такие подзадачи часто оформляются в виде уже упоминавшихся ***легковесных процессов*** (***тредов***); с точки зрения программиста тред предстаёт в виде возможности написать некую функцию и запустить её на выполнение параллельно с основной программой, причём с сохранением доступа ко всему, к чему имеет доступ основная программа, в том числе, например, к глобальным переменным или к структурам данных, которые сформированы в основной программе и в других тредах.

Программирование с использованием тредов в английском языке обозначается словом *multithreading*. С переводом этого слова на русский имеются определённые проблемы. Английское слово *thread* в программистском контексте чаще всего так и «переводят» путём транслитерации, то есть говорят попросту о *тредах*, не утруждая себя подбором адекватного русского термина; именно так поступаем и мы в нашей книге. Но со словом *mutlithreading* этот номер проходит несколько хуже — точнее, можно сказать, что не проходит совсем из-за его тяжеловесности. Наиболее часто встречающийся русскоязычный термин — ***многопоточное программирование***. Подразумевается, что легковесный процесс обозначается словом *поток* (управления), что частично оправдывается существованием в английском языке словосочетания *control flow*, которое с некоторой натяжкой можно перевести как «поток управления»; хорошо знакомые нам блок-схемы по-английски называются *flow charts*.

Всё бы хорошо, но попытка перевести слово *thread* как «поток» натыкается на другую лингвистическую проблему. Английское слово *stream* тоже переводится на русский язык как «поток», причём если мы попытаемся передать смысловую разницу между *stream* и *flow*, то *flow* придётся переводить скорее не как «поток», а как «течение». При этом слово *stream*, совершенно законно переведённое

именно как *поток*, в программировании активно используется в значении «поток данных» — так, мы уже хорошо знакомы с понятием *потока ввода-вывода*. Попросту говоря, русское слово *поток* в программистском контексте давно и прочно занято.

Несмотря на это, термин «многопоточное программирование» в русском языке вполне прижился; а вот термин «поток» для обозначения отдельно взятого треда приживаться не хочет, и это вполне можно понять.

С появлением и массовым распространением *многоядерных процессоров*, представляющих собой фактически несколько независимых процессоров в одной микросхеме, тематика параллельных вычислений стала настолько популярной, что возможности, ориентированные исключительно на распараллеливание программы, проникли даже в стандарты языков программирования, включая Си и Си++, окончательно эти стандарты испортив. На самом деле в подавляющем большинстве задач, решаемых с помощью компьютеров, видимая скорость выполнения определяется не процессорным временем, а такими факторами, как время отклика в компьютерной сети, скорость обмена с внешними запоминающими устройствами, пропускная способность шины компьютера, но не количество доступной вычислительной мощности центральных процессоров, которой как раз обычно достаточно. При этом от многоядерных процессоров оказывается на удивление мало пользы. Задачи, в принципе допускающие распараллеливание на независимые последовательности вычисления, вообще встречаются крайне редко, поскольку при решении большинства задач последующие шаги зависят от результатов предыдущих шагов и не могут начаться раньше, нежели эти результаты будут получены. Ещё реже встречаются программисты, умеющие эффективно выделять независимые подпоследовательности из общего алгоритма и разносить их на отдельные потоки управления так, чтобы от этого был какой-то ощутимый положительный эффект. Едва ли не единственная область программистской деятельности, в которой параллельные вычисления действительно нужны — это числовые расчёты, в большинстве случаев связанные с математическим моделированием физических явлений; с такими задачами мы обычно сталкиваемся в суперкомпьютерных вычислительных центрах, принадлежащих научным организациям. Обычный пользователь с объёмными расчётами может встретиться разве что при обработке собственноручно отснятого цифрового видео, но этим занимаются далеко не все; писать же *программы*, предполагающие преобразование видеоданных, приходится совсем небольшому количеству программистов. Самое интересное, что даже в такой ситуации может оказаться предпочтительным использование полноценных процессов, а не тредов.

Удачным примером использования легковесных процессов может служить интерактивная программа, работающая со сжатыми данными (например, музыкальный редактор, способный работать с mp3-файлами).

С одной стороны, операции упаковки/распаковки данных требуют больших объемов вычислений и могут длиться по несколько минут и даже десятков минут. С другой стороны, интерактивная программа не может себе позволить в течение нескольких минут не реагировать на действия пользователя: например, пользователь может переместить окно программы на экране (что потребует его отрисовки) или даже принять решение об отмене текущей операции. В такой ситуации логично использовать один легковесный процесс для выполнения вычислений (упаковки/распаковки) и другой — для обработки интерфейсных событий, то есть для реакции на действия пользователя.

Отметим, что действительно удачных примеров, когда легковесные процессы реально применяются «по делу», очень мало; привести второй пример в дополнение к вышеописанному не так просто, как кажется. Существует точка зрения, что легковесные процессы становятся очень актуальны в связи с развитием «многоядерных» процессоров, якобы при этом легковесные процессы позволяют распределить выполнение одной программы на несколько процессоров и таким образом ускорить её работу. В действительности мало какую программу вообще можно эффективно распараллелить: для этого в ней должны производиться серьёзные вычисления, что встречается не так часто. Большинство программ, написанных с использованием легковесных процессов, запускает их лишь затем, чтобы они почти тут же остановились в состоянии блокировки и ждали результатов работы друг друга.

Гораздо чаще многопоточное программирование используется, как говорится, совершенно не по делу. В §6.4.3 мы рассматривали пример ситуации, в которой имеются *независимые источники внешних событий*, и эти события нуждаются в своевременной обработке. При этом внешние события могут выстраиваться в логические цепочки, такие, например, как вопросы, заданные пользователю-человеку, и получение ответов на эти вопросы; если такая цепочка у нас одна, мы можем написать последовательность операторов, реализующих, например, логику диалога: задать вопрос, дождаться ответа, обработать ответ, задать следующий вопрос, дождаться ответа и так далее. При работе с несколькими независимыми источниками событий мы не можем себе позволить роскошь *дожидаться* чего-то конкретного, поскольку если мы будем так делать, то все наши клиенты кроме одного — того, ответа от которого мы ждём — будут вынуждены ждать уже нас.

Решение, которое мы предложили в предыдущей части книги, состоит в том, чтобы построить приложение в виде *цикла обработки событий*, на каждой итерации которого мы сначала производим выборку события, то есть узнаём, из какого источника пришло очередное событие и принимаем это событие (например, читаем данные из потока ввода), после чего обрабатываем полученное событие. Мы назвали такой стиль ра-

боты ***событийно-управляемым программированием***. При этом во время обработки событий мы не имеем права делать ничего такого, что может нашу программу заблокировать; в частности, производить чтение из потоков ввода мы можем лишь тогда, когда уверены, что данные туда уже пришли. Не можем мы также выполнять и «большие» секции вычислений, занимающие ощутимое время — при необходимости их приходится разбивать на отдельные части и выполнять порциями, при условии, что новых событий не поступило. Для каждой логической цепочки связанных внешних событий — например, для каждого сеанса обслуживания отдельно взятого клиента — мы при этом должны помнить в явном виде состояние дел, всё то же «на чём мы остановились». Часто при получении очередного события мы только меняем соответствующим образом состояние и больше ничего не делаем, поскольку для действий время ещё не пришло. Событийно-ориентированное програмирование требует от программиста определённого мастерства, поскольку мышление в терминах явного выделения состояния несколько отличается от традиционного мышления в терминах последовательностей действий.

Довольно часто встречаются программы, авторы которых то ли не умеют работать в терминах явных состояний, то ли вообще не знают о возможности событийно-ориентированного построения программы, зато, к несчастью, знают о возможности запуска тредов. Проблему с независимыми источниками событий они решают очень просто: на каждый такой источник запускают свой собственный тред. «Параллельная» программа, написанная таким образом, обычно имеет весьма характерную особенность: *все её потоки управления, сколько бы их ни было, дружно «стоят»*, заблокировавшись на очередной операции чтения или в ожидании другого события. Естественно, такое построение программы не даёт и не может дать никакого выигрыша в быстродействии. Квалифицированный программист не станет применять легковесные процессы в качестве хранилища состояния, поскольку на это тратится слишком много памяти, причём как памяти задачи, так и памяти ядра системы. С другой стороны, кажущаяся простота многопоточной обработки событий оборачивается неожиданно серьёзными проблемами, имманентно присущими параллельному программированию.

Важнейшей особенностью параллельного программирования является возникновение ***разделяемых данных***, то есть таких данных, к которым одновременно имеют доступ два и более независимых «действующих лица», будь то процессы или треды. Обычно это переменные в памяти нашего процесса, но с таким же успехом это могут быть данные, хранящиеся в файлах на диске; проблема разделяемых данных при этом никуда не исчезает.

Многопоточное программирование требует определённого мастерства и аккуратности, как и любое программирование с использованием разделяемых данных. При появлении разделяемых данных приходится принимать меры, чтобы разные процессы никогда не осуществляли доступ к разделяемым данным одновременно; для этого вводится понятие ***критической секции*** (участка программы, где происходит работа с разделяемыми данными) и создаются специальные средства ***взаимного исключения***, благодаря которым процессы не попадают в критическую секцию, когда в этой же секции находится другой процесс.

Проблемы, которые возникают при работе с разделяемыми данными, примечательны уже тем, что их *довольно трудно осознать*; многие программисты, использующие многопоточную модель программирования, при этом не понимают, зачем нужны средства взаимоисключений и прочие нетривиальные концепции, сопровождающие поддержку тредов. Вторая особенность проблем с разделяемыми данными — в сравнительно низкой вероятности их проявления. Неправильно написанная многопоточная программа может довольно долго работать без ошибок, а подведёт, естественно, в самый неподходящий момент.

В статье Википедии, посвящённой многопоточному программированию, приводится удачная цитата из работы [12] профессора университета Беркли Эдварда Ли:

> Хотя потоки выполнения кажутся небольшим шагом [в сторону] от последовательных вычислений, по сути этот шаг огромен. Они отказываются от наиболее важных и привлекательных свойств последовательных вычислений: понятности, предсказуемости и детерминированности. Потоки выполнения как модель вычислений страшно недетерминированы; работа программиста превращается в отстригание этого недетерминизма.

Впрочем, даже если принять решение об отказе от многопоточного программирования, квалифицированный программист в любом случае должен понимать, что за проблемы возникают при работе с разделяемыми данными — хотя бы затем, чтобы точно знать, *почему* не следует применять многопоточное программирование без крайней необходимости, и уметь аргументированно показать неправильность чужой программы. Кроме того, иногда можно встретить ситуации, в которых появление разделяемых данных неизбежно. Например, именно так обстоят дела внутри ядер операционных систем, если, конечно, система способна использовать больше одного процессора (а таковы *все* современные системы).

Вообще говоря, для появления разделяемых данных даже не обязательно сознательно пытаться написать ту или иную программу в

«параллельном» виде. Нередко бывает, что к одним и тем же данным вынуждены обращаться совершенно разные программы. Хрестоматийный пример — файл почтового ящика на сервере электронной почты, куда почтовый сервер должен дописывать новые письма, а клиентским программам и серверам финальной доставки почты (pop3/imap) в то же самое время нужен доступ к этому файлу для чтения писем, а также для их удаления по требованию пользователя. Но здесь всё довольно просто: каждая из программ, обращающаяся к файлу почтового ящика, использует системные средства для извещения остальных о том, что файл сейчас находится в работе и лезть к нему не надо. Намного жёстче ситуация с базой данных, доступ к которой нужен многим людям одновременно; количество таких людей варьируется от двух-трёх (например, несколько бухгалтеров, совместно обрабатывающих поступающие документы на предприятии) до сотен тысяч (система бронирования авиабилетов). Очевидно, что здесь проблема уже не в том, какие средства мы выбираем для работы, а в самой задаче: разделяемые данные могут быть свойством предметной области. Если это так, то избежать работы с разделяемыми данными мы никак не сможем.

7.1. О работе с разделяемыми данными

7.1.1. Ситуация гонок (race condition)

Рассмотрим простой пример, подразумевающий доступ к разделяемым данным. Два различных процесса имеют доступ к разделяемой памяти, в которой содержится целочисленная переменная. Пусть её начальное значение равно 5. Оба процесса в некий момент пытаются увеличить значение переменной на 1, так что в итоге переменная должна получить значение 7. Для этого нужно загрузить значение из памяти в регистр, увеличить значение регистра на 1 и выгрузить его значение обратно в память[1]:

```
mov eax, [var]
inc eax
mov [var], eax
```

Представим, что первый процесс выполнил команду `MOV` для загрузки значения переменной в регистр, то есть в регистре теперь содержится

[1]Как мы знаем, процессоры семейства i386 могут выполнять некоторые арифметические операции непосредственно в памяти, тогда этот пример не сработает, т. к. каждая инструкция процессора выполняется атомарно. Это, однако, не означает неправильности примера: представьте себе, что число хранится в памяти в виде строкового представления — в этом случае операция его приращения никак не сможет быть атомарной. Кроме того, i386 — не единственная в мире архитектура; существуют процессоры, имеющие лишь две операции над областями памяти — загрузку значения из памяти в регистр и выгрузку значения из регистра в память.

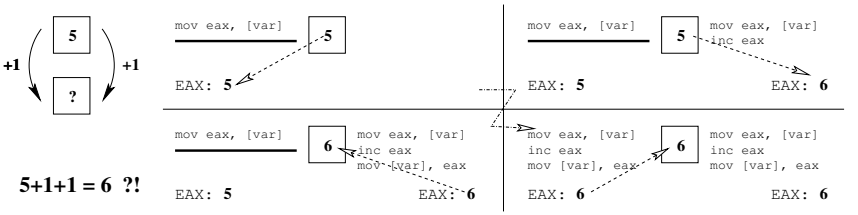

Рис. 7.1. Одновременное увеличение разделяемой переменной

число 5, и в этот момент произошло прерывание по таймеру, в результате которого процесс оказался снят с исполнения и помещён в очередь процессов, готовых к выполнению (см. рис. 7.1). В это время второй процесс также выполняет команду `MOV` (в его регистре `EAX` теперь находится значение 5), потом команду `INC` для увеличения значения в регистре (получается 6) и команду `MOV` для выгрузки содержимого регистра в память. Переменная теперь равна 6.

Между тем первый процесс дождался своей очереди и снова поставлен на выполнение. Первую команду `MOV` он уже выполнил, и в его регистре сейчас число 5. Он выполняет команду `INC` (в регистре теперь 6) и команду `MOV` (значение 6 выгружается в память). Получается, что в итоге переменная получила значение 6, а не 7. Заметим, что значением было бы именно 7, если бы первый процесс не оказался прерван после первой команды. Узнать, был процесс прерван или нет, вообще говоря, невозможно, как и запретить прерывать процесс. Получается, что **конечный результат зависит от того, в какой конкретно последовательности произойдут события в независимых процессах**. С подобным мы уже сталкивались ранее (см. §5.4.3) и знаем, что это называется ***ситуацией гонок*** или ***состязаний*** (англ. *race condition*).

Рассмотрим более серьёзный пример. Имеется база данных, содержащая остатки денег на банковских счетах. Допустим, на счетах под номерами 301, 515 и 768 содержится соответственно \$1000, \$1500 и \$2000. Один оператор проводит транзакцию по переводу суммы в \$100 со счёта 301 на счёт 768, а другой — транзакцию по переводу \$200 со счёта 768 на счёт 515 (см. рис. 7.2, *а, б*). Если эти действия провести в разное время, остатки на счетах составят, понятно, \$900, \$1700 и \$1900.

Теперь предположим, что процесс, запущенный вторым оператором, был прерван после операции чтения остатков на счетах 515 и 768. Прочитанные остатки (соответственно \$1500 и \$2000) процесс сохранил в своих переменных, и именно в этот момент случилось прерывание таймера, в результате которого управление получил процесс, запущенный первым оператором. Этот процесс произвёл чтение остатков счетов 301 и 768 (\$1000 и \$2000), вычислил новые значения остатков (\$900 и \$2100)

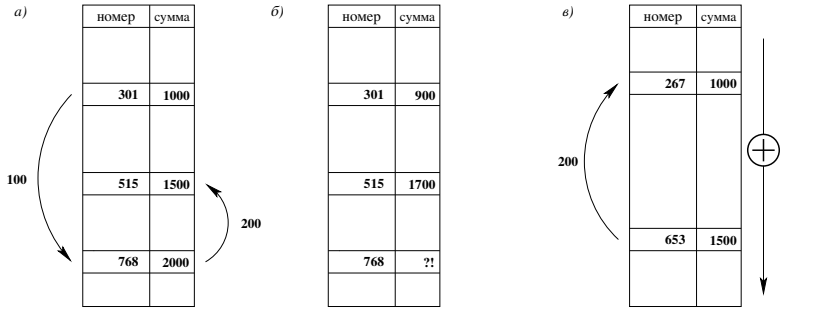

Рис. 7.2. Пример: банковские переводы

и записал их в базу данных, после чего завершился. Затем в системе снова был запущен на выполнение первый процесс, уже прочитавший остатки; он вычисляет новые остатки для счетов 515 и 768 на основе ранее прочитанных (\$1500 и \$2000), получая \$1700 и \$1800. Именно эти остатки он и записывает в базу данных.

В итоге владелец счёта 768 обнаружит на счету недостаток суммы в \$100 (остаток окажется равен \$1800 вместо \$1900). Как нетрудно убедиться, общая сумма остатков на счетах 301, 515 и 768 уменьшилась на \$100, то есть деньги клиента попросту бесследно исчезли. Могло получиться и наоборот: если бы после чтения был прерван первый процесс, а второй в это время выполнил свою операцию, клиент обнаружил бы на своём счету лишние \$200 (\$2100 вместо \$1900). Клиент, возможно, порадуется, а акционеры банка — вряд ли.

Представим теперь третий процесс, вычисляющий общую сумму остатков на счетах всех клиентов банка. Очевидное решение — прочитать в цикле остатки на каждом счёте и просуммировать их. Однако если во время работы этого процесса какой-нибудь другой процесс произведёт перевод денег с одного из счетов в конце таблицы, сумма которого ещё не учтена, на один из счетов в начале, то есть на такой, баланс которого уже попал в суммирование (см. рис. 7.2, *в*), полученная в результате итоговая общая сумма будет отличаться от реальной как раз на сумму транзакции по переводу, поскольку переведённые деньги окажутся не попавшими в подсчёт ни на новом, ни на старом месте — на новый счёт они пришли уже после того, как он был учтён, а со старого сняты до того, как он был учтён. Как видим, **нежелательные эффекты возможны даже тогда, когда один из участников не производит никаких изменений, а только читает данные**.

Причина получения неправильных результатов в наших примерах в том, что **вычисленный результат основывается на исходных данных, которые к моменту окончания вычислений уже устарели**. В примере с целочисленной переменной (стр. 250) второй процесс

записывает в разделяемую переменную значение, вычисленное прибавлением единицы к значению, которое само по себе уже нельзя считать правильным, поскольку его изменил другой процесс. В примере с двумя «встречными» переводами денег (стр. 251) та транзакция, которая завершается позднее, вычисляет новый остаток на счёте 768, пользуясь устаревшими сведениями об исходном количестве денег на нём. Наконец, в примере с сумматором к моменту окончания суммирования успевает устареть информация об остатке на том счёте, который, находясь ближе к началу таблицы, успел поучаствовать в посторонней транзакции между моментом, когда его остаток был учтён в суммировании (то есть повлиял на результат), и моментом окончания суммирования.

7.1.2. Взаимоисключения. Критические секции

Ситуации гонок, очевидно, можно избежать, если сделать так, чтобы процессы, обращающиеся к разделяемым данным, не могли работать одновременно: например, пока работает один процесс, обращающийся к базе данных, другой процесс, которому также нужна эта база данных, блокируется при запуске до тех пор, пока первый не завершится. Такое решение представляется неудачным, поскольку внесение изменений в базу данных может быть не единственной задачей процесса. Вполне возможно, что процесс бóльшую часть времени занимается работой, никак с разделяемыми данными не связанной, причем время жизни процесса может оказаться слишком большим, чтобы на всё это время запрещать доступ к данным кому бы то ни было ещё. Вполне можно представить себе постоянно работающий сервер, с помощью которого клиенты банка могут узнавать остатки на своих счетах. Такой сервер сравнительно редко обращается к базе данных, но при этом работать может месяцами без перерыва. Ясно, что блокировать на всё это время доступ к базе данных других процессов нельзя, ведь это парализовало бы работу банка. В связи с этим вводится понятие ***критической секции***, которое нуждается в дополнительных пояснениях.

Проблема, которую мы решаем, состоит в возможности получения и, возможно, записи обратно в разделяемую область данных такого значения, которое вычислено на основе других значений, уже, возможно, устаревших к моменту окончания вычисления. Следовательно, избежать проблем мы можем, если каким-то образом предотвратим устаревание разделяемых данных, которые мы уже начали использовать в своих вычислениях — до окончания вычислений. Интересно, что момент окончания вычислений в разных случаях определяется по-разному. Если вычисляемое нашей программой значение должно быть записано обратно в разделяемое хранилище, где его могут использовать для дальнейших вычислений другие процессы, то вычисления мы можем считать оконченными не раньше, чем будет завершена запись в храни-

лище последнего из вычисленных результатов. Если же вычисляемый результат предназначен для передачи куда-то «наружу», то есть на нём не будут основываться дальнейшие вычисления в нашей системе — как это подразумевается в примере с вычислением суммы по всем счетам, — то такой результат «имеет право» устареть даже раньше, чем мы предъявим его пользователю, и пользователь, как правило, об этом осведомлён; единственное ограничение здесь в том, чтобы результат не основывался на устаревших сведениях *на момент его получения*, или, иначе говоря, был правильным хотя бы по состоянию на *какой-то* момент времени. Момент окончания вычислений в этом случае обычно определяется как момент, когда последнее из вычисляемых значений вычислено до конца; но можно подойти к вопросу более аккуратно и заметить, что коль скоро само по себе вычисляемое значение «имеет право устареть», можно считать критическую секцию оконченной, когда *прочитано* последнее из значений разделяемых данных, участвующее в вычислении результата.

Можно сказать, что **под критической секцией понимается такая часть программы, в которой производятся логически связанные манипуляции с разделяемыми данными**. Иначе говоря, критическая секция начинается в момент, когда прочитано первое из значений разделяемых данных, способных повлиять на результаты вычисления, и заканчивается, когда вычислены все значения, которые предполагалось вычислить (либо как минимум для них уже получены все исходные данные), и те из значений, которые должны быть записаны обратно в разделяемое хранилище, уже не только вычислены, но и записаны. Так, в примерах предыдущего параграфа критическая секция в процессах, увеличивающих переменную на единицу, начинается перед чтением исходного значения переменной и заканчивается после записи нового значения; критическими секциями в процессах, осуществляющих перевод денег со счёта на счёт, являются действия с момента чтения первого остатка до момента записи последнего остатка; в процессе, подсчитывавшем сумму всех остатков, критическая секция начинается в момент начала просмотра таблицы и заканчивается с его окончанием.

Понятие критической секции очевидным образом дуалистично: речь может идти о *фрагменте программного кода* (от этого места в программе до вот этого), а может — о *временно́м интервале* (с момента начала выполнения вот этого действия до момента окончания вот этого). Конечно, эти две ипостаси понятия критической секции прочно связаны друг с другом: критической секцией во втором смысле оказывается тот временной интервал, в течение которого выполняется критическая секция в первом смысле. Из контекста обычно понятно, о чём идёт речь.

Отметим, что о критической секции можно говорить только в применении к конкретным разделяемым данным. Так, если два процесса

обращаются к *разным* данным, пусть и разделяемым с кем-то ещё, друг другу они при этом помешать не могут. Поэтому часто можно встретить выражение *критическая секция по переменной* x или *критическая секция по файлу* f и т. п. Кроме того, важно отличать критические секции, в которых разделяемые данные только читаются, от критических секций, предполагающих изменения разделяемых данных. Последние обычно называют ***модифицирующими критическими секциями***.

Говорят, что две критические секции *конфликтуют* между собой, если среди разделяемых данных, которые в них затрагиваются, есть такие, которые затрагиваются в обеих критических секциях, и при этом минимум одна по этим данным является модифицирующей. Если в двух критических секциях осуществляется доступ к одним и тем же данным, но только на чтение, то между ними никакого конфликта нет и они могут выполняться одновременно.

Такая организация работы процессов, при которой два (и более) процесса не могут одновременно находиться в критических секциях, конфликтующих между собой, называется ***взаимным исключением*** (англ. *mutual exclusion*). Следует учитывать, что взаимные исключения могут породить дополнительные проблемы, так что при выборе метода их реализации нужно соблюдать известную осторожность.

Остановимся на одном моменте, который часто вызывает сложности у студентов. **Процессы, участвующие во взаимоисключении, следуют соглашениям добровольно**. Средства взаимоисключения не создают никаких физических препятствий в обращении к разделяемым переменным; напротив, процессы *самостоятельно* проверяют, разрешён ли им доступ, и лишь после этого такой доступ осуществляют. Иначе говоря, процесс, имеющий доступ к разделяемым данным, *может* обратиться к ним в любой момент, ему ничто (внешнее) не мешает это сделать; просто программы, исполняемые в этих процессах, написаны так, чтобы (сугубо добровольно!) выполнять мероприятия для взаимного исключения и не лезть в критическую секцию, покуда это не окажется позволено в соответствии с используемой моделью взаимоисключения.

Сформулируем требования, налагаемые на механизм взаимного исключения:

- два и более процесса не должны ни при каких условиях находиться одновременно в критических секциях, конфликтующих между собой;
- в программе не должно быть никаких предположений о скорости выполнения процессов и о количестве процессоров в системе[2];

[2]Если такие предположения есть, в условиях мультизадачной системы всегда может получиться ситуация, которая в эти предположения не впишется: например, один из процессов может оказаться приостановлен сразу после начала очередного кванта времени из-за наличия в системе процесса с более высоким приоритетом, а

- процесс, находящийся вне критических секций, не должен при этом быть причиной блокировки других процессов;
- недопустима ситуация «вечного ожидания», при которой некоторый процесс никогда не получит доступ в нужную ему критическую секцию (такая ситуация обычно называется ***ресурсным голоданием***, англ. *resource starvation*);
- процесс, заблокированный в ожидании разрешения на вход в критическую секцию, не должен расходовать процессорное время, то есть не должно использоваться активное ожидание[3].

7.1.3. Взаимоисключение с помощью переменных

Все подходы, перечисленные в этом параграфе, используют активное ожидание. Специально акцентировать на этом внимание мы не будем, тем более что у этих подходов есть и другие недостатки. Есть у них, впрочем, и достоинство, хоть и всего одно, зато несомненное: для их работы не требуется ничего, кроме обычных переменных, расположенных в разделяемой памяти.

Начнём с подхода, который кажется работающим лишь на первый взгляд, а в действительности взаимоисключения не обеспечивает. Подход называется «**блокировочная переменная**». Пусть имеются некоторые данные, доступ к которым имеют несколько процессов. Заведём в разделяемой памяти целочисленную переменную (будем называть ее `s`); значение переменной `1` будет означать, что с разделяемыми данными никто не работает, а значение `0` — что один из процессов в настоящее время работает с разделяемыми данными и надо подождать, пока он не закончит. При запуске системы присвоим переменной `s` значение `1`. Доступ к данным организуем следующим образом:

```
while(s == 0) {} /* пустой цикл, пока нельзя входить
                                 в критическую секцию */
s = 0;          /* запретили доступ другим процессам */

section(); /* ... работа с разделяемыми данными ... */

s = 1;                            /* разрешили доступ */
```

Предположим, что все процессы, которым нужен доступ к этим данным, будут следовать такой схеме. Возникает ощущение, что оказаться одновременно в критической секции (в примере она показана вызовом функции `section`) два процесса не могут — ведь если один из процессов

другому процессу при этом «повезёт» больше, в итоге он будет выполняться в разы (или даже на несколько порядков) быстрее.

[3] В реальности это правило иногда нарушается, если есть уверенность в небольшом времени ожидания; по возможности, однако, активного ожидания следует избегать.

собрался войти в секцию, он предварительно заносит `0` в переменную `s`, что заставит любой другой процесс, имеющий намерение зайти в критическую секцию, подождать, пока первый процесс не закончит работу в секции и не занесет в `s` снова значение `1`.

К сожалению, всё не так просто. Выполнение процесса может быть прервано (например, прерыванием таймера, в результате которого планировщик сменит текущий процесс) точно в тот момент, когда он уже «увидел» число `1` в переменной `s` и вышел из цикла `while`, но присвоить переменной значение `0` не успел. В этом случае другой процесс может также «увидеть» значение `1`, присвоить `0` и войти в секцию; затем управление будет возвращено первому процессу, но ведь проверку значения он уже выполнил, так что он также произведёт присваивание нуля и войдёт в секцию. В результате оба процесса окажутся в секции одновременно, то есть произойдет то, чего мы пытались избежать.

Пример блокировочной переменной иллюстрирует потребность в ***атомарности*** некоторых действий при организации взаимоисключения. В самом деле, если бы цикл `while` и присваивание `s = 0` в приведённом примере выполнялись как одна неделимая операция без возможности прервать её на середине, то проблема бы не возникла.

Логично приходит в голову идея о *запрете внешних (аппаратных) прерываний* на время выполнения критической секции. К сожалению, этот вариант неприемлем по целому ряду причин. Во-первых, запрет прерываний годится лишь для кратковременных критических секций: длительное запрещение прерываний нарушит работу аппаратуры — перестанут приниматься и передаваться данные по сети, прекратится запись и чтение файлов и т. д.

Во-вторых, запрет прерываний пригоден только для работы с данными, находящимися в оперативной памяти, поскольку для любого обмена с внешними устройствами (в том числе для чтения и записи файлов) аппарат прерываний должен работать. Заметим, при этом нужно тем или иным способом гарантировать, что данные действительно находятся в памяти, ведь если процесс запретит прерывания, а потом обратится в область памяти, в настоящее время откачанную на диск, операция подкачки либо просто не будет выполнена (что приведет к аварии), либо операционная система всё-таки разрешит прерывания, что может, в свою очередь, привести к получению управления другим процессом и нарушению взаимного исключения. В-третьих, запрет прерываний обычно касается только одного процессора. В системе с несколькими процессорами это эффекта не даст.

Далее, в комбинации с активным ожиданием запрет прерываний (в однопроцессорной системе) приведёт к зависанию системы, ведь при запрещённых прерываниях ни один другой процесс (в том числе и ви-

```
for(;;) {
    while(turn != 0) {}
    section();
    turn = 1;
    noncritical_job();
}
```

```
for(;;) {
    while(turn != 1) {}
    section();
    turn = 0;
    noncritical_job();
}
```

Рис. 7.3. Взаимоисключение на основе чередования

новник блокировки) не может получить управление и снять блокировку, исчезновения которой мы активно ожидаем.

Наконец (и это, пожалуй, самое важное соображение) допускать запрет прерываний пользовательскими процессами слишком опасно. Даже если исключить злой умысел со стороны пользователей, что уже само по себе странно для многопользовательской системы, остаётся возможность случайных ошибок. Если, к примеру, процесс, запретивший прерывания, случайно зациклится, система в результате этого «повиснет» и её придется перезагружать. Поэтому запрет прерываний считается привилегированным действием и для пользовательских процессов недоступен. Отметим, что ядро ОС при этом само достаточно часто использует кратковременные запреты прерываний для обеспечения атомарности некоторых операций; как мы увидим позже, в современных системах обеспечение взаимоисключения иногда возлагается на ядро ОС.

Следующий любопытный способ взаимоисключения, называемый ***чередованием***, заключается в том, что процессы по очереди передают друг другу право работы с разделяемыми данными на манер эстафетной палочки. На рис. 7.3 показаны два процесса, обращающиеся к разделяемым данным (функция `section`) в соответствии с *маркером чередования*, который хранится в переменной `turn`. Значение `0` означает, что право на доступ к разделяемым данным имеет первый процесс, значение `1` соответствует праву второго процесса. Завершив работу в критической секции, процесс «передаёт ход» другому процессу и приступает к выполнению действий, не требующих доступа к разделяемым данным (функция `noncritical_job`).

Такой способ действительно не даёт процессам оказаться в критической секции одновременно, но имеет, к сожалению, другой недостаток. Если один из процессов, передав ход другому, быстро выполнит все некритические действия и снова попытается войти в критическую секцию, может получиться так, что второй процесс в это время до своей критической секции так и не дошел (и, соответственно, не передал ход первому процессу). В результате второй процесс, не нуждаясь в доступе к разделяемым данным, тем не менее будет мешать работать перво-

```
void enter_section() {
    interested[0] = TRUE;
    who_waits = 0;
    while(who_waits==0 &&
          interested[1]) {}
}
void leave_section() {
    interested[0] = FALSE;
}
```

```
void enter_section() {
    interested[1] = TRUE;
    who_waits = 1;
    while(who_waits==1 &&
          interested[0]) {}
}
void leave_section() {
    interested[1] = FALSE;
}
```

Рис. 7.4. Алгоритм Петерсона

му процессу, то есть нарушится второе из сформулированных выше условий.

От недостатков предыдущих подходов избавлен ***алгоритм Петерсона***, рассчитанный на случай двух процессов; эти процессы у нас будут фигурировать под номерами `0` и `1`. Аналогичные алгоритмы существуют и для произвольного числа процессов — например, к таковым относится известный «алгоритм булочной» (*bakery algorithm*), придуманный Лесли Лампортом; рассматривать его мы не будем, ограничившись алгоритмом Петерсона — он позволяет почувствовать общий принцип. Для взаимоисключения нам в этот раз потребуется создать в разделяемой памяти массив из двух логических переменных `interested[2]`, показывающих, нуждается ли соответствующий (нулевой или первый) процесс в выполнении критической секции; во время выполнения секции соответствующее логическое значение также будет истинным. Кроме того, введём (также в разделяемой памяти) переменную `who_waits`, которая будет показывать, который из процессов в случае столкновения должен подождать завершения критической секции второго.

Показав свою заинтересованность во входе в критическую секцию, то есть присвоив значение «истина» соответствующему элементу массива `interested`, процесс заявляет о своей готовности подождать, если нужно, занеся в переменную `who_waits` свой номер. Затем он будет ждать до тех пор, пока либо не изменится номер `who_waits` (это означает, что второй процесс проявил готовность подождать), либо второй процесс не окажется *не заинтересован* во вхождении в секцию.

На рис. 7.4 алгоритм Петерсона показан в виде двух процедур: `enter_section()` (вход в критическую секцию) и `leave_section()` (выход из критической секции). Единственным недостатком алгоритма Петерсона и более сложных алгоритмов, построенных на этой идее, таких как алгоритм булочной, оказывается активное ожидание. К сожалению,

этого вполне достаточно, чтобы считать эти решения неприемлемыми для большинства возникающих ситуаций.

7.1.4. Понятие мьютекса

Подходы к построению взаимного исключения, перечисленные в предыдущем параграфе, характерны наличием ***активного ожидания*** — такого состояния процесса, при котором он в ожидании момента, когда можно будет войти в критическую секцию, вынужден постоянно опрашивать определённые переменные в разделяемой памяти, при этом не выполняя никаких полезных действий, но занимая время центрального процессора. Чтобы процесс, ожидающий входа в критическую секцию, не расходовал попусту процессорное время, следует, очевидно, заблокировать его до тех пор, пока нужные ему разделяемые данные не окажутся свободны, то есть не выделять ему квантов времени, пока не будет известно, что все остальные процессы критическую секцию покинули. С другой стороны, в нужный момент процесс нужно «разбудить», то есть перевести из состояния блокировки в состояние готовности; желательно при этом сразу пометить разделяемые данные как занятые, чтобы процессу не пришлось снова выдерживать конкурентный поединок с другими процессами за вход в критическую секцию. Всего этого можно добиться, если ввести специальные объекты, среди которых особенно примечательны *мьютексы* и *семафоры Дейкстры*.

Под ***мьютексом***[4] в общем случае понимается объект, имеющий два состояния (открыт/заперт) и, соответственно, две операции: `lock` (запереть) и `unlock` (открыть). Операция `unlock` проходит успешно (и немедленно возвращает управление) в любом случае, переводя объект в состояние «открыт». Операция `lock` в её исходном варианте, если применить её к открытому мьютексу, закрывает его и возвращает управление; при применении её к закрытому мьютексу она блокирует вызвавший процесс до тех пор, пока мьютекс не окажется открыт, после чего закрывает его и возвращает управление. Операция `lock` также может быть реализована как булевская функция; при применении её к открытому мьютексу она закрывает его и возвращает значение «истина» (успех), при применении к закрытому мьютексу функция возвращает значение «ложь» (неудача). Такой вариант реализации называется *неблокирующим*. Обычно реализация мьютекса предоставляет оба варианта операции `lock`.

Важнейшее свойство операций `lock` и `unlock` — их *атомарность*. Это означает, что обе операции выполняются как единое целое и не могут быть прерваны[5]. В частности, операция `lock` не может быть

[4] Англ. *mutex* — сокращение от слов *mutual exclusion*.

[5] Кроме случая, когда операция `lock` блокирует вызвавший процесс — тогда на время блокировки прерывание операции разрешается.

прервана между проверкой текущего значения мьютекса и изменением этого значения на «закрыто». Вопрос о том, как такой атомарности достичь, мы рассмотрим чуть позже.

Вспомним теперь нашу попытку выполнить взаимоисключение с помощью блокировочной переменной (см. стр. 256). Заменим обычную переменную на мьютекс, обозначив его, как и блокировочную переменную, буквой `s`; вместо присваиваний `s = 0` и `s = 1` применим соответственно операции `lock` и `unlock`. Рассмотрим для начала неблокирующую реализацию `lock`:

```
while(!lock(s)) {} /* ждем, пока не получим мьютекс */
section();            /* ... критическая секция ... */
unlock(s);     /* разрешаем другим процессам доступ */
```

В отличие от варианта с использованием блокирующей переменной, данный вариант является корректным. В самом деле, операция `lock` атомарна, так что выход из цикла (по истинному значению функции `lock`) означает, что в некий момент мьютекс оказался открыт, то есть никто в это время не работал с разделяемыми данными, и нашему процессу удалось его закрыть, причём никто другой вклиниться между проверкой и закрытием не мог. Если применить блокирующий тип реализации операции `lock`, наш код станет ещё проще и из него исчезнет цикл активного ожидания:

```
lock(s);                      /* получаем мьютекс */
section();        /* ... критическая секция ... */
unlock(s); /* разрешаем другим процессам доступ */
```

7.1.5. О реализации мьютексов

Заблокировать процесс может только операционная система. Если бы процессу было точно известно, через какой промежуток времени нужный ему ресурс окажется освобождён, он мог бы выполнить системный вызов, подобный функции `sleep`, чтобы отказаться от выполнения на заданный период. Однако момент освобождения нужного ресурса процессу не известен, поскольку зависит от функционирования других процессов. Здесь вполне логично возникает идея возложить управление пометками занятости/освобождения ресурсов на операционную систему, создав ещё один способ взаимодействия процессов.

Следует отметить, что такой подход, кроме избавления от активного ожидания, имеет и другое важное преимущество. Операционная система, в отличие от процесса, имеет возможность при необходимости запрещать прерывания на время исполнения определённых действий внутри ядра, обеспечивая, таким образом, атомарность сколь угодно сложных

операций[6]. При этом исчезает необходимость в сложных ухищрениях, подобных алгоритму Петерсона.

В то же время такой подход имеет недостаток, почти всегда фатальный: системный вызов, как мы знаем, — довольно дорогостоящее действие. К примеру, если в роли разделяемых данных выступает простая целочисленная переменная, пусть даже находящаяся в разделяемой памяти, а для обращения к ней нам приходится организовывать критическую секцию, обращаясь на входе и выходе из неё к операционной системе, то вся затея начисто лишается смысла: проще будет вообще отказаться от разделяемой переменной, а для взаимодействия процессов использовать какой-нибудь канал или сокет.

К счастью, если процессор поддерживает команду `XCHG` (см. т. 2, §3.2.15) или подобную, мьютекс можно реализовать так, чтобы в большинстве случаев обращение к ядру не требовалось. Напомним, что команда `XCHG` имеет два операнда, один из которых регистровый, а второй может быть либо регистровым, либо операндом типа «память»; нам потребуется как раз такой вариант. Команда за одно неделимое (атомарное) действие меняет местами содержимое своих операндов, и эта неделимость позволяет реализовать мьютекс; посмотрим, как это делается.

Несмотря на то, что команда `XCHG` может работать с операндами любой длины — байтами, словами и двойными словами, — при работе в 32-битном режиме эффективнее всего будет использовать в качестве мьютекса четырёхбайтное целое. Поэтому в роли мьютекса мы будем использовать четырёхбайтную переменную, которую пометим меткой `mutex`; договоримся, что значение `1` в мьютексе соответствует состоянию «открыт», а значение `0` — состоянию «закрыт» (можно и наоборот, конкретные значения не так важны). Операция *открытия* мьютекса реализуется простым занесением нуля в переменную, с этим проблем нет. Если же нужно *закрыть* мьютекс, мы сначала заносим в регистр (например, `EAX`) значение `1`, затем выполняем команду `XCHG` между этим регистром и мьютексом и смотрим, какое значение в итоге оказалось в регистре. Если там по-прежнему `1`, то, стало быть, мьютекс уже был закрыт и наша команда ничего не изменила, он так и остался закрытым, а наша операция закрытия должна быть признана неуспешной и нам надо тем или иным способом подождать, не заходя в критическую секцию. Если же после `XCHG` в регистре оказался ноль — значит, мьютекс был открыт, а теперь в результате нашей команды он закрыт.

В самом примитивном варианте операции `lock` и `unlock` можно представить в виде подпрограмм без параметров, а «блокировку» вызывающего процесса реализовать через активное ожидание:

[6]Конечно, сложность атомарных операций должна оставаться в разумных пределах, ведь длительный запрет прерываний может отрицательно сказаться на функционировании всей вычислительной системы.

```
section .data

mutex   dd 1              ; изначально мьютекс открыт

section .text

lock:   mov eax, 1
.again: xchg eax, [mutex]
        cmp eax, 0
        jne .again
        ret

unlock: mov dword [mutex], 0
        ret
```

Такой вариант будет работать, но, конечно, активное ожидание в большинстве практических случаев неприемлемо. Если представить себе, что у нас есть процедура переключения управления на другой тред того же процесса, можно будет перед переходом на начало процедуры `lock` вставить обращение к процедуре переключения тредов, чтобы уменьшить потери от активного ожидания:

```
lock:   mov eax, 1
.again: xchg eax, [mutex]
        cmp eax, 0
        je .ok
        call switch_threads
        jmp .again
.ok     ret
```

В практических реализациях всё происходит существенно сложнее. Треды, не получившие нужный мьютекс, помечаются как заблокированные, при этом для каждого мьютекса поддерживается список тредов, ожидающих его открытия, и когда какой-то из тредов открывает этот мьютекс, один из ожидающих этого события тредов разблокируется. Надо сказать, что никакой «процедуры переключения тредов» у нас обычно нет: в большинстве случаев тред представляет собой единицу планирования с точки зрения операционной системы, так что передать управление от одного треда другому можно только через планировщик, то есть для этого придётся обратиться к ядру. Но если к ядру всё равно придётся обращаться, проще будет попросить ядро не о переключении треда, а о блокировке данного конкретного треда до тех пор, пока нужный мьютекс не окажется доступен; с другой стороны, тред, «отпускающий» мьютекс, может проверить, заблокирован ли кто-нибудь из других тредов на этом мьютексе, и если такие есть (и только в этом случае) — попросить ядро кого-нибудь из них разблокировать.

Операционные системы обычно предоставляют специальные системные вызовы для таких случаев; так, в ОС Linux поддерживается вызов `futex`. Описание этого вызова само по себе достаточно сложно, а программирование с его использованием представляет собой сочетание довольно нетривиальных трюков. Библиотека Си даже не предоставляет обёртки для этого вызова; соответствующая страница `man` содержит замечание о том, что этот вызов не предназначен для обычных пользователей (программистов). Технические подробности организации работы с вызовом `futex` любопытствующий читатель может найти, например, в статье [13]; не стоит огорчаться, если текст статьи покажется вам слишком сложным: даже профессиональные программисты зачастую не могут понять, что там к чему.

Следует обратить внимание, что **подавляющее большинство операций блокировки и разблокировки мьютексов не требует обращения к ядру**. Если говорить точнее, то системный вызов потребуется лишь в случаях, когда операция захвата мьютекса прошла неудачно: сначала тому треду или процессу, который захватывал мьютекс, придётся обратиться к ядру с просьбой его усыпить, а потом тред или процесс, который будет мьютекс освобождать, будет вынужден тоже обратиться к ядру, чтобы ядро разбудило кого-нибудь из тех, кто ранее на этом мьютексе заблокировался. Грамотно написанные параллельные программы в такую ситуацию попадают относительно редко, поскольку — если, подчеркнём, делать всё правильно — длительность критических секций невелика в сравнении с общим временем работы программы; ну а до тех пор, пока все операции захвата мьютексов проходят успешно, обращения к ядру не потребуется ни при захвате мьютексов, ни при их освобождении (поскольку на них никто не блокирован).

7.1.6. Семафоры Дейкстры

Семафор Дейкстры в классическом варианте представляет собой целочисленную переменную, про которую известно, что она принимает только неотрицательные значения и над которой определены две операции: `up` и `down`. Операция `up` всегда проходит успешно, увеличивая значение переменной на 1, и немедленно возвращает управление. Операция `down` должна, наоборот, уменьшать значение на 1, но сделать это она может только когда текущее значение строго больше нуля, ведь значение семафора не может быть отрицательным. При положительном значении семафора операция `down` уменьшает его значение на 1 и немедленно возвращает управление. В случае же нулевого значения семафора операция блокирует вызвавший процесс до тех пор, пока значение не станет положительным, после чего уменьшает значение и возвращает управление. Как и в случае с мьютексом, операции над семафором обязательно должны быть реализованы *атомарно*: необходимо полно-

стью исключить ситуации, когда результат нескольких независимых операций над семафором может зависеть от времени вызова операций независимыми процессами.

Ясно, что с помощью семафора можно имитировать мьютекс, если считать значение 0 состоянием «закрыт», значение 1 — состоянием «открыт», а операции `lock` и `unlock` заменить на `down` и `up`. Нужно только следить, чтобы операция `up` никогда не применялась к положительному семафору, а этого можно достичь, если применять её только в начале программы (один раз), а также после того, как прошла операция `down`, и никогда иначе. Однако семафор может выступать в более сложной роли, а именно, как *счётчик доступных ресурсов*, которых может быть больше одного; именно так его обычно и используют. Чтобы понять, о чём идёт речь, представьте себе, что в критической секции не просто осуществляется некий доступ к данным, а производится *распределение* между работающими процессами чего-то такого, чего не хватает на всех. В этом случае процессу нет смысла заходить в критическую секцию, когда дефицитный ресурс исчерпан: нужно подождать, пока он снова появится. Пример такой ситуации мы рассмотрим в §7.2.1.

Некоторые реализации обобщают операции над семафорами, вводя дополнительный целочисленный параметр. Операция `up(sem, n)` увеличивает значение семафора на `n`, операция `down(sem, n)` ждёт, пока значение не окажется бóльшим либо равным `n`, и после этого уменьшает значение на `n`. Кроме того, реализации обычно имеют неблокирующий вариант операции `down`, аналогичный нашей операции `lock` для мьютексов, представленный в виде логической функции; этот вариант вместо ожидания сразу возвращает управление, указывая, что операция прошла неудачно. Большинство существующих реализаций позволяет узнать текущее значение семафора или даже установить его без всяких блокировок и проверок, что бывает удобно при инициализации программы.

Известная реализация семафоров из System V IPC (см. [2]) предоставляет *массивы* семафоров, над которыми можно производить сложные атомарные операции, например, увеличить третий семафор на два, а четвертый уменьшить на три за одно (атомарное!) действие. Кроме того, в этой реализации есть дополнительная операция «блокироваться, пока семафор не станет равен нулю».

Что касается классического семафора без дополнительных операций, то реализовать его можно теми средствами, которые у нас уже есть. В самом деле, возьмём обыкновенный канал (см. §5.5.3) — именованный или неименованный, неважно. Операцию `up` реализуем в виде записи в канал одного байта с помощью `write` (содержание байта нас в данном случае не волнует), а операцию `down` — в виде чтения байта из канала с помощью `read`. Когда во внутреннем буфере канала нет ни одного байта, вызов `read` заблокируется в ожидании поступления данных, то есть до тех пор, пока кто-нибудь не выполнит запись в канал (наш аналог операции `up`). Нетрудно видеть, что такое взаимодействие полностью соответствует классическому определению семафора. Единственный нюанс здесь в том, что буфер канала отнюдь не бесконечен, так что у семафора, реализованного таким способом, будет довольно скромная верхняя граница принимаемых значений (для современных версий Linux это будет 2^{16}); но аналогичное огра-

ничение есть у всех реально существующих семафоров, и хотя обычно граница пролегает выше — чаще всего это $2^{32} - 1$ — такое различие скорее количественное, нежели качественное, и в любом случае для большинства задач это неважно, поскольку на практике значения семафоров редко превышают несколько десятков. Проблема с такой реализацией, как и с реализацией из System V IPC, в том, что для любой операции над семафором требуется системный вызов.

7.2. Классические задачи взаимоисключения

В этой главе мы рассмотрим несколько классических примеров программирования с использованием мьютексов и семафоров. Примеры мы постараемся сделать максимально абстрактными, не уточняя, настоящие процессы имеются в виду или треды, какая конкретно используется реализация семафоров и мьютексов и т. п. Приводимые примеры будут даже написаны не на Си, а скорее на си-подобном псевдокоде: обозначив мьютекс буквой `m`, мы будем записывать операции над ним как `lock(m)` и `unlock(m)`, а операции над семафором `s` обозначим как `up(s)` и `down(s)`, не обращая внимания на то, что передача параметров в Си происходит только по значению[7]. Примеры, корректные с технической точки зрения, мы приведём в следующей главе.

7.2.1. Задача производителей и потребителей

Пусть имеется несколько процессов, *производящих* данные, и эти данные нуждаются в дальнейшей обработке. Это могут быть, например, процессы, опрашивающие какие-то датчики; также это могут быть процессы, получающие некую информацию по сети с других машин и преобразующие её во внутреннее представление; возможно, что данные получаются в результате интерактивного взаимодействия с пользователем или, наоборот, вычисляются в ходе математических расчётов. Назовём эти процессы *производителями*.

С другой стороны, есть несколько процессов, обрабатывающих (*потребляющих*) данные, подготовленные процессами-производителями. Эти процессы мы назовём потребителями.

Задача производителей и потребителей, предназначенная для иллюстрации применения семафоров, рассматривает следующий вариант передачи информации от производителей потребителям. Пусть

[7]Заметим, в Си++ имеется передача параметров по ссылке, так что наиболее педантичным читателям мы можем предложить считать наш псевдокод написанным на Си++; кроме того, вы можете считать, что переменные `m` и `s` — это не сами семафоры и мьютексы, а указатели на них или какие-нибудь дескрипторы наподобие файловых. Подчеркнём, что для иллюстративных целей, которые мы перед собой ставим в этой главе, всё это неважно.

каждая порция информации, создаваемая производителем и обрабатываемая потребителем, имеет фиксированный размер. В разделяемой памяти организуем буфер, способный хранить N таких порций информации. Ясно, что работа с буфером требует взаимоисключения. Для упрощения работы будем считать, что буфер представляет собой единое целое[8], и на время работы любого процесса с буфером исключать обращения к буферу других процессов.

Легко видеть, что проблема взаимоисключения в данном случае оказывается не единственной. При взаимодействии потребителей и производителей возможны две (симметричные) ситуации, требующие блокировки:

- потребитель готов к получению порции данных, но в буфере ни одной порции данных нет (буфер пуст);
- производитель подготовил к записи в буфер порцию данных, но записывать ее некуда (все слоты буфера заняты).

В обоих случаях процессу нужно дождаться, когда другой процесс изменит состояние буфера: в первом случае потребителю — когда производитель запишет новые данные; во втором случае, наоборот, производителю — когда потребитель заберёт какие-то уже имеющиеся данные.

Простейшая возможная стратегия — блокировать операции над буфером (войти в критическую секцию), проверить, не изменилось ли соответствующим образом состояние буфера, и если нет, выйти из критической секции, ничего в буфере не поменяв, а затем опять войти, и т. д., то есть просто выполнять циклически проверку изменений. Иначе говоря, такая стратегия представляет собой активное ожидание, только не на входе в критическую секцию, а в процессе работы. Мы уже знаем, что активное ожидание — идея крайне неудачная, и описываемая ситуация — не исключение: в самом деле, *через какое время процессу следует снова заходить в критическую секцию?* Через секунду? Через десятую долю секунды? Или, наоборот, через десять секунд? Никаких сведений для вычисления этого интервала у нас нет, и если мы выберем его слишком большим, это повлечёт задержки в работе взаимодействующих процессов: производители могут продолжать ждать, когда в буфере уже давно появились свободные слоты, потребители могут «висеть», когда в буфере уже лежит для них информация (может быть, даже весь буфер заполнен). Если же временной интервал выбрать слишком коротким, процессы будут в основном заняты «пиханием локтями» на входе в критическую секцию, заходя туда и выходя обратно, возможно, по много тысяч раз, прежде чем удастся сделать что-то полезное; при этом они будут не только расходовать процессорное время, но и мешать друг другу.

[8] Это так и есть, если в буфере, например, имеются глобальные данные о том, какие из слотов свободны, а какие заняты.

```
void producer()
{
  /* ... подготовить
         данные ...
   */
  down(empty);
  lock(m);
  put_data();
  unlock(m);
  up(full);
}
```

```
void consumer()
{
  down(full);
  lock(m);
  get_data();
  unlock(m);
  up(empty);
  /* ... обработать
         данные ...
   */
}
```

Рис. 7.5. Производители и потребители

Было бы, по-видимому, правильно придумать такой механизм, при котором процессы в такой ситуации вообще не будут заходить в критическую секцию, пока в буфере не появятся данные (для потребителя) либо свободные слоты (для производителя); это и будет решением задачи о производителях и потребителях. Такой механизм можно создать на основе семафоров Дейкстры в качестве счётчиков доступных ресурсов. Задействуем два семафора: один будет считать свободные слоты, и на нём будут блокироваться процессы-производители, если свободных слотов нет; второй будет считать слоты, заполненные данными, и на нём будут блокироваться процессы-потребители, если нет готовых данных. Назовём эти семафоры соответственно `empty` и `full`; в начале работы одновременно с созданием буфера выполним операцию `up(empty)` столько раз, сколько в буфере имеется свободных слотов. Наличие этих семафоров не отменяет необходимости взаимного исключения операций с разделяемым буфером. Для этой цели мы введём мьютекс, который назовём `m`.

Будем считать, что для помещения данных в буфер и извлечения данных из буфера у нас есть процедуры `put_data` и `get_data`, которые не делают никаких операций с семафорами и мьютексами: они написаны в предположении, что процесс уже находится в критической секции и что наличие в буфере нужного ресурса (свободного или занятого слота) кто-то уже гарантировал, и выполняют только рутинную работу вроде «найти нужный слот, скопировать информацию в него или из него, пометить его как занятый или свободный».

Итоговое решение показано на рис. 7.5. Подчеркнём ещё раз, что семафоры в данном случае используются не для взаимоисключения критических секций, а для блокирования процессов, которые всё равно не могут (прямо сейчас) сделать ничего полезного. Взаимоисключение обеспечивает только мьютекс.

7.2.2. Задача о пяти философах и проблема тупиков

При совместной работе нескольких процессов с несколькими разделяемыми объектами возможна ситуация, при которой два или больше участников взаимодействия оказываются в состоянии блокировки, из которого каждый мог бы выйти, если бы другой освободил какой-то из объектов, но этот другой освободить объект не может, поскольку сам тоже находится в блокированном состоянии. Это называется ***тупиком***[9].

Для иллюстрации проблемы тупиков Эдсгер Дейкстра предложил шуточную ***задачу о пяти обедающих философах***. За круглым обеденным столом сидят пять философов, размышляющих о высоких философских материях. В середине стола стоит большая тарелка спагетти. Между каждыми двумя философами на столе располагается вилка, т. е. вилок тоже пять, причём каждый философ может взять вилку слева и вилку справа, если, конечно, вилкой в данный момент не пользуется сосед. Поскольку спагетти отличаются изрядной длиной, для еды каждому философу нужно две вилки[10]. Поэтому каждый философ, поразмыслив некоторое время о непреходящих категориях и решив подкрепиться, пытается сначала взять в левую руку вилку, находящуюся от него слева; если вилка занята, философ с поистине философским спокойствием ожидает, пока вилка не освободится. Завладев левой вилкой, философ точно так же пытается правой рукой взять вилку справа, не выпуская при этом первую вилку из левой руки. Философы, как известно, отличаются философским отношением к житейским трудностям, так что каждый философ готов ждать появления нужной ему вилки хоть до скончания веков — точнее, до наступления голодной смерти, ибо бренные тела, в отличие от просветлённых умов, требуют иногда пищи отнюдь не духовной.

Завладев двумя вилками, философ некоторое время утоляет голод, затем кладёт вилки обратно на стол и продолжает размышлять о вечных вопросах, пока снова не проголодается. Проблема заключается в том, что все пять философов могут проголодаться одновременно (с точностью

[9]Полезно помнить, что соответствующий английский термин — *deadlock*, буквально означающий что-то вроде «заклинивания намертво»; точного перевода на русский для этого слова нет. «Тупик» в некомпьютерном смысле по-английски будет *dead end* или вовсе *cul-de-sac*; если попытаться обозначить этими словами ситуацию с заблокировавшими друг друга процессами, вас, скорее всего, не поймут.

[10]Студенты по разные стороны океана с завидной регулярностью задают вопрос, как же (технически) использовать две вилки для поедания спагетти. Можете как-нибудь на досуге попробовать: одной вилкой накалываете несколько спагеттин, а второй вилкой наматываете их на первую и в таком виде отправляете в рот. Впрочем, позже был предложен другой вариант условий задачи: за столом сидят *восточные* философы, перед ними блюдо с рисом, а на столе лежат не вилки, а палочки для еды. Всем известно, что этих палочек нужно две; правда, держат их всё же одной рукой. Автор этих строк как-то раз наткнулся на вариант задачи, в котором философы едят креветок, а вторая вилка нужна, судя по всему, чтобы этих креветок разделывать.

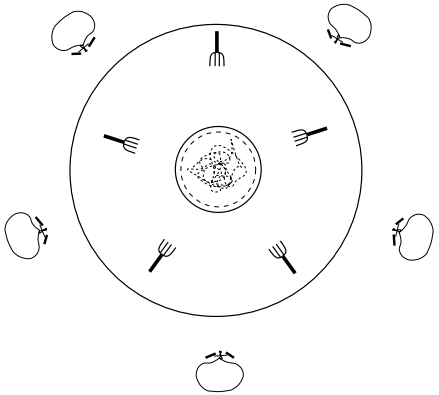

Рис. 7.6. Обедающие философы

до времени, затрачиваемого на процедуру завладения вилкой). В этом случае все философы успеют взять вилки в левые руки, да так и замрут с этими вилками в ожидании, когда же появится вилка справа. Однако справа вилка может появиться только тогда, когда правый сосед утолит голод, а этого не происходит, ведь у него тоже только одна вилка. В результате наши достойнейшие мудрые мужи безвременно покинут сей мир, так и не дождавшись вторых вилок. Экая неприятность!

Именно такие ситуации и называются ***тупиками***. Для возникновения тупика, вообще говоря, достаточно двух процессов и двух ресурсов. Пусть имеются два мьютекса `m1` и `m2`. Если первый процесс выполняет код, содержащий вызовы

```
lock(m1);
lock(m2);
```

а второй в это же время выполняет код, содержащий те же вызовы в обратном порядке:

```
lock(m2);
lock(m1);
```

то при неудачном стечении обстоятельств оба процесса успеют сделать по одному вызову и войдут во взаимную блокировку на вторых, попав, таким образом, в тупик.

Более того, ситуации взаимоблокировки возможны не только с участием семафоров и мьютексов. Рассмотрим для примера одну такую ситуацию. Пусть нам понадобилось запустить команду `ls` для получения в текстовом виде списка файлов в текущем каталоге. Начинающие программисты часто делают характерную ошибку, применяя примерно такой код:

```
char buf[100];
int rc;
int fd[2];
pipe(fd);
if(fork()==0) {
    dup2(fd[1], 1);
    close(fd[1]);
    close(fd[0]);
    execlp("ls", "ls", NULL);
    perror("ls");
    exit(1);
}
close(fd[1]);
wait(NULL);         /* !!! так делать не стоило */
while((rc = read(fd[0], buf, sizeof(buf)))>0) {
    /* ... */
}
```

Любопытно, что такая программа, вообще говоря, может и заработать, однако может и «зависнуть». Экспериментируя с ней, мы обнаружим, что программа корректно работает в каталогах со сравнительно небольшим количеством файлов, а на больших каталогах «зависает». Прежде чем читать дальше, рекомендуем читателю попытаться самостоятельно догадаться о причинах.

Итак, рассмотрим программу подробнее. Запускаемая нами в порождённом процессе программа `ls` в качестве дескриптора стандартного вывода получает входной дескриптор канала, и прежде чем завершиться, она будет записывать в канал имена файлов из текущего каталога. Между тем родительский процесс, движимый благородной целью недопущения засорения системной таблицы зомби-процессами, выполняет вызов `wait`, в результате чего блокируется до тех пор, пока порождённый процесс не завершится. Лишь после этого родительский процесс выполняет чтение из канала.

В результате получается, что во время работы порождённого процесса (программы `ls`) никто из канала не читает. Как нам известно из §5.5.3, размер буфера канала ограничен[11], и когда буфер заполнится, очередной вызов `write`, выполненный программой `ls`, заблокируется в ожидании освобождения места в буфере. Однако буфер освобождать некому, поскольку родительский процесс, блокированный на вызове

[11] До недавних пор размер этого буфера в ОС Linux составлял 4096 байт, но в современных ядрах под буфер канала по умолчанию выделяется 64 Кб; для большинства директорий этого достаточно, что несколько затрудняет возможность увидеть вышеприведённую программу в состоянии тупика. Впрочем, в других версиях ОС Unix буфер может иметь любой размер, возможна даже реализация вовсе без буфера. Но загнать нашу программу в состояние тупика можно и на современных системах: попробуйте добавить команде `ls` ключик `-l` и запустить программу, например, в директории `/usr/bin`.

`wait`, до первого вызова `read` не дошёл и не дойдет, пока порождённый процесс не завершится.

Как видим, здесь возникает замкнутый круг: родительский процесс ожидает, что потомок завершится, и не выполняет чтение из канала, а потомку, чтобы завершиться, нужно, в свою очередь, чтобы родительский начал читать. Это и есть, собственно говоря, ***тупик***. Как уже, несомненно, догадался читатель, в данном случае взаимоблокировка возникнет только тогда, когда выдача `ls` для данного каталога превысит размер буфера. Ясно, что приведённое решение очень просто превратить в правильное: достаточно перенести вызов `wait` на несколько строк ниже, чтобы он выполнялся уже *после* выполнения вызовов `read`.

Вернёмся к задаче о пяти философах и попробуем её решить. Для начала рассмотрим вариант, никак не защищённый от вышеописанной тупиковой ситуации. Заведём массив из пяти мьютексов, каждый из которых связан с соответствующей вилкой; обозначим этот массив идентификатором `forks`[12]. И философов, и вилки занумеруем числами от 0 до 4. Опишем две вспомогательные функции, позволяющие вычислить номер соседа справа и слева:

```
int left(int n) { return (n - 1 + 5) % 5; }
int right(int n) { return (n + 1) % 5; }
```

Будем считать, что номер вилки, лежащей слева от философа, совпадает с номером самого философа. Жизненный цикл философа тогда можно будет представить следующей процедурой:

```
void philosopher(int n)
{
    for(;;) {
        think();
        lock(forks[n]);         /* ! */
        lock(forks[right(n)]);
        eat();
        unlock(forks[n]);
        unlock(forks[right(n)]);
    }
}
```

Ясно, что при одновременном выполнении таких процедур для `n` от `0` до `4` возможна ситуация, когда все они успеют выполнить блокировку, помеченную в листинге восклицательным знаком. При этом все пять «вилок» (мьютексов) окажутся блокированы, так что все процессы также

[12] Английское слово *fork* буквально переводится как «вилка» (которой едят) или «развилка» (на дороге). Системный вызов `fork` назван с использованием второго значения, а мы здесь используем первое.

заблокируются на следующей строке процедуры при попытке получить вторую вилку.

Избежать тупика можно, например, применив семафор, не позволяющий философам приступать к трапезе всем одновременно. Заведём семафор и назовём его `sem`. Тогда жизненный цикл философа примет следующий вид:

```
void philosopher(int n)
{
    for(;;) {
        think();
        down(sem);
        lock(forks[n]);
        lock(forks[right(n)]);
        eat();
        unlock(forks[n]);
        unlock(forks[right(n)]);
        up(sem);
    }
}
```

Тупик теперь невозможен, но полученное решение трудно назвать удачным. В самом деле, какое значение присвоить семафору перед началом работы? Если присвоить ему значение 1, употреблять спагетти в любой момент сможет лишь один философ, остальным придётся ждать. Мы получим нерациональный простой ресурсов, ведь условия позволяют есть двум философам одновременно, не мешая друг другу. Начальное значение, равное двум, не спасёт ситуацию, ведь «по закону подлости» за семафор обязательно пройдут философы, сидящие рядом, так что пока один философ будет кушать, никто другой подкрепиться не сможет: соседу, прошедшему за семафор, не достанется вилки, а остальные за семафор не пройдут.

Очевидно, максимальное возможное значение семафора — четыре, в противном случае теряется его смысл. При таком значении возможна ситуация, когда три философа успели взять по одной вилке и лишь один взял две. Пока он не поест, остальные будут ждать. «Правильного» начального значения здесь просто нет, все возможные значения имеют те или иные недостатки. Впрочем, как мы сейчас увидим, для задачи возможны гораздо более простые решения, хотя с каждым из них связаны определённые проблемы.

Довольно красивое решение — посадить за стол одного левшу и предложить четверым философам по-прежнему сначала брать левую вилку, а затем правую, тогда как левше предложить сначала брать вилку справа, а затем слева. Тупика в этом случае не получится. Официальное название этого метода — «иерархия ресурсов», и это решение

предложил сам автор задачи про философов — Дейкстра. В простейшем изложении этого метода предлагается занумеровать разделяемые объекты и оформить все процессы так, чтобы они осуществляли захват объектов в порядке возрастания их номеров. Если использовать нашу нумерацию вилок, то в роли «левши» окажется философ, сидящий между вилками №0 и №4: вилка №0 лежит справа от него, но поскольку 0 меньше, чем 4, ему в соответствии с предложенным правилом придётся брать её первой.

В общем случае для разделяемых объектов достаточно установить так называемое *отношение частичного порядка* при условии, что *несоразмеримые* (в соответствии с этим отношением) объекты никогда не используются одним процессом. Как ни странно, иногда это оказывается проще, чем перенумеровывать объекты, в особенности когда множество доступных объектов динамически меняется в ходе работы системы.

Решение на основе иерархии ресурсов позволяет избежать тупика, но при внимательном рассмотрении оказывается не слишком удачным с точки зрения эффективности. В применении к философам легко заметить, что возможна ситуация, когда философ №3 кушает, а все остальные его ждут: философы №№0, 1 и 2 при этом держат вилки в левых руках, философ №4 ничего не держит — он ждёт освобождения вилки №0. Что касается задач из реального мира, процессам часто требуется не два разделяемых объекта, а больше, причём узнать, какой объект понадобится следующим, процесс может уже после захвата других объектов; такие ситуации — обычное дело при работе с базами данных. Если номер очередного нужного объекта окажется меньше, чем номера уже захваченных, процессу придётся сначала эти объекты освободить, захватить объект с меньшим номером, а затем снова попытаться захватить только что освобождённые объекты; вся эта канитель ведёт к существенным потерям в быстродействии.

Ещё одно достаточно изящное решение задачи о философах состоит в том, чтобы только одному из них разрешать брать вилки. Описывая это решение, обычно говорят о некоем «слуге» или «официанте», который в каждый момент времени может стоять за спиной одного из философов. Ощутив желание подкрепиться, философ поднимает руку и ждёт, когда слуга окажется за его спиной, после чего действует стандартным способом: берёт сначала одну вилку, затем вторую. Слуга, увидев чью-то поднятую руку, подходит к этому философу и не отходит от него до тех пор, пока философ не возьмёт две вилки; лишь когда обе вилки окажутся в руках философа, слуга отходит от него и смотрит, не поднимет ли руку кто-то ещё. Кушать спагетти и класть вилки обратно на стол философы могут без оглядки на слугу. В применении к процессам слуга превращается в простой мьютекс. Если назвать этот мьютекс `steward`, жизненный цикл философа примет следующий вид:

```
void philosopher(int n)
{
    for(;;) {
        think();
        lock(steward);
        lock(forks[n]);
        lock(forks[right(n)]);
        unlock(steward);
        eat();
        unlock(forks[n]);
        unlock(forks[right(n)]);
    }
}
```

К сожалению, это решение тоже не вполне годится: если один из философов кушает, а любой из его соседей проголодался, позвал слугу и, дождавшись его, взял одну из вилок, никто больше не сможет кушать, пока первый философ не насытится — ведь до этого момента его сосед так и будет сидеть с одной вилкой, а слуга — стоять за его спиной.

Задача о философах позволяет также проиллюстрировать ***проблему ресурсного голодания*** (*resource starvation problem*), когда процесс бесконечно повторяет попытки захвата нужных ему ресурсов, но из-за неправильного построения системы взаимоисключений или планирования никогда их не получает. Допустим, философы, не теряя философского отношения к жизни, обрели, тем не менее, толику здравого смысла и решили, что каждому, кто просидит минуту с одной вилкой, следует положить её обратно на стол, подождать десять минут и тогда уже повторить попытку. Такое «простое и очевидное» решение приводит вместо тупика к ресурсному голоданию: если все философы одновременно возьмут по вилке, то через минуту они — опять же одновременно — положат вилки обратно, дружно подождут десять минут и снова — увы, одновременно — возьмут левые вилки. В английском языке для такой ситуации имеется термин *livelock* (в противоположность *deadlock*'у). Проблемы подобного рода чаще всего решаются введением в систему датчика случайных чисел, определяющего, через какой интервал времени следует повторить попытку; в применении к философам можно предложить каждому из них, положив вилку после минуты безуспешного ожидания, бросить игральную кость и повторить попытку взятия вилок через столько минут, сколько очков выпадет на кости. Надо сказать, что современные реализации мьютексов обычно позволяют ограничить время блокировки вызвавшего процесса неким заданным тайм-аутом, но использовать эту возможность следует аккуратно, чтобы не устроить ситуацию голодания.

В книге [5] Э. Танненбаум приводит решение[13], в котором философы, подкрепляясь, при этом не мешают никому, кроме своих соседей. В этом решении каждому философу соответствует переменная, хранящая его *состояние*: `hungry`, `thinking` или `eating`; массив этих переменных назовём `state`. Кроме того, каждому философу соответствует мьютекс, на котором он блокируется до того момента, когда ему будет можно приступить к трапезе, чтобы при этом никому не мешать. Таковым считается момент, когда ни один из его соседей (ни слева, ни справа) не приступил к еде и не принял решение приступать к еде. Если в тот момент, когда философ проголодался, оба соседа размышляли, философ сам себе взводит свой мьютекс, позволяя самому себе начать трапезу, то есть выполняет операцию `unlock`; если же один из соседей в этот момент ел, философ мьютекс не взводит. Несколькими шагами позже философ пытается «захватить» собственный мьютекс, что удаётся ему, только если перед этим он его взвёл. В противном случае философ будет ждать в режиме блокировки на мьютексе до тех пор, пока сосед, утолив голод, не предложит ему подкрепиться. При этом философ приступит к трапезе только в том случае, если второй его сосед также в настоящий момент не ест; в противном случае он продолжит ждать, уповая на то, что уже второй сосед, насытившись, напомнит нашему мудрецу, что пришло время утолить голод.

Отметим, что в этом решении мьютексы, связанные с вилками, оказываются не нужны: алгоритм и так гарантирует, что два философа не попытаются схватить одну вилку одновременно. Зато нам потребуется один общий мьютекс для защиты массива `state`. Соответствующий код приведён ниже. Центральное место в нём занимает функция `test`. С её помощью каждый философ, проголодавшись, определяет, следует ли ему прямо сейчас приступать к трапезе. Утолив голод, философ вызывает функцию `test` для соседей (это и есть наше любезное предложение подкрепиться), в результате чего, если соответствующий сосед голоден, а соседи соседа в этот момент не едят, происходит взведение мьютекса и философ, находившийся в состоянии блокировки на нём, приступает к трапезе.

```
enum possible_states { hungry, eating, thinking };
int state[5] =
    { thinking, thinking, thinking, thinking, thinking };
mutex mut[5];     /* в начале они заперты */
mutex state_mut;  /* в начале открыт */
void philosopher(int n)
{
    for(;;) {
        think();
        take_forks(n);
```

[13] Наш текст, приведенный ниже, от решения Танненбаума несколько отличается.

```
            eat();
            put_forks(n);
        }
    }
    void take_forks(int i)
    {
        lock(state_mut);
        state[i] = hungry;
        test(i);
        unlock(state_mut);
        lock(mut[i]);
        /* если философ не разрешил сам себе начать трапезу, здесь
           он будет ждать, пока ему о трапезе не напомнят соседи */
    }
    void put_forks(int i)
    {
        lock(state_mut);
        state[i] = thinking;
        /* теперь любезно поинтересуемся,
           не хотят ли наши соседи кушать */
        test(left(i));
        test(right(i));
        unlock(state_mut);
    }
    void test(int i)
    {
        if(state[i] == hungry &&
           state[left(i)] != eating && state[right(i)] != eating)
        {   /* настал черед i-го философа поесть */
            state[i] = eating;
            unlock(mut[i]);
        }
    }
```

Самое интересное здесь — это то, что даже такое сложное и по-своему красивое решение оказывается не совсем правильным. Несмотря на то, что здесь вроде бы нет никаких тайм-аутов, наш алгоритм оказывается подвержен проблеме голодания: если, к примеру, философы с номерами 1 и 3 окажутся изрядными чревоугодниками и будут наслаждаться поеданием спагетти столь долго и столь часто, что вдвоём «закроют» всю ось времени, сидящий между ними философ № 2 будет вынужден положить зубы на полку, и, в отличие от голодания, возникающего при захватах с тайм-аутами, здесь никакой датчик случайных чисел не поможет.

Ещё одно решение, при котором ни один из философов не мешает кушать никому, кроме своих соседей, было предложено американскими

учёными К. М. Чанди и Дж. Мисрой[14]. Это решение, вдобавок к предыдущим, свободно ещё и от проблемы голодания; иной вопрос, что вместо мьютексов для синхронизации используется обмен сообщениями.

Вилки в решении Чанди-Мисры вообще никогда не кладутся на стол: философы держат вилки в руках и передают их соседу (левую — левому, правую — правому) по их просьбе, то есть каждая вилка в каждый момент времени находится в руке одного из двух философов. Вилка может быть *чистой* и *грязной*; вилка становится грязной, когда её используют для еды, но каждый философ, протерев вилку салфеткой, может сделать её чистой, и именно так наши вежливые философы и поступают, прежде чем передать вилку соседу. Изначально все вилки считаются грязными; это не совсем логично в наших декорациях, но для решения задачи нужно считать именно так.

Решив подкрепиться, философ проверяет, есть ли у него обе вилки. Для каждой недостающей вилки он направляет соответствующему соседу просьбу о передаче вилки, после чего, если хотя бы одну просьбу пришлось передать (то есть хотя бы одной вилки у него нет), ждёт, пока ему дадут вилки, и тогда начинает кушать. Если у философа есть обе вилки, он может начать кушать немедленно.

Действия философа при получении от соседа просьбы о предоставлении вилки зависят от того, чистая эта вилка или грязная. Если вилка грязная, философ протирает её, делая чистой, и передаёт соседу; интересно, что философ при этом может быть голоден; в этом случае вилку он всё равно отдаёт, но сопровождает её просьбой вернуть вилку, когда сосед подкрепится. Если же вилка чистая, философ оставляет её при себе, но запоминает, что сосед просил ему эту вилку отдать. Завершая трапезу, философ вспоминает, не просил ли кто-то из соседей вилки, и если просил — протирает соответствующую вилку, делая её чистой, и отдаёт соседу. Теперь мы можем догадаться, почему изначально все вилки считаются грязными: если исходно считать их чистыми, ни один философ не отдаст свою вилку соседу — и, следовательно, ни один не сможет приступить к еде.

Проблема голодания тут снимается как раз через смену статуса вилок (чистые-грязные). Если бы «чистота» вилок не учитывалась, не в меру активные соседи могли бы то и дело выдёргивать вилки из рук отдельно взятого философа, которому не повезло. Конечно, отдавая вилку, голодный философ всегда просил бы её вернуть, и соседи ему бы её даже возвращали — лишь для того, чтобы едва ли не сразу опять потребовать её назад; проблема здесь в том, что такой момент, когда у философа есть обе вилки сразу, может не настать никогда. Но право философа оставить себе вилку, когда она чистая, означает, что философ, *готовый* воспользоваться вилкой (как только получит вторую), но ещё

[14] K. Mani Chandy, Jayadev Misra.

не воспользовавшийся ею, не отдаёт вилку до тех пор, пока *один раз* не поест. Легко видеть, что в такой системе время ожидания до начала трапезы всегда будет конечным.

Завершим обсуждение задачи о философах вопросом о том, почему их именно пять. Многие специалисты полагают, что это число Дейкстра выбрал случайно, из эстетических или каких-то других внетехнических соображений, но это не так; в действительности именно при пяти действующих лицах задача приобретает какой-то смысл, и сейчас мы в этом убедимся.

При наличии у нас всего одного философа задача вырождается: никаких взаимоисключений больше нет, ситуация тупика невозможна, философ кушает, когда ему вздумается (конечно, вилок ему придётся всё же выдать две) и никто ему в этом не мешает. Этот случай нам попросту неинтересен. Если философов двое, то формально устроить тупиковую ситуацию уже возможно, но при более внимательном взгляде на задачу мы обнаружим, что реализация, при которой каждый из философов (процессов) захватывает каждый из ресурсов (вилок) независимо, изначально идиотична: в самом деле, вилок всё равно хватит только на одного, почему бы в таком случае не рассматривать эти вилки как *один* ресурс, а не два разных? Например, философы могли бы договориться класть вилки в футляр; решив подкрепиться, философ, если нужно, ждёт, пока футляр не окажется на столе, берёт его себе, вынимает из него вилки, кушает, после чего кладёт вилки обратно и возвращает футляр на стол. В применении к реальным программистским задачам это означает, что два имеющихся процесса взаимоисключают доступ к разделяемым данным с помощью одного мьютекса; никакого тупика здесь, понятное дело, не получится.

То же самое можно сказать про случай трёх философов: вилок на двоих всё равно не хватит, так что в любой момент времени кушать может только один из них. Если по какой-то причине нужны именно три вилки, философы могут договориться, к примеру, придвигать к себе блюдо со спагетти, и лишь после этого брать вилки, а после трапезы блюдо отодвигать; опять-таки, взаимоисключение здесь обеспечивается одним мьютексом, и никаких тупиков не возникает.

При наличии четырёх философов ситуация усложняется, поскольку кушать могут уже двое; но и здесь можно обнаружить, что задача имеет упрощённое решение. Когда один из философов подкрепляется, кушать одновременно с ним может лишь тот, кто сидит напротив. Иначе говоря, философы могут подкрепляться исключительно фиксированными парами; внутри каждой из двух пар конфликта доступа к вилкам произойти не может. В терминах философов, вилок и блюда можно предложить, например, такое решение: блюдо, стоящее на столе, сделать не круглым, а вытянутым на манер селёдочницы, и поставить

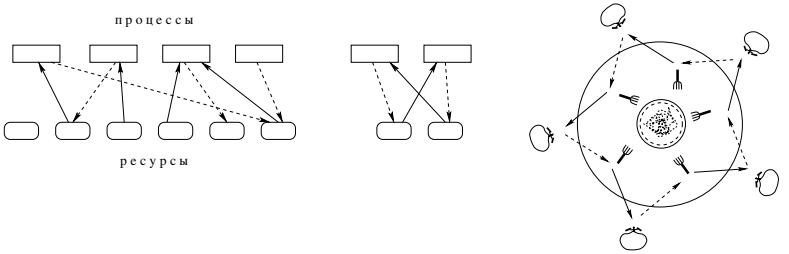

Рис. 7.7. Примеры графа ожидания

его так, чтобы исходно оно своими концами смотрело между сидящих. Решив поесть, философ смотрит на положение блюда; если оно стоит одним концом к нему, философ попросту начинает трапезу, если блюдо стоит в «нейтральном положении», философ его поворачивает одним из концов к себе, и только если блюдо развёрнуто поперёк (что может иметь место лишь в случае, если один или оба философа из противоположной пары сейчас как раз кушают), философ ждёт, пока блюдо не вернётся в нейтральное положение. В реальной программистской задаче можно применить, например, уже известный нам алгоритм чередования; можно также завести переменную, в которой будет храниться в виде четырёх логических флагов список философов, принимающих пищу либо намеренных это делать, и т. д.

Так или иначе, решение для двух пар можно сделать качественно проще, нежели для пяти действующих лиц. Именно число пять оказывается минимальным, при котором задача об обедающих философах не допускает упрощённых решений; при этом для бóльшего числа философов решение концептуально будет точно таким же, как для пяти, отличаясь чисто количественно.

7.2.3. Граф ожидания

Существуют различные подходы к автоматическому отслеживанию наступления тупиковых ситуаций, и, пожалуй, самый простой из них основан на анализе так называемого ***графа ожидания*** — двудольного ориентированного графа, вершины которого соответствуют процессам (первая доля) и ресурсам (вторая доля). Ситуация «процесс монопольно владеет ресурсом» изображается ориентированной дугой от ресурса к процессу; ситуация «процесс заблокирован в ожидании освобождения ресурса» изображается дугой от процесса к ресурсу. **Появление в графе ожидания ориентированных циклов означает, что система зашла в тупик**.

На рис. 7.7 даны примеры графа ожидания. Слева показан граф ожидания с четырьмя процессами и шестью ресурсами; в этой системе

тупиковой ситуации пока нет. В середине приведён пример простейшей тупиковой ситуации (этот пример нами уже рассматривался на стр. 270). Справа показан граф ожидания для задачи о пяти философах в тот момент, когда все пятеро одновременно взяли по одной (левой) вилке.

Редукция графа ожидания состоит в пошаговом отбрасывании от него дуг, имеющих начало в вершине, в которую ни одна дуга не входит, или конец в вершине, из которой ни одна дуга не выходит. Если в какой-то момент в графе ещё остаются дуги, но ни одна из них не может быть отброшена, констатируется наступление ситуации тупика.

В системах, где количество разделяемых данных и обращающихся к ним процессов делает тупиковые ситуации практически неизбежными, обращения к разделяемым данным объединяют в ***транзакции***, каждая из которых до своего завершения помнит, какие из данных поменяла, и может быть подвергнута ***откату***, при котором все изменённые значения восстанавливаются к состоянию, имевшему место на момент начала транзакции. В графе ожидания в этом случае фигурируют не процессы, а транзакции. Система периодически производит редукцию графа ожидания, а при обнаружении ориентированного цикла сравнивает стоимость отката каждой из транзакций, участвующих в цикле, и, откатив самую дешёвую, продолжает редукцию графа ожидания, пока он не опустеет. Такой транзакционный доступ к данным применяется в системах управления базами данных (СУБД). Способность транзакции откатываться к началу существенно отличает её от критической секции; впрочем, критические секции при транзакционном доступе тоже приходится применять, но они сводятся к тому, чтобы прочитать или обновить одну запись в базе данных и пометить эту запись как закреплённую за транзакцией, так что транзакция за время своей работы может побывать в изрядном количестве критических секций. Транзакционная модель работы позволяет не заводить по мьютексу для каждой записи, что при наличии нескольких миллиардов записей попросту технически невозможно.

7.2.4. Проблема читателей и писателей

Ещё один интересный пример связан с базой данных, к которой одни процессы («читатели») осуществляют доступ только на чтение, а другие («писатели») могут производить запись (и чтение, разумеется, тоже).

Как мы неоднократно убеждались, доступ нескольких процессов на запись одних и тех же данных приводит к проблемам (ситуациям гонок). Даже если модификацией разделяемых данных занимается только один из двух или нескольких процессов, а остальные довольствуются их чтением, это всё равно может привести к ситуации гонок, как мы уже видели в примере с подсчётом остатков денег на банковских счетах

(см. стр. 252). В то же время процессы, осуществляющие одновременный доступ только на чтение (без вмешательства пишущих процессов), помешать друг другу не могут. Задача читателей и писателей состоит в том, чтобы позволить одновременный доступ к данным произвольному числу читателей, но при этом так, чтобы наличие хотя бы одного читателя исключало доступ писателей, а наличие хотя бы одного писателя исключало доступ вообще кого бы то ни было, включая и читателей.

Для решения задачи введём общую переменную, которая будет показывать текущее количество читателей (назовём её `rc` от слов *readers count*). Это позволит первому пришедшему читателю узнать, что он первый, и блокировать доступ к базе для писателей, а последнему уходящему читателю — узнать, что он последний, и разблокировать доступ. Для блокировки доступа к данным воспользуемся мьютексом `db_mutex`, а для защиты целостности переменной `rc` нам потребуется ещё один мьютекс — `rc_mutex`. Процедура записи в область общих данных («писатель») будет достаточно простой:

```
void writer(...)
{
    lock(db_mutex);
    /* ... пишем данные в общую память ... */
    unlock(db_mutex);
}
```

Процедура «читателя» окажется существенно сложнее, поскольку требует манипуляций с переменной `rc`. Читатель прежде всего проверяет, не первый ли он среди читателей. Если есть другие читатели, осуществляющие доступ к общей памяти в этот момент, читатель просто присоединяется к ним, отразив факт своего присутствия в переменной `rc`; если же других читателей нет, первый пришедший читатель сначала дожидается, пока критическую секцию не покинет писатель (если, конечно, он там есть) — это делается с помощью блокировки мьютекса `db_mutex`. Если в это время к входу в критическую секцию подойдут другие читатели, они блокируются на мьютексе, защищающем переменную `rc` (`rc_mutex` в этот момент всё ещё удерживает читатель, пришедший первым).

```
void reader(...)
{
    lock(rc_mutex);
    rc++;
    if(rc == 1)                    /* первый! */
        lock(db_mutex);
    unlock(rc_mutex);
    /* ... читаем данные из общей памяти ... */
    lock(rc_mutex);
```

```
    rc--;
    if(rc == 0)        /* уходя, гасите свет */
        unlock(db_mutex);
    unlock(rc_mutex);
}
```

Приведённое решение имеет серьёзный недостаток, неизменно ускользающий от внимания теоретиков. Если читателей много, их число может очень долго не достигать нуля: одни читатели будут уходить, выполнив свою задачу, но на смену им будут появляться другие. Для иллюстрации рассмотрим какой-нибудь телефонный справочный колл-центр линий на пятьсот. Если линий и операторов достаточно для обслуживания потока входящих звонков, то дождаться, пока хотя бы одна линия окажется свободна, будет несложно; но попробуйте дождаться такого момента, когда *ни одна* линия не будет занята! Такой момент может не наступить ни разу в течение нескольких лет. Представим себе, что операторы, приняв звонок клиента, зачитывают ему какой-нибудь текст, записанный на большой доске на стене колл-центра, то есть выступают в роли *читателей*, а нам нужно изменить текст на доске — часть его стереть и записать новый на его место (исполнить роль *писателя*). Чтобы не сбивать операторов с толку, мы можем решить подождать такого момента, когда ни один из операторов не будет разговаривать с клиентом. Долго же нам придётся ждать!

В параграфе, посвящённом пяти философам, мы уже встречали эту проблему и знаем, что она называется *голоданием* (*starvation*). Решить эту проблему оказывается неожиданно просто, добавив в систему ещё один мьютекс, который мы назовём `barrier`. Все читатели, прежде чем хотя бы попытаться зайти в критическую секцию, будут этот мьютекс закрывать и сразу же открывать; если мьютекс изначально открыт, никаких препятствий читателям он не создаст. Что касается писателя, то он, намереваясь войти в критическую секцию, заблокирует этот новый мьютекс, обозначив таким образом свои намерения. Заметим, что читатели никоим образом не могут помешать писателю это сделать, поскольку закрывают мьютекс лишь на очень краткое время, после чего сразу же его открывают; если же мьютекс уже закрыт другим писателем, то это никак не ухудшает нашу работу, ведь писатели всё равно не могут работать одновременно. После закрытия `barrier` писатель, скорее всего, заблокируется на попытке закрыть также `db_mutex`.

С момента блокировки мьютекса `barrier` ни один новый читатель в критическую секцию попасть более не сможет, все они будут блокироваться на том же мьютексе, при этом читатели, уже прошедшие за барьер, беспрепятственно продолжат работу, но рано или поздно все они её завершат и покинут критическую секцию. Последний из них, уходя, откроет `db_mutex`, разрешив работу писателю. Писатель, выпол-

нив свою задачу, откроет оба мьютекса, и читатели, в том числе те, что ждали на мьютексе `barrier`, снова смогут заходить в критическую секцию; фрагмент с закрытием и открытием `barrier` они преодолеют по очереди, но поскольку каждый из них потратит на это очень немного времени, то и суммарное время, которое потребуется всем ранее заблокированным на «барьере» читателям, чтобы пройти за «барьер», тоже будет невелико. Обновлённый вариант решения будет выглядеть так:

```
void reader(...)
{
    lock(barrier);        /* барьер на входе */
    unlock(barrier);
    lock(rc_mutex);
    rc++;
    if(rc == 1)
        lock(db_mutex);
    unlock(rc_mutex);
    /* ... читаем данные из общей памяти ... */
    lock(rc_mutex);
    rc--;
    if(rc == 0)
        unlock(db_mutex);
    unlock(rc_mutex);
}

void writer(...)
{
    lock(barrier); /* закрываем барьер для читателей */
    lock(db_mutex);
    /* ... пишем данные в общую память ... */
    unlock(db_mutex);
    unlock(barrier);              /* отпираем барьер */
}
```

7.2.5. Задача о спящем парикмахере

Задачу о спящем парикмахере, как и задачу о пяти философах, часто приписывают Эдсгеру Дейкстре. Приведём классическую формулировку этой задачи. Имеется парикмахерская с одним парикмахером, у которого есть кресло для работы. Кроме того, в парикмахерской есть холл, в котором стоит N стульев для клиентов, ждущих своей очереди.

Закончив стричь и отпустив очередного клиента, парикмахер идёт в холл, и если там есть ждущие клиенты, провожает одного из них в рабочее кресло и принимается стричь. Когда клиентов нет, парикмахер возвращается к своему креслу, усаживается в него и засыпает.

Клиент, зайдя в парикмахерскую, должен посмотреть, занят ли парикмахер, и если нет — то разбудить его; проснувшись, парикмахер усадит разбудившего его клиента в кресло и будет стричь. Если же парикмахер занят, клиент направляется в холл и либо занимает там один из стульев и ждёт, пока его позовут стричься, либо, если свободных стульев нет, уходит из парикмахерской нестриженым.

В реальной парикмахерской с этим сценарием работы вроде бы не должно возникнуть никаких проблем, но у нас ведь речь на самом деле идёт о взаимодействующих процессах, выполняющих определённые действия: проверить, есть ли клиенты, посмотреть, не спит ли парикмахер, занять один из стульев и т. д. Каждое такое действие в системе взаимодействующих процессов будет занимать какое-то время, причём это время, вообще говоря, заранее не известно, так что проблемы нам, как водится, обеспечены. Представим себе, к примеру, что парикмахер кого-то стрижёт, при этом холл забит почти под завязку — там есть всего одно свободное место. В парикмахерскую заходят одновременно два новых клиента, видят, что парикмахер занят, идут в холл, видят там одно свободное место и оба одновременно пытаются на него сесть; ничего хорошего из этого не выйдет.

Пусть теперь парикмахер стрижёт единственного клиента, в комнате ожидания при этом никого нет. В какой-то момент в парикмахерскую заходит клиент, заглядывает к парикмахеру, видит, что тот занят, и направляется в холл, чтобы сесть там на стул. В это же самое время парикмахер заканчивает стрижку, отпускает клиента и тоже направляется в холл за очередным клиентом, причём действует достаточно быстро, так что *опережает* только что пришедшего клиента, видит пустой холл (потому что клиент до стула ещё не дошёл), возвращается к себе и засыпает. Клиент, в свою очередь, всё-таки добирается до холла, усаживается на свободный стул и ждёт. В терминах процессов получается, что они оба заблокировались, каждый в ожидании действия второго, и вывести из этого состояния их может только кто-то третий (в парикмахерскую должен прийти ещё один клиент), а это может случиться нескоро.

Задача о спящем парикмахере допускает разные решения в зависимости от используемых средств синхронизации. Чаще всего в литературе встречается решение, в котором используется целочисленная переменная для хранения количества свободных мест в холле, мьютекс для синхронизации доступа к этой переменной, ещё один мьютекс, показывающий, занят ли парикмахер (именно на этом мьютексе блокируются клиенты в ожидании, когда их будут стричь) и семафор, равный количеству ожидающих клиентов — на нём засыпает парикмахер, когда клиентов нет:

```
int seats = TOTAL_SEATS_COUNT;
```

```
mutex seats_mutex;        /* исходно открыт */
mutex barber;             /* исходно закрыт */
semaphore customers;      /* начальное значение 0 */
```

Парикмахер будет работать по следующей схеме:

```
for(;;) {
    down(customers);      /* засыпаем, если стричь некого */
    lock(seats_mutex);
    unlock(barber);       /* приглашаем клиента */
    seats++;
    unlock(seats_mutex);
    BARBER_WORK();        /* стрижём клиента */
}
```

Здесь стоит обратить внимание, что мьютекс `barber`, на котором «спят» клиенты, разблокируется внутри критической секции по переменной `seats`. Если разблокировать его до начала критической секции, то клиент, зашедший в парикмахерскую в это же самое время, может зайти в критическую секцию, увидеть, что свободных стульев нет и уйти, хотя на самом деле один из стульев только что освободился. Напротив, если его разблокировать после критической секции, то такой (зашедший в парикмахерскую «в неподходящий момент») клиент может попытаться занять *несуществующий* свободный стул: в самом деле, между критической секцией и приглашением стричься возникает момент, когда счётчик свободных стульев уже увеличен, но клиента пока никто не стрижёт, так что реально он всё ещё занимает стул в холле (количество клиентов, заблокированных на мьютексе `barber`, равно количеству стульев). Парикмахер избегает обеих некорректных ситуаций, соединив два действия — приглашение очередного клиента на стрижку и освобождение стула в холле — в одно неделимое мероприятие, когда никто другой не может «влезть» в происходящее.

Поведение клиента будет чуть более сложным:

```
lock(seats_mutex);
if(seats > 0) {
    seats--;
    up(customers);   /* если парикмахер спал --
                        это его разбудит */
    unlock(seats_mutex);
    lock(barber);    /* если парикмахер занят -- ждём */
    CUSTOMER_WORK(); /* получаем услуги по стрижке волос */
} else {
    /* мест нет -- придётся уйти нестриженым */
    unlock(seats_mutex);
}
```

Клиент начинает с того, что входит в критическую секцию — для доступа к переменной `seats`, но не только. Если в холле нет свободных мест, клиент сразу же освобождает мьютекс, выходя тем самым из критической секции, и покидает парикмахерскую, не получив желаемого. Если же свободные места в холле есть, клиент для начала занимает одно из них (уменьшает переменную `seats` на единицу), затем поднимает на единичку семафор, выходит из критической секции и ждёт, когда его пригласят стричься — для этого он пытается захватить мьютекс `barber`, в результате чего блокируется до тех пор, пока парикмахер этот мьютекс не откроет; если есть несколько клиентов, заблокировавшихся на мьютексе `barber`, то согласно определению мьютекса каждый раз, когда парикмахер этот мьютекс открывает, разблокируется при этом только один из клиентов — именно он и идёт стричься.

Здесь можно заметить, что *клиент будит парикмахера* внутри критической секции по той же переменной. Причина этого, опять же, в том, что в парикмахерскую в любой момент может зайти ещё один клиент. Если клиенты станут будить парикмахера до того, как займут свои стулья, то два клиента одновременно могут увеличить семафор (тем самым они заявят парикмахеру, что уже ждут, когда их постригут), но потом один из них обнаружит, что ему не хватает стула, и будет вынужден уйти; казалось бы, он при этом может уменьшить семафор, и всё будет в порядке, но это не так: если клиент попадётся достаточно нерасторопный, то за время, прошедшее между обнаружением отсутствия стула и уменьшением семафора, парикмахер может успеть перестричь всю имеющуюся очередь клиентов и попытаться пригласить стричься клиента, который вообще-то на это не рассчитывает (он уже принял решение уйти из парикмахерской нестриженым, то есть его программа уже выполняется по ветке, не предполагающей ни стрижку, ни ожидание). В терминах наших мьютексов и семафоров это будет означать, что парикмахер повторно откроет мьютекс `barber`, который и так уже открыт, и попытается начать стрижку, не имея для этого клиента.

Остаётся вопрос, что же будет, если попробовать увеличивать семафор `customers` *после* критической секции. Сценарий, демонстрирующий некорректность этого варианта, оказывается ещё хитрее. Допустим, парикмахер спит; в это время в парикмахерскую почти одновременно заходят два клиента и по очереди занимают свободные стулья (ещё не успев разбудить парикмахера, то есть не успев поднять семафор `customers`). Потом оба по очереди «будят» парикмахера, но что-то отвлекает их, и дальше они пока что не идут (в реальной жизни процесс может оказаться, например, откачан на диск и будет сравнительно долго находиться в состоянии блокировки в ожидании, пока система его не подкачает обратно). Парикмахер, подскочив в своём кресле, увеличива-

ет количество свободных стульев, зовёт одного из клиентов стричься и приступает к стрижке (в наших примерах — исполняет процедуру `BARBER_WORK`. В реальной жизни, конечно, парикмахер не сочтёт стрижку законченной (и даже начатой), пока клиент не окажется в кресле, но ведь наша парикмахерская — не настоящая; коль скоро парикмахер не спит на семафоре `customers`, остановиться ему больше негде. Процесс, работающий в роли парикмахера, проскочит действия, предусмотренные процедурой `BARBER_WORK`, и снова попытается заснуть на семафоре, но ведь семафор был поднят *дважды*, так что он не заснёт и следующим своим действием попытается открыть мьютекс `barber`, который и так уже открыт — и снова примется за «стрижку» (уже второй раз!), хотя ни один из клиентов ещё не выполнил свою часть процедуры стрижки — `CUSTOMER_WORK`.

Если клиенты будут будить парикмахера внутри критической секции, ничего подобного не произойдёт: клиенты, как и положено, будут обслуживаться строго по одному.

Осознанию изложенной проблемы обычно мешает наше представление о процессе стрижки — нам кажется, что парикмахер никак не может начать стрижку, если клиент ещё не уселся в его рабочее кресло; но в применении к задаче о взаимоисключениях и синхронизации это означало бы, что внутри процедур `BARBER_WORK` и `CUSTOMER_WORK` присутствуют *какие-то ещё средства синхронизации* помимо тех, которые мы рассматриваем в решении задачи. Между тем, наша задача как раз и состоит в том, чтобы описать *все* используемые средства синхронизации и то, как с их помощью организовано взаимодействие процессов.

7.3. Многопоточное программирование в ОС Unix

В области программирования, как и в других инженерных дисциплинах, никакая теория не стоит ломаного гроша, если она не подкреплена практикой. Наш рассказ о критических секциях, мьютексах, семафорах и классических задачах останется пустопорожней болтовнёй, если всё это будет не на чем попробовать.

Автор этих строк, будучи жёстким противником применения многопоточного программирования, вынужден, тем не менее, признать, что среди всех имеющихся окружений, содержащих семафоры и мьютексы, наиболее простым и доступным следует признать семафоры и мьютексы POSIX, реализованные в связке с многопоточностью и исходно как раз для многопоточных программ предназначавшиеся. Именно это обстоятельство сыграло определяющую роль при выборе иллюстративного

материала для версии курса «Операционные системы», который автору довелось читать на факультете ВМК МГУ с 2004 по 2012 год.

Кроме методических соображений, в пользу рассказа о многопоточном программированиии говорит упоминавшееся уже выше обстоятельство: даже если мы отказываемся от его использования, делать это нужно осознанно и при этом точно знать, от чего конкретно мы отказались. В частности, к сожалению, у вашего будущего работодателя может возникнуть странное подозрение, что вы, мол, просто не знаете, как работать с тредами и потому не хотите с ними работать; чтобы опровергнуть подобные домыслы, вам нужно будет знать, как работать с тредами, и уметь это продемонстрировать.

Прежде чем приступить к рассказу о функциях работы с тредами, напомним некоторые базисные моменты. **Легковесные процессы, они же «треды», запускаются в рамках одного обычного процесса**, работают в его адресном пространстве (попросту говоря, в его памяти) и, как следствие, имеют ничем не ограниченный доступ ко всем переменным своего основного процесса, а равно и других тредов, работающих в нём; естественно, основной процесс, называемый также «основным тредом», тоже имеет ничем не ограниченный доступ ко всем переменным всех своих «дополнительных» тредов. Каждый тред представляет собой самостоятельную *единицу планирования* с точки зрения планировщика времени центрального процессора; кроме того, у каждого треда имеется свой собственный стек, что и понятно, ведь нужно же ему где-то располагать параметры функций, локальные переменные и адреса возврата; но на этом «самостоятельность» треда заканчивается, даже небезызвестный идентификатор процесса (`pid`) у тредов «один на всех» — вызовы `getpid` и `getppid` во всех тредах одного процесса возвращают одни и те же значения.

Вообще, если речь идёт о тредах, то можно обнаружить, что практически ничего *действительно своего* у них нет. Даже стек, который заводится специально для каждого запускаемого треда, в действительности располагается в общем адресном пространстве, так что тред этим стеком пользуется, но нельзя в полной мере сказать, что он им *владеет*.

7.3.1. Библиотека `pthread`

В 1995 году был принят стандарт, описывающий функции управления тредами, под общим названием ***Posix Threads*** (`pthread`). Этот стандарт в той или иной степени поддерживается во всех операционных системах семейства Unix, а также и в системах линии Windows. Согласно pthread, тред должен иметь главную функцию (аналогично тому, как процесс имеет функцию `main`) следующего вида:

```
void* my_thread_main(void *arg)
{
    /* ... */
}
```

Как мы видим, треду в качестве стартового параметра можно передать указатель на произвольную область памяти (`void*`); тред может при завершении сообщить другим потокам результат своей работы в виде опять-таки произвольного указателя. Здесь прослеживается некоторая аналогия с обычными процессами, которые при запуске получают в качестве аргумента главной функции командную строку, а при завершении формируют числовой код возврата (см. § 5.4).

Каждый поток имеет свой уникальный идентификатор, который можно сохранить в переменной типа `pthread_t`. Для создания (и запуска) потока используется функция `pthread_create`:

```
int pthread_create(pthread_t* thr, pthread_attr_t* attr,
                   void*(*start_routine)(void*), void* arg);
```

Параметр `thr` указывает, в какую переменную следует записать идентификатор нового потока. Аргумент `attr` позволяет задать специфические параметры работы нового потока; в большинстве случаев такие параметры не нужны, так что можно в качестве этого аргумента передать нулевой указатель. Параметр `start_routine` указывает на главную функцию потока. Именно эта функция будет запущена в новом потоке, причём на вход ей будет передан указатель, который при вызове `pthread_create` мы указали в качестве параметра `arg`. Функция `pthread_create`, как и все функции этого семейства, возвращает `0` в случае успеха либо код ошибки, если создать новый поток не удалось; для `pthread_create` это может быть только код `EAGAIN`, который означает, что для создания потока не хватило системных ресурсов или было достигнуто предельное количество потоков для одного процесса.

Тред может завершиться двумя способами: вернув управление из своей главной функции подобно тому, как процесс возвращает управление из функции `main`, или вызвав функцию `pthread_exit`:

```
void pthread_exit(void *retval);
```

В первом случае результатом работы потока станет значение, возвращённое из главной функции (напомним, оно имеет тип `void*`), во втором случае — значение аргумента `retval`. Здесь снова прослеживаются аналогии с управлением процессами, на сей раз — с функцией `exit`.

Поток может дождаться завершения другого потока с помощью функции `pthread_join`:

```
int pthread_join(pthread_t th, void **result);
```

Аргумент `th` задаёт идентификатор треда, завершения которого мы хотим подождать. Через параметр `result` передаётся *адрес* указателя типа `void*`, в который следует записать результат работы потока. Это несколько напоминает функционирование вызова `waitpid()` для обычных процессов.

Результат выполнения завершённого потока должен где-то храниться; если его не востребовать вызовом `pthread_join`, он будет впустую занимать системные ресурсы, как это происходит с процессами (зомби). Однако для потоков этого можно избежать, переведя поток в ***«отсоединённый» режим*** (англ. *detached mode*). Это делается функцией `pthread_detach`:

```
int pthread_detach(pthread_t th);
```

Недостаток «отсоединённых» потоков в том, что их невозможно дождаться с помощью `pthread_join` и, соответственно, нет способа проанализировать результат их работы. Функции `pthread_detach` и `pthread_join` возвращают, как и `pthread_create`, `0` в случае успеха или код ошибки, если выполнить действие не удалось.

Узнать свой собственный идентификатор поток может с помощью функции `pthread_self`:

```
pthread_t pthread_self();
```

Например, поток может перевести сам себя в «отсоединённый режим», выполнив вызов

```
pthread_detach(pthread_self());.
```

Поток может досрочно завершить другой поток, вызвав функцию

```
int pthread_cancel(pthread_t th);
```

В этом случае результатом работы потока `th` будет специальное значение `PTHREAD_CANCELED`. Следует отметить, что вызов `pthread_cancel` не уничтожает поток, а *отменяет* его, что, вообще говоря, не всегда приводит к немедленному прекращению выполнения другого потока: возможно, что поток завершится, только дойдя до вызова одной из функций библиотеки pthread, входящей в число ***точек отмены*** (англ. *cancellation points*). Такие функции, кроме основных действий, производят проверку наличия запроса на отмену данного потока. Список функций, являющихся точками отмены, можно узнать из документации на `pthread_cancel`.

7.3.2. Семафоры и мьютексы

Библиотека `pthread` включает как семафоры, так и мьютексы, но мьютексы из `pthread` были изначально предназначены для взаимоисключений между тредами, тогда как интерфейс семафоров исходно был задуман как допускающий взаимодействие между обычными процессами; поэтому функции для работы с семафорами, как мы увидим чуть позже, не имеют префикса `pthread_` в именах.

В качестве мьютексов pthread использует переменные типа `pthread_mutex_t`, определение которого в большинстве реализаций достаточно сложное, но нас оно, вообще говоря, волновать не должно — это проблемы создателей конкретной реализации библиотеки. Начальное значение такой переменной, соответствующее состоянию «мьютекс открыт», следует задать инициализатором `PTHREAD_MUTEX_INITIALIZER`, например:

```
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Возможны и другие, более сложные варианты инициализации мьютекса, в том числе с помощью специальной функции, однако для наших иллюстративных целей достаточно одного. Следует обратить внимание, что `PTHREAD_MUTEX_INITIALIZER` представляет собой именно *инициализатор*, т. е., вообще говоря, попытка *присвоить* мьютексу это значение, скорее всего, приведёт к ошибке при компиляции: результат макроподстановки этого макроса может содержать (и практически всегда содержит) фигурные скобки, допустимые в инициализаторе, но не предусмотренные в обычных выражениях.

Напомним, что под мьютексом понимается объект, способный находиться в одном из двух состояний (открытом и закрытом), над которым определены две операции: открытие (`unlock`) и закрытие (`lock`), причём первая всегда переводит мьютекс в открытое состояние и возвращает управление, вторая же, если ее применить к открытому мьютексу, закрывает его и возвращает управление, если же её применить к закрытому мьютексу, может либо вернуть управление, сигнализируя о неудаче (неблокирующий вариант), либо блокировать вызвавший процесс (или, в данном случае, поток), дождаться, пока кто-то не откроет мьютекс, закрыть его и только после этого вернуть управление (блокирующий вариант). В pthread основные операции над мьютексами производятся с помощью функций

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Эти функции осуществляют открытие мьютекса (`unlock`), блокирующее закрытие мьютекса (`lock`) и неблокирующее закрытие мьютекса

(`trylock`). Все функции возвращают 0 в случае успеха или ненулевой код ошибки, причём в случае, если `pthread_mutex_trylock` применяется к закрытому мьютексу, она возвращает код `EAGAIN`. Мьютекс можно уничтожить вызовом функции `pthread_mutex_destroy`:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

которая высвободит используемые мьютексом ресурсы, если таковые есть; в реализации мьютексов в ОС Linux весь мьютекс целиком умещается в переменной `pthread_mutex_t`, так что высвобождать оказывается нечего. На момент уничтожения мьютекса он должен находиться в состоянии «открыт», иначе функция вернёт ошибку, и если говорить о реализации мьютексов, имеющейся в ОС Linux, то проверкой этого условия действия `pthread_mutex_destroy`, собственно говоря, и ограничиваются. В других реализациях это может быть не так, поэтому, когда очередной мьютекс становится вам не нужен, к нему, прежде чем соответствующая переменная исчезнет, нужно обязательно применить `pthread_mutex_destroy` и проверить, что она вернула значение «успех» (т. е. ноль). Впрочем, это следует сделать, даже если ваша программа предназначена для работы исключительно в Linux, поскольку физическое уничтожение переменной-мьютекса, на которой кто-то из ваших тредов всё ещё заблокирован, обычно приводит к аварии, но не сразу: программа может ещё некоторое время проработать и лишь затем «свалиться», так что найти причину может оказаться очень сложно. Ненулевое значение, возвращённое при попытке уничтожить мьютекс, позволит понять, что ваша программа работает некорректно, и исправить ошибку, не тратя лишнее время на поиски.

Семафоры POSIX представляются переменными типа `sem_t`; ещё раз отметим, что префикс `pthread_` тут не используется. Дело в том, что спецификация семафоров POSIX исходно предназначалась для работы не только с тредами, но и с обычными процессами; хотя мы такой вариант рассматривать не будем, всё же придётся иметь в виду, что он был предусмотрен.

Некоторые реализации семафоров POSIX не предусматривали возможности взаимодействия через них для обычных процессов, выдавая ошибку при попытке такого их использования; в частности, так до сравнительно недавних пор обстояли дела в Linux. Современные реализации, в том числе и в Linux, такое использование допускают; для этого переменную-семафор нужно разместить в разделяемой памяти и при её инициализации указать, что к ней предполагается разделяемый доступ (т. е. доступ из разных процессов).

Инициализация семафора производится функцией `sem_init`:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Параметр `sem` задаёт адрес инициализируемого семафора. Параметр `pshared` указывает, будет ли семафор доступен для других процессов; если наша переменная типа `sem_t` описана как обычная переменная (не размещена в разделяемой памяти), то такое использование в любом случае невозможно, так что параметр `pshared` должен быть равен нулю. Наконец, параметр `value` задаёт начальное значение семафора. Функция `sem_init` возвращает `0` в случае успеха, `-1` в случае ошибки, а сам код ошибки заносится в переменную `errno`. Как видим, даже подход к возвращаемому значению для «семафорных» функций отличается от принятого в `pthread`; здесь используется конвенция, характерная для системных вызовов. В дальнейшем изложении мы не будем делать на этот счёт специальных оговорок, поскольку *все* функции, составляющие интерфейс к семафорам POSIX, ведут себя именно так: возвращают `0` при успехе, а при неудаче возвращают `-1` и заносят код ошибки в `errno`. Пусть вас, впрочем, не обманывает используемая конвенция; ни одна из этих функций не является системным вызовом и вообще не обращается к ядру, кроме случая, когда им не остаётся иного выхода, кроме как «заснуть».

Как мы помним, семафор есть по определению объект, внутреннее состояние которого представляет собой неотрицательное целое число, и над которым определены две операции: `up` и `down`. Первая из них увеличивает значение семафора на `1` и немедленно возвращает управление. Вторая, если значение семафора равно нулю, блокирует вызвавший процесс или поток до тех пор, пока кто-то другой не увеличит значение семафора (если значение изначально ненулевое, блокировки не происходит), после чего уменьшает значение семафора на `1` и возвращает управление. Для семафоров POSIX соответствующие операции выполняются функциями

```
int sem_post(sem_t *sem);    /* up   */
int sem_wait(sem_t *sem);    /* down */
```

Также имеется неблокирующий вариант операции `down`:

```
int sem_trywait(sem_t *sem);
```

Если семафор на момент вызова имеет значение `0`, эта функция, вместо того чтобы блокировать вызвавший процесс, немедленно завершается, возвратив значение `-1` и установив `errno` в значение `EAGAIN`.

Текущее значение семафора можно узнать с помощью функции `sem_getvalue`:

```
int sem_getvalue(sem_t *sem, int *sval);
```

Значение семафора возвращается через параметр `sval`.

Если семафор больше не нужен, его следует ликвидировать с помощью функции `sem_destroy`:

```
int sem_destroy(sem_t *sem);
```

При этом не должно быть ни одного треда (а для случая «разделяемых» семафоров — ни одного процесса), находящегося в состоянии ожидания на этом семафоре, то есть выполняющего в настоящий момент `sem_wait` с тем же параметром, что и `sem_destroy`. Надо сказать, что с семафорами в этом плане дело обстоит гораздо жёстче, чем с мьютексами: `sem_destroy` не проверяет, есть ли или нет такие треды или процессы, а просто приводит к неопределённому поведению (*undefined behaviour*).

7.3.3. Пример

Рассмотрим задачу, в которой нам потребуется реализовать взаимоисключение вида «производители-потребители». Пусть даны несколько источников данных, из которых поступают в текстовом виде числа с плавающей точкой. В роли источников могут выступать обычные файлы, а также сокеты, FIFO или символьно-ориентированные устройства, то есть прочитать последовательно сначала один источник, потом другой и т. д. нельзя. Получаемые из источников числа нужно подвергнуть определённой обработке: вычислить для каждого числа натуральный логарифм, а результат учесть таким образом, чтобы в каждый момент времени можно было выдать среднее арифметическое вычисленных значений (для этого достаточно, например, хранить сумму всех результатов и их общее количество).

Вычисление логарифмов требует заметного расхода процессорного времени. Допустим, в нашей системе может быть несколько физических процессоров, так что применение параллельных потоков способно ускорить обработку. Вместе с тем неизвестно, с какими скоростями будут поступать числа от источников, причём, возможно, эти скорости окажутся существенно непостоянны. В результате процессоры могут оказаться часть времени перегружены работой, что приведёт к задержкам в приёме данных от источников, а часть времени — простаивать. Чтобы сгладить эти эффекты, можно использовать единый буфер данных достаточной вместимости.

Реализуем задачу в многопоточной схеме, выделив по одному потоку на чтение каждого источника данных, и запустим N потоков для обработки данных (логарифмирования и суммирования). Передавать данные от первых к последним будем через общий буфер по схеме «производители-потребители». Поскольку суммирование придётся вести в общих переменных, доступ к ним нужно будет оформить в виде критических секций, для взаимоисключения которых потребуется мьютекс. Чтобы уменьшить потери на ожидание потоками освобождения этого мьютекса, будем накапливать данные в локальных переменных потока, а доступ к глобальной сумме осуществлять по неблокирующему принци-

пу: если захватить мьютекс удалось, сбрасываем накопленные данные, иначе работаем дальше, накапливая данные в локальном сумматоре.

Имена файлов источников программа получит через аргументы командной строки. Чтобы работать с программой было интереснее, сделаем так, чтобы главная программа каждые пять секунд выдавала значение суммы и вычисленного среднего, а счетчики обнуляла. Наконец, по мере исчерпания источников (при возникновении ситуации «конец файла») будем завершать потоки-«производители». Поскольку с завершением последнего «производителя» дальнейшая работа программы теряет смысл, предусмотрим механизм подсчета оставшихся «производителей». Чтобы не возиться с мьютексом и глобальной переменной, воспользуемся обыкновенным семафором; в этом случае будем использовать семафор исключительно ради атомарности действий над ним, без блокировок.

Полностью программа приведена в листинге на стр. 297–299. Чтобы опробовать программу, создайте несколько именованных каналов с помощью команды `mkfifo`. Запустив несколько программ `xterm`, создайте процессы, читающие с клавиатуры и пишущие в только что созданные каналы. Это проще всего сделать с помощью команды `cat` и перенаправления вывода. В отдельном окне запустите программу, указав ей в командной строке имена каналов. Теперь можно вводить числа на вход программам `cat`; программа будет их обрабатывать.

Заключение

Подчеркнём ещё раз, что с механизмом тредов можно поэкспериментировать, но прежде чем его использовать на практике, следует очень серьёзно подумать. Обратиться к разделяемым данным можно случайно, не заметив этого, а в некоторых случаях даже не подозревая, что что-то подобное происходит: например, ваш коллега по проекту или автор какой-нибудь библиотеки может использовать в своих функциях глобальную или статическую переменную, но забыть вам об этом сказать, и вы, вызвав его функции из разных тредов, устроите себе ситуацию гонок; что самое противное, такая ситуация гонок может никак себя не проявлять, пока вы не закончите тестирование, а затем «выстрелит» в самый неподходящий момент, например, у заказчика в другом городе.

Применение многопоточного программирования оправдано только в одной ситуации: когда, разбросав независимые части одного вычисления по разным тредам, вы можете добиться видимого выигрыша по времени исполнения за счёт использования нескольких ядер центрального процессора. Отметим, что автор книги не встретил ни одной такой задачи за всю свою практику.

Листинг примера «производители-потребители»

```
/* prod_cons.c */
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <math.h>

#define BUFFER_SIZE 4096

/* Буфер обмена между производителями и потребителями */
struct buf_str {
    int count;
    double values[BUFFER_SIZE];
} buffer;
void init_buffer()
{
    buffer.count = 0;
}
void put_buffer_item(double v)
{
    buffer.values[buffer.count] = v;
    buffer.count++;
}
double get_buffer_item()
{
    buffer.count--;
    return buffer.values[buffer.count];
}

/* семафоры и мьютекс для организации работы с буфером */
sem_t buf_empty;
sem_t buf_full;
pthread_mutex_t buf_mutex = PTHREAD_MUTEX_INITIALIZER;

/* переменные для суммирования и мьютекс для их защиты */
double grand_total = 0;
long grand_count = 0;
pthread_mutex_t grand_mutex = PTHREAD_MUTEX_INITIALIZER;

/* семафор для подсчета оставшихся "производителей" */
sem_t producers_count;
```

```
/* Поток для чтения данных ("производитель") */
void *producer_thread(void *v)
{             /* получаем в v указатель на имя источника */
    double val;
    FILE *f = fopen((char*)v, "r");
    if(!f) {
        perror((char*)v);
        sem_wait(&producers_count);
        return NULL;
    }
    while(!feof(f)) {
        if(1 != fscanf(f, "%lf", &val))
            continue;
        sem_wait(&buf_empty);     /* алгоритм производителя */
        pthread_mutex_lock(&buf_mutex);
        put_buffer_item(val);
        pthread_mutex_unlock(&buf_mutex);
        sem_post(&buf_full);      /*        -------          */
    }
    sem_wait(&producers_count);
    return NULL;
}
/* Поток-потребитель. Получаемое входное значение игнорирует */
void *consumer_thread(void *ignored)
{
    double local_total = 0; /* локальные сумматоры */
    long local_count = 0;
    for(;;) {
        double val;
        sem_wait(&buf_full);      /* алгоритм потребителя */
        pthread_mutex_lock(&buf_mutex);
        val = get_buffer_item();
        pthread_mutex_unlock(&buf_mutex);
        sem_post(&buf_empty);     /*        -------       */
        /* теперь можно заняться вычислениями */
        local_total += log(val); local_count++;
        /* если есть возможность, сбрасываем данные */
        if(0==pthread_mutex_trylock(&grand_mutex)) {
             grand_total += local_total;
             grand_count += local_count;
             local_total = 0; local_count = 0;
             pthread_mutex_unlock(&grand_mutex);
        }
    }
}
```

```
int main(int argc, char **argv)
{
    pthread_t thr;
    int i;
    /* инициализируем глобальные данные */
    init_buffer();
    sem_init(&buf_empty, 0, BUFFER_SIZE);
    sem_init(&buf_full, 0, 0);
    sem_init(&producers_count, 0, 0);
    /* запускаем "производителей" */
    for(i = 1; i<argc; i++) {
        sem_post(&producers_count);
        pthread_create(&thr, NULL, producer_thread, argv[i]);
    }
    /* запускаем "потребителей" */
    for(i = 0; i<10; i++)
        pthread_create(&thr, NULL, consumer_thread, NULL);

    /* теперь каждые 5 секунд печатаем и обнуляем результат */
    for(;;) {
        int p_c;
        sleep(5);
        pthread_mutex_lock(&grand_mutex);
        /* во избежание деления на 0 проверим наличие данных */
        if(grand_count>0) {
            printf("total average: %f (sum = %f; count = %ld)\n",
                   grand_total/((double)grand_count),
                   grand_total, grand_count);
        } else {
            printf("No data yet...\n");
        }
        grand_total = 0; grand_count = 0;
        pthread_mutex_unlock(&grand_mutex);
        sem_getvalue(&producers_count, &p_c);
        if(p_c == 0) {
            printf("No more producers\n");
            break;
        }
    }
    return 0;
}
```

7.4. Разделяемые данные на диске

Избежать возникновения разделяемых данных в оперативной памяти обычно можно (а с определённых точек зрения — и нужно); но, как мы уже отмечали в предисловии к этой части книги (см. стр. 250), данные на внешних запоминающих устройствах (т. е. на дисках) во многих случаях оказываются разделяемыми в силу природы поставленной задачи, и поделать здесь ничего нельзя. Впрочем, прежде чем начать рассказ об имеющихся инструментах — которые, к сожалению, все как один кривые — позволим себе один совет: покуда вы *можете* избежать ситуации доступа к одному файлу нескольких программ или процессов, избегайте её. Средства, рассмотренные в этой главе, следует использовать лишь тогда, когда иного выхода нет.

7.4.1. Обзор имеющихся возможностей

Для начала нам придётся условиться о терминах. Операционная система предоставляет нам (т. е. программисту; или, с другой точки зрения, *процессу*) средства, с помощью которых мы можем известить другие процессы в системе, что данный файл у нас прямо сейчас находится в работе, так что доступ к нему может привести к нежелательным последствиям. Эта сущность по-английски называется *file lock*. В большинстве русскоязычных источников слово *lock* в этом контексте переводят русским словом «блокировка», но такой вариант неудачен: мы уже активно использовали термин «блокировка», оригиналом которого послужило слово *blocking*.

В повседневной работе программисты обычно вообще не переводят слово *lock* с английского, так что вы вполне можете услышать выражения вроде «залочить файл», «сбросить лок» и т. п. Конечно, это откровенный сленг, но использование сленга в некоторых случаях можно считать допустимым; вот только проблема в том, что слово «лок», будучи напечатанным в тексте, смотрится совершенно кошмарно, тут даже можно не сразу догадаться, о чём идёт речь — в отличие от ситуации, когда оно произносится, но не пишется. Поскольку нам предстоит активное использование соответствующего термина в тексте книги, придётся подыскать что-то более подходящее.

Не найдя ничего лучшего, мы воспользуемся для перевода термина *lock* словом «захват». Этот вариант тоже имеет свои недостатки, и мы это с готовностью признаём, но другого, более удачного перевода нам придумать не удалось. Если вам известен лучший вариант, свяжитесь с автором книги.

Отметим, что файловые захваты, имеющиеся в unix-системах, в большинстве своём никого ни к чему не обязывают: процесс, имеющий

достаточно прав на чтение файла, может его читать, а имеющий права на запись — может файл модифицировать, и установленные другими процессами захваты никак ему не помешают. По-английски такие «ни к чему не обязывающие» захваты называются *advisory locks*; чтобы не усугублять и без того плачевную ситуацию с терминами, мы не будем пытаться перевести в этом контексте слово *advisory* на русский язык, а любителям терминов вроде «консультационный захват» или «консультативная блокировка» посоветуем прекратить издеваться над родным языком. Поскольку файловые захваты практически всегда именно таковы, мы просто не будем это каждый раз уточнять.

Поскольку установление захвата на файл реально не создаёт препятствий к чтению или записи его содержимого, ясно, что для успешной работы все процессы, использующие данный файл, должны *добровольно* придерживаться установленного порядка работы. Само по себе это не так уж страшно — достаточно вспомнить, что мьютексы и семафоры тоже сами по себе ничему не препятствуют. Здесь важно другое: если какая-нибудь из программ, работающих с файлом, не будет ничего знать про захваты, то все наши усилия по обеспечению корректной работы пойдут прахом; между тем такую программу может — просто по незнанию — запустить сам пользователь, и с этим мы ничего не сделаем.

В литературе и Интернете вам могут встретиться упоминания *обязательных захватов* (*mandatory locks*) — таких, установка которых реально исключает выполнение другими процессами операций чтения и записи. Этот механизм так и не получил широкого распространения. В системах семейства BSD вообще отсутствует поддержка обязательных захватов; в ОС Linux они поддерживаются ядром, но для этого нужно для начала смонтировать файловую систему с параметром, указывающим на их применение, и, кроме того, установить «хитрые» права доступа к файлу, на котором захваты должны иметь «обязательный» характер (см. комментарий на стр. 50). Автор этих строк ни разу за всю свою практику не видел, чтобы файловые системы монтировались таким образом; сам механизм, введённый в ядро в 1996 году, изначально считался и продолжает считаться проблемным [14]. Исходя из предположения, что обязательные захваты в реальной жизни не применяются, мы, по-видимому, будем недалеки от истины; поэтому рассматривать этот вариант файловых захватов мы не станем.

Основных подходов к захвату доступа к файлу существует три, причём один из них вообще не требует поддержки со стороны операционной системы: он заключается в том, что рядом с захватываемым файлом создаётся ещё один файл, который показывает другим процессам, что в настоящее время осуществление доступа нежелательно. Два других механизма, известные как *BSD locks* (системный вызов `flock`)

и *POSIX locks* (функция `lockf`, работающая через системный вызов `fcntl`) реализованы в ядре операционной системы; коль скоро один процесс установил файловый захват, другим процессам ядро откажет в предоставлении захвата, конфликтующего с уже установленным.

Отметим одно неочевидное обстоятельство, которое следует учитывать в работе. Среди всех файловых систем несколько особое место занимают такие, которые физически находятся на другом компьютере (файловом сервере); доступ к ним осуществляется через компьютерную сеть. **Файловые захваты, поддерживаемые на уровне операционной системы, не работают для файлов, находящихся на сетевых дисках**. Если написанные вами программы в своей работе полагаются на эти механизмы, нужно обязательно отразить в документации, что соответствующие файлы должны находиться на локальных файловых системах, в противном случае всё перестанет работать. Прежде чем вы это сделаете, учтите, что в реальной практике часто встречаются компьютеры, не имеющие никаких дисков, кроме сетевых; декларируя свою зависимость от файловых захватов, поддержанных операционной системой, вы тем самым заявляете, что пользователи таких компьютеров вообще не смогут применять ваши программы.

Наиболее правильным представляется гибкий подход, при котором программа умеет применять все три возможных механизма захватов, в том числе одновременно, а то, какие из них следует использовать в конкретной ситуации, указывает пользователь с помощью настроек.

7.4.2. Создание дополнительного файла

Пожалуй, самый простой способ известить других «действующих лиц» о том, что в настоящий момент с файлом работаете вы — это создать рядом другой файл (чаще всего пустой); факт существования этого дополнительного файла рассматривается как извещение о захвате основного файла. Например, если вам нужно работать с файлом, который называется `foobar.dat`, то вполне логично воспользоваться именем `foobar.dat.lock`, расположенным в той же директории, для его захвата. Для этого, прежде чем начинать работу с `foobar.dat`, нужно попытаться создать файл `foobar.dat.lock`, воспользовавшись флажком `O_EXCL` (см. §5.3.3), т. е. потребовать от системы именно что создать новый файл, а если это невозможно — сообщить об ошибке. Если вызов `open` вернёт `-1` и переменная `errno` примет значение `EEXIST` — значит, файл `foobar.dat.lock` уже кем-то создан и нужно некоторое время подождать — например, одну секунду. По окончании работы с `foobar.dat` следует удалить файл `foobar.dat.lock` с помощью вызова `unlink` (см. §5.3.4). Выглядеть это всё будет примерно так:

```
for(;;) {
    int lckd;
    lckd = open("foobar.dat.lock", O_WRONLY|O_CREAT|O_TRUNC, 0666);
    if(lckd != -1) {
        close(lckd);
        break;  /* всё в порядке, мы установили захват */
    }
    if(errno != EEXIST) {
        /* Ошибка, НЕ связанная с тем, что кто-то сейчас работает
           с нашим файлом; например, у нас могло не хватить
           полномочий для создания foobar.dat.lock, или произошло
           что-то ещё.  Обрабатываем как обычную ошибку.  */
    }
    /* файл уже есть; следовательно, надо подождать */
    sleep(1);
}
/* работаем с foobar.dat */
/* ... */
unlink("foobar.dat.lock");    /* снимаем захват */
```

Здесь можно заметить аналогию с мьютексом: создание «захватного» файла[15] играет роль закрытия мьютекса, а его удаление — роль открытия мьютекса. Как и в случае с мьютексом, здесь крайне важна *атомарность* операций. Во всех современных системах семейства Unix операция «эксклюзивного» открытия файла атомарна, так что программа, построенная по приведённой схеме, будет работать корректно; но так было не всегда. Например, для всё тех же сетевых файловых систем поддержка флага `O_EXCL` появилась только начиная с протокола `NFSv3`; при её отсутствии ядро системы обычно эмулировало эффект от `O_EXCL` за два обращения к файловому серверу, так что никакой атомарности не обеспечивалось. В частности, ядра Linux начали полноценно (атомарно) поддерживать `O_EXCL` для сетевых файловых систем (естественно, при условии, что соответствующая поддержка есть в протоколе) только с версий `2.6.*`, то есть сравнительно недавно.

Для ситуаций, в которых атомарность открытия файла с флагом `O_EXCL` гарантировать невозможно, файловые захваты с использованием дополнительного файла всё ещё можно организовать, но уже по существенно более сложной схеме. Так, в man-странице по вызову open для Linux предлагается создать (обязательно на том же диске, где и предполагаемый «захватный» файл; проще всего делать это в одной и той же директории) временный файл с произвольным именем, таким, чтобы минимизировать вероятность совпадения выбранного имени с каким-то ещё; например, можно использовать в имени такого файла некую комбинацию из своего номера процесса и текущего времени. Этот файл не будет использоваться в качестве индикации захвата сам по себе; вместо этого при попытке захватить доступ мы потребуем от системы создать *жёсткую ссылку* на него с помощью вызова `link`; как раз эта ссылка будет играть роль индикатора действующего захвата.

[15] Оригиналом для нашего словосочетания «захватный файл» послужил английский термин *lock file*; как видим, трудности с переводом продолжаются.

Если вызов `link` пройдёт успешно — всё в порядке, мы создали жёсткую ссылку; но это ещё не конец истории. Поскольку обычно все эти пляски устраивают в расчёте на сетевые диски, придётся припомнить, что именно на сетевых дисках `link` может вернуть ошибку, но при этом всё равно создать требуемую ссылку: например, ядро нашей системы обратилось к файловому серверу, он создал жёсткую ссылку и послал нашему ядру ответ с сообщением об успехе, но ответ по каким-то причинам не дошёл, так что наше ядро сообщает нам об ошибке. Поэтому в случае, когда `link` вернул ошибку, нам придётся, не доверяя ему, проверить, не создалась ли ссылка. Для этого нужно применить к нашему временному файлу системный вызов `stat` и проверить, не стало ли количество жёстких ссылок на него равно двум; соответствующая информация будет содержаться в поле `st_nlink` структуры `struct stat`.

По окончании работы с разделяемым файлом нужно будет удалить жёсткую ссылку, тем самым сообщив другим программам, что мы сняли свой захват. Временный файл тоже можно удалить, а можно оставить, чтобы снова воспользоваться им через некоторое время; важно только не забыть удалить его перед завершением программы.

Подход с использованием «захватного» файла обладает несомненным достоинством — своей простотой. При этом подходе вообще не требуется никакой специальной поддержки со стороны операционной системы, и (с учётом всего вышесказанного) его можно применять даже на «заковыристых» сетевых файловых системах. С другой стороны, подход также не лишён и недостатков.

Прежде всего заметим, что программы, взаимоисключающие свой доступ к разделяемому файлу с использованием этого подхода, должны не только знать о существовании друг друга и о том, что используется именно подход с «захватным» файлом, но и знать, как этот «захватный» файл называется. Пока мы взаимодействуем только с другими экземплярами той же программы или только с программами, написанными тем же коллективом программистов, всё, скорее всего, будет в порядке — мы как-нибудь договоримся; но если будет нужно обеспечить работу с разделяемым файлом для программ от разных авторов или производителей, соглашение об имени «захватного» файла вполне может стать камнем преткновения. Между прочим, такие файлы даже не всегда создают в одной директории с файлом, доступ к которому захватывают; некоторые программы предпочитают использовать для этой цели системную директорию `/var/lock`.

Есть и ещё один недостаток «захватных» файлов, не менее очевидный: при некорректном завершении программы она может не успеть стереть ранее созданный «захватный» файл, так что остальные участники взаимодействия будут считать, что захват всё ещё установлен. Это тоже приходится учитывать, усложняя программы и делая их менее надёжными. Так, мы могли бы в вышеприведённом примере ограничить количество безуспешных попыток установления захвата, после чего считать, что тот, кто его установил, благополучно помер. Идея

в большинстве случаев работающая, но явно не слишком удачная: а вдруг он на самом деле не помер, а просто слишком долго «тормозит»? Некоторые программы записывают в «захватный» файл свой идентификатор процесса, а при обнаружении чужого захвата пытаются тем или иным способом связаться с процессом, установившим захват, и если это не удаётся — считают, что процесса, установившего захват, уже нет. Конечно, такой вариант обычно работает лишь для экземпляров одной и той же программы. Наконец, можно честно известить пользователя о проблеме и попросить его «разрулить» ситуацию, сказав ему что-то вроде «если в действительности никакие другие программы не работают с таким-то файлом, удалите такой-то («захватный») файл». У системных администраторов и прочих профессионалов такие ситуации трудностей не вызывают, а вот конечный пользователь вполне может от таких «заявочек» впасть в депрессию.

7.4.3. Системный вызов `flock`

Так называемые *BSD locks* (файловые захваты в стиле BSD) предполагают, что программы, работающие с разделяемым файлом, оповещают друг друга о входе в критическую секцию и о выходе из неё через системный вызов `flock`:

```
int flock(int fd, int operation);
```

Файл у нас должен быть уже открыт, его дескриптор передаётся в вызов первым параметром. Второй параметр может принимать одно из трёх основных значений: `LOCK_SH` — установить «разделяемый» захват (*shared lock*), `LOCK_EX` — установить «эксклюзивный» захват (*exclusive lock*) и `LOCK_UN` — убрать ранее установленный захват. На одном и том же файле могут быть одновременно установлены несколько «разделяемых» захватов, то есть если один процесс установил такой захват, то это не помешает сделать то же самое другому процессу. «Эксклюзивный» захват может быть установлен только один, причём он не может существовать одновременно ни с другим эксклюзивными, ни с разделяемыми захватами.

Если требуемый захват установить нельзя из-за захвата, установленного другим процессом, по умолчанию вызов `flock` блокирует вызвавший процесс до тех пор, пока конкурирующий захват не будет снят. Это поведение можно изменить, добавив ко второму параметру — как обычно, через операцию побитовго «или» — флаг `LOCK_NB` (*non-blocking lock*), то есть указать значение `LOCK_SH|LOCK_NB` или `LOCK_EX|LOCK_NB`. Совместно с `LOCK_UN` флаг использовать бессмысленно, эта операция и так никогда не блокируется.

Назначение двух разных видов захватов — эксклюзивного и разделяемого — становится понятно, если мы припомним, что читать

разделяемые данные можно сразу многим «действующим лицам», тогда как наличие одной *модифицирующей* критической секции, то есть такой, в которой предполагается изменение разделяемых данных, требует исключить на время её работы как запись, так и чтение данных для всех остальных участников взаимодействия. В §7.2.4 мы обсуждали одну из классических задач синхронизации — проблему читателей и писателей, решение которой направлено на то, чтобы «читатели» (то есть немодифицирующие критические секции) друг другу не мешали, а на время работы одного «писателя» (модифицирующей критической секции) все другие критические секции были исключены. Можно сказать, что задачу читателей и писателей вызов `flock` решает сам, то есть если такая задача перед нами встанет в применении к файлу, то `flock` даст нам уже готовое решение.

Здесь есть, впрочем, одна шероховатость. Обсуждая проблему читателей и писателей, мы отметили, что если читателей достаточно много, писатель может никогда не дождаться момента, когда в критической секции не будет ни одного читателя; это, как мы уже говорили, называется ***ресурсным голоданием***. Реализация вызова `flock` вполне могла бы решать эту проблему тоже — например, не выдавать никому новых разделяемых захватов, если есть хотя бы один процесс, заблокированный в ожидании эксклюзивного захвата; но, увы, авторы спецификации этим вопросом не озаботились, так что ничего подобного `flock` не делает. Продемонстрировать этот факт нам поможет простенькая демонстрационная программа:

```
   /* flock_starve.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <fcntl.h>

#ifdef SHARED
#define LOCK_MODE LOCK_SH
#define LOCK_NAME "shared"
#else
#define LOCK_MODE LOCK_EX
#define LOCK_NAME "exclusive"
#endif
#define LOCKED_SLEEP  1900000
#define UNLOCKED_SLEEP 100000

int main(int argc, char** argv)
{
    int i;
    if(argc < 2) {
        fprintf(stderr, "No argument specified\n");
        return 1;
```

```
    }
    int fd = open(argv[1], O_RDWR|O_CREAT, 0666);
    if(fd == -1) {
        perror(argv[1]);
        return 2;
    }
    for(i = 0;; i++) {
        int res;
        printf("%d Getting %s lock... ", i, LOCK_NAME);
        fflush(stdout);
        res = flock(fd, LOCK_MODE);
        printf("got it\nNow sleeping\n");
        usleep(LOCKED_SLEEP);
        printf("%d Releasing %s lock... ", i, LOCK_NAME);
        fflush(stdout);
        res = flock(fd, LOCK_UN);
        printf("done\nNow sleeping\n");
        usleep(UNLOCKED_SLEEP);
    }
}
```

Для демонстрации эффекта ресурсного голодания нам потребуются два исполняемых файла, полученных из этого исходного текста: первый будет откомпилирован с флагом `-D SHARED` и будет устанавливать разделяемый захват, второй мы откомпилируем без дополнительных флагов, так что захват он будет устанавливать эксклюзивный:

```
$ gcc -Wall -g -D SHARED flock_starve.c -o flock_sh
$ gcc -Wall -g flock_starve.c -o flock_ex
```

В обоих вариантах программа принимает аргументом командной строки имя файла, открывает его на чтение/запись с возможностью создания, после чего циклически устанавливает на нём захват соответствующего типа, спит 1,9 секунды, снимает захват, спит 0,1 секунды, после чего начинает цикл сначала. Запустив в разных окнах несколько экземпляров `flock_sh`, вы можете убедиться, что они друг другу совершенно не мешают. Убрав все экземпляры `flock_sh`, кроме одного, и запустив один экземпляр `flock_ex`, вы увидите, как они работают «в противофазе». Но вот если после этого запустить ещё один экземпляр `flock_sh`, скорее всего — а точнее, с вероятностью 90% — писатель `flock_ex` окажется заблокирован навсегда, поскольку у двух ваших читателей отсутствует пересечение временны́х промежутков нахождения вне критической секции. Для трёх читателей вероятность «голодной смерти» писателя окажется уже больше 99%, то есть вариант, что вас «случайно пронесёт», можно всерьёз не рассматривать; ну а запустить четырёх читателей так, чтобы писатель иногда мог проникнуть в критическую секцию, вам, скорее всего, просто не удастся (хотя тут есть одна хитрость: если сначала запустить писателя, а всех читателей запускать во

время его нахождения в критической секции, то все они станут работать синхронно; но это уже некое жульничество).

Проблему можно решить примерно так, как мы решили её в §7.2.4. Напомним, там мы ввели дополнительный мьютекс. При работе с файловыми захватами нам тоже никто не мешает завести дополнительный файл (скорее всего, пустой), на котором и читатели, и писатели будут устанавливать исключительный захват, но читатели будут это делать перед входом в критическую секцию, после чего тут же снимать захват, тогда как писатели будут этот захват устанавливать на всё время нахождения в критической секции.

Довольно нетривиально поведение вызова `flock`, если его применить к потоку ввода-вывода, на котором вы уже установили захват. В этом случае вызов может заменить эксклюзивный захват на разделяемый и наоборот, причём замена разделяемого захвата на эксклюзивный может оказаться невозможна прямо сейчас (если кто-то другой тоже обладает разделяемым захватом на том же файле), и вызов при этом заблокируется в ожидании, пока требуемая операция не окажется возможна. Следует отметить, что смена типа захвата не атомарна, и это прямо отражено в спецификации вызова: сначала снимается имеющийся захват, затем устанавливается новый, и между этими операциями может теоретически «вклиниться» другой процесс. Следовательно, нельзя — недопустимо! — производить смену типа захвата внутри критической секции, то есть до тех пор, пока не будут доведены до «логической точки» все вычисления, основанные на прочитанных данных, и не будут записаны все данные, которые должны быть записаны.

Теперь самое время обратить внимание на некоторые тонкости. Прежде всего подчеркнём, что **захват, установленный вызовом `flock`, связан с потоком ввода-вывода как объектом ядра операционной системы**. Напомним, что с одним и тем же объектом ядра может быть связано несколько файловых дескрипторов, причём как в разных процессах, так и в одном. При создании процесса с помощью `fork` новый процесс обладает всеми теми же дескрипторами потоков, но *объект потока ввода-вывода в ядре ОС при этом не копируется*; точно так же не копируются объекты ядра ни при выполнении вызовов `dup` и `dup2`, ни при дублировании дескриптора с помощью `fcntl` (командой `F_DUPFD`). С другой стороны, если ваш процесс откроет один файл дважды, в ядре появятся два независимых объекта, с каждым из которых может быть связан захват.

Эта особенность работы `flock` может проявиться довольно неожиданным образом. Представьте себе, что ваш процесс установил захват, после чего выполнил `fork`, а порождённый процесс затребовал снятие захвата. В этом случае захват будет снят с объекта потока ввода-вывода в ядре, то есть обладать захватом не будет ни порождённый процесс,

ни родительский — ведь объект потока у них общий. С другой стороны, можно представить, что вы открыли некий файл, установили на нём захват, после чего вызвали какую-нибудь библиотечную функцию, которая открыла тот же самый файл и тоже попыталась установить на нём захват. Если этот захват несовместим с вашим исходным (то есть хотя бы один из них эксклюзивный), ваша библиотечная функция благополучно заблокируется, так что вы попадёте в хорошо известную нам тупиковую ситуацию (deadlock) с самим собой.

Установленные на потоке ввода-вывода захваты сохраняют своё действие при замене выполняемой программы с помощью функций `exec*`. Это тоже следует учитывать: в большинстве случаев программа, которую вы запускаете, не имеет ни малейшего понятия о наличии захвата на каком-то из полученных ею «в наследство» дескрипторов, так что ей не придёт в голову этот захват снять. Как следствие, захват сохранит своё действие до тех пор, пока запущенная программа не закроет этот поток или не завершится; при невнимательном отношении к работе это может привести к тому, что другие участники взаимодействия будут вынуждены ждать не пойми чего.

Ещё раз напомним, что **захваты, устанавливаемые с помощью `flock`, не работают на сетевых дисках**.

С другой стороны, у захватов `flock` имеется одно несомненное преимущество перед рассмотренными в предыдущем параграфе «захватными файлами»: если ваша программа завершится аварийно, то все её дескрипторы автоматически закроются, и если с потоком, на котором она поставила захват, больше не связано дескрипторов (например, в порождённых `fork`'ом процессах), то захват исчезнет вместе с объектом потока ввода-вывода.

7.4.4. Файловые захваты POSIX

Файловые захваты, описанные в спецификации POSIX, как и рассмотренные в предыдущем параграфе захваты BSD, реализованы в ядре операционной системы; по своим свойствам эти два вида захватов достаточно сильно различаются. Прежде всего, захват POSIX устанавливается не на файл целиком, а на определённый его фрагмент.

Второй момент не столь очевиден. В отличие от захватов BSD, которые связаны с объектом потока ввода-вывода в ядре ОС, **захваты POSIX устанавливают отношение между файлом (дисковым объектом, а не объектом открытого потока) и конкретным процессом**. Так, если в вашем процессе есть несколько дескрипторов, связанных с одним и тем же файлом, в том числе если они открыты независимо друг от друга вызовом `open` (то есть связаны с разными объектами потоков ввода-вывода), то установку и снятие захвата можно производить через любой из них, это в любом случае будет один и

тот же захват. С другой стороны, **захваты POSIX не наследуются порождённым процессом (!)**, то есть если вы установили захват на тот или иной файл, после чего выполнили `fork`, система будет считать, что ваш родительский процесс обладает захватом этого файла, тогда как порождённый — не обладает.

Для работы с захватами POSIX можно использовать два разных интерфейса: системный вызов `fcntl` (см. §5.3.3) с командами `F_SETLK`, `F_SETLKW` и `F_GETLK`, а также функцию `lockf`:

```
int lockf(int fd, int cmd, int len);
```

В большинстве систем, в том числе в Linux и FreeBSD, эта функция реализована через системный вызов `fcntl`; при этом возможности системного вызова оказываются *шире* — в частности, `fcntl` позволяет устанавливать как захваты для чтения (аналог «разделяемых» захватов из предыдущего параграфа), так и захваты для записи, тогда как `lockf` устанавливает только эксклюзивные захваты.

Первый параметр функции `lockf` — это открытый файловый дескриптор, связанный с нужным файлом; файл должен быть открыт на запись — либо в режиме `O_WRONLY`, либо в режиме `O_RDWR` (это тоже отличает захваты POSIX от захватов BSD, где файл может быть открыт в произвольном режиме). Вторым параметром передаётся «команда», в роли которой может выступать одно из значений `F_LOCK` (установить захват; если прямо сейчас этого сделать нельзя, функция блокируется до тех пор, пока такая возможность не появится), `F_TLOCK` (*test and lock*: попытаться установить захват, но не блокироваться — немедленно вернуть управление), `F_UNLOCK` (снять захват) и `F_TEST` (не устанавливать никаких новых захватов, только проверить, возможно ли это). Третьим параметром функция получает длину захватываемого фрагмента файла, которая всегда отсчитывается от *текущей позиции* — той, с которой началась бы следующая операция чтения или записи. Можно в качестве этого параметра передать отрицательное число, тогда длина будет отсчитываться от текущей позиции *назад* — в направлении начала файла.

Функция возвращает `0` в случае успеха и `-1` в случае неудачи; в частности, при использовании команды `F_TEST` возвращается `0`, если указанный фрагмент файла можно захватить, и `-1`, если этому мешает захват, установленный кем-то ещё.

Работа через вызов `fcntl` выглядит несколько сложнее. Когда вторым параметром вызова указана одна из команд `F_SETLK`, `F_SETLKW` или `F_GETLK`, третьим параметром нужно передать адрес структуры типа `struct flock` (при том, что никакого отношения к функции `flock` всё это не имеет); выглядит это примерно так:

```
struct flock s1;
int res;
/* ... */
res = fcntl(fd, F_SETLKW, &s1);
```

Структура `struct flock` имеет следующие поля (в каком порядке они расположены — неизвестно): `l_type`, `l_whence`, `l_start`, `l_len` и `l_pid`. Поле `l_type` может принимать значения `F_RDLCK`, `F_WRLCK` или `F_UNLCK`, которые означают соответственно установку захвата на чтение (аналог «разделяемого» захвата), установку захвата на запись (аналог «эксклюзивного» захвата) и снятие захвата. Поле `l_start` задаёт позицию, с которой начинается нужный нам фрагмент файла. Число, находящееся в `l_start`, может отсчитываться от начала файла, от текущей позиции в файле или от конца файла в зависимости от того, какое значение будет помещено в `l_whence`; это может быть `SEEK_SET`, `SEEK_END` или `SEEK_CUR`, имеющие в точности такой же смысл, как для системного вызова `lseek` (см. §5.3.3). Поле `l_len` задаёт длину захватываемого фрагмента, и, как и для функции `lockf`, может быть отрицательным: в этом случае позиция «начала» рассматривается как конец фрагмента, а длина отсчитывается «влево» (в сторону начала файла). Кроме того, в поле `l_len` можно занести ноль, что будет означать захват от заданной точки начала до конца файла.

Поле `l_pid` используется только для команды `F_GETLK`, которая предназначена, чтобы узнать, имеется ли в настоящий момент на данном файле захват, конфликтующий с тем, который мы хотели бы установить. В него вызов записывает `pid` процесса, установившего этот захват. При использовании этой команды нужно предварительно заполнить все перечисленные поля структуры `struct flock`, описав захват, который мы хотели бы установить. Если такой захват может быть установлен, то вызов запишет значение `F_UNLCK` в поле `l_type` переданной ему структуры, а остальные поля оставит без изменений; если же в настоящий момент в системе имеется захват, установленный кем-то другим и конфликтующий с нашим, вызов заполнит поля `l_type` (в этом случае тут будет значение `F_RDLCK` или `F_WRLCK`), `l_whence`, `l_start`, `l_len` и `l_pid` в соответствии с тем, какой захват нам мешает и какой процесс его установил.

Команды `F_SETLK` и `F_SETLKW` предназначены для установки и снятия захвата; первая пытается выполнить запрошенную операцию, и если это не удалось — немедленно возвращает управление (естественно, вызов `fcntl` при этом вернёт `-1`); вторая же блокируется, если запрошенная операция не может быть выполнена прямо сейчас, то есть в системе имеется конфликтующий захват. Буква `W` в названии команды, как несложно догадаться, означает *wait*. Для обеих команд нужно заполнить поля `l_type`, `l_whence`, `l_start` и `l_len`.

Любопытно будет отметить, что снять захват можно с фрагмента, лишь частично совпадающего с тем, который ранее был захвачен. В этом случае остаток фрагмента продолжает быть захваченным, причём этих захваченных фрагментов может стать два: например, если вы захватите в файле `1000` байт, начиная с позиции `500`, а затем освободите `100` байт с позиции `700`, то у вас останется два захвата: с позиции `500` длиной `200` и с позиции `600` длиной `700`. Точно так же возможны слияния и разделения имеющихся захватов, если вы попытаетесь установить новый захват (другого типа) на фрагмент, который уже полностью или частично захвачен.

Отметим, что захваты POSIX могут работать с сетевыми дисками, если используются достаточно новые версии протокола и ядра системы; это выгодно отличает их от захватов BSD, которые на сетевых дисках просто не работают.

7.4.5. Некоторые проблемы файловых захватов

Файловые захваты POSIX имеют одну очень странную особенность: если ваш процесс закрывает с помощью `close` любой из дескрипторов, связанных с некоторым файлом, то при этом автоматически снимаются все захваты, которыми данный процесс обладал в отношении данного файла. В принципе можно понять, откуда взялась такая особенность: если процесс тем или иным способом завершается, все его захваты должны исчезнуть, но в отличие от захватов BSD, которые связаны с объектом потока ввода-вывода, захваты POSIX связаны с парой «процесс-файл»; по-видимому, создатели исходной реализации захватов POSIX решили не дублировать код ядра и «повесили» сброс всех захватов на закрытие потока, которое, как известно, происходит при завершении процесса автоматически. Проблема в том, что из-за этой особенности вы можете потерять все свои захваты совершенно неожиданно для себя — если какая-нибудь библиотека, функции которой вы вызываете, откроет какой-то из файлов, с которыми вы работаете в основной программе, и закроет его. Аналогичные проблемы возникнут, если в основной программе вы установите эксклюзивный захват (на запись), а некая подсистема вашей программы откроет тот же файл второй раз и попытается захватить его для чтения. В результате ваш эксклюзивный захват превратится в разделяемый. В некоторых случаях такие вещи довольно сложно предугадать, и особенно жёстко эта непредсказуемость проявляется в многопоточных программах; пожалуй, это ещё одна причина (хотя, надо признать, и не слишком серьёзная) отказаться от использования многопоточности.

Ещё одна проблема связана с обоими вариантами захватов, поддерживаемыми ядром (то есть как с захватами POSIX, так и с захватами BSD). Если некий файл доступен на запись только процессам определён-

ного пользователя или группы, а на чтение — либо всем пользователям системы, либо, по крайней мере, более широкому их кругу, нежели на чтение (например, записывать может один пользователь, а читать — некая группа), и при этом процессы, записывающие информацию в этот файл, используют взаимоисключение через файловые захваты, то (несколько неожиданно для нас) любой процесс, имеющий доступ на чтение, может нарушить работу всех процессов, осуществляющих запись и использующих для этого захваты — достаточно попросту установить разделяемый захват; больше того, доступа на чтение достаточно, чтобы установить даже эксклюзивный захват, если только это будет захват в стиле BSD.

Небезынтересен также вопрос о взаимоотношении различных типов файловых захватов. Как уже говорилось, функция `lockf` практически всегда реализована через `fcntl`, так что захваты, осуществляемые через эти два интерфейса — это, собственно говоря, одни и те же захваты. Иное дело — вызов `flock`. В современных версиях ОС Linux захваты, устанавливаемые через `flock` и `fcntl`, друг друга вообще не видят и никак друг на друга не влияют; но в той же FreeBSD это не так, там ядро отслеживает оба типа захватов в связке. Пикантности ситуации добавляет ещё и то, что в некоторых системах функция `flock` реализована через `fcntl`, как и `lockf`, причём в ОС Linux достаточно старых версий дело тоже обстояло именно так.

Если рассматривать имеющуюся картину целиком, получится, что относительно использования функций `flock`, `lockf` и `fcntl` (в части файловых захватов) вообще невозможно указать никаких способов их использования, которые бы были заведомо корректны. Поэтому мы просто повторим рекомендацию, которую вы уже видели: избегайте появления разделяемого доступа к файлам, покуда это хоть как-то возможно.

Часть 8

Ядро системы: взгляд за кулисы

Читатель, скорее всего, уже представляет, зачем нужна операционная система и каковы её роль и место в происходящем. Мы активно использовали системные вызовы, чтобы обращаться к системе за услугами и управлять объектами ядра. В этой части книги мы попытаемся показать отдельные аспекты того, что происходит внутри ядра и обычно остаётся за кадром для всех, кроме программистов, создающих само ядро и его отдельные компоненты — драйверы устройств и другие «ядерные» модули.

Пытаться самостоятельно писать модули ядра мы не будем — это довольно специфическая область, для которой высок порог вхождения. Книги, посвящённые этому, вынужденно обрушивают на читателя поток разнообразных сведений, без которых невозможно обойтись при работе внутри ядра — чтобы в конце позволить написать явно искусственный и не имеющий никакой, даже потенциальной ценности модуль вроде убогого драйвера клавиатуры или какого-нибудь совершенно бесполезного виртуального устройства. Что самое неприятное здесь — это то, что все полученные сведения применимы обычно только к одной конкретной версии одного конкретного ядра; с выходом следующей версии того же ядра значительную часть полученных сведений приходится выбросить.

Программирование внутри ядра ОС можно назвать одним из самых увлекательных, но и самых сложных аспектов системного программирования. Если вы захотите этим заниматься, к вашим услугам масса специальной литературы и сайтов в Интернете; надо сказать, что здесь будет лучше воспользоваться англоязычными источниками, поскольку к моменту перевода таких текстов на русский язык они успевают изрядно устареть.

С другой стороны, в функционировании ядра системы остаётся целый ряд интересных аспектов, которые желательно представлять хотя бы на качественном уровне, то есть понимать в общих чертах, как всё это устроено. Этому мы и посвятим заключительную часть третьего тома книги.

8.1. Основные принципы работы ОС

Прежде чем двигаться дальше, кратко напомним основы. Ядро операционной системы загружается в память раньше других программ, благодаря чему выполняется в *привилегированном режиме* центрального процессора; оно запускает пользовательские программы в виде так называемых *процессов*, которые работают в *ограниченном режиме*. До передачи управления процессам ядро настраивает *обработчики* всех «особых» событий — аппаратных прерываний, исключений/ловушек (внутренних прерываний) и системных вызовов (программных прерываний) так, чтобы при возникновении любого из этих событий управление снова передавалось коду ядра, а поскольку режим процессора переключается с ограниченного обратно в привилегированный только при наступлении таких событий, всё это гарантирует, что только код ядра и будет выполняться на компьютере в привилегированном режиме ЦП. Ограниченный режим ЦП позволяет процессу только преобразовывать данные в отведённой ему памяти — и больше ничего; для всего остального, в том числе чтобы выдать какое-нибудь сообщение и даже чтобы просто завершить выполнение, процесс вынужден обращаться к ядру через механизм системных вызовов.

На всякий случай повторим ещё раз, что **никакой код никакого пользовательского процесса ни при каких условиях не может быть исполнен в привилегированном режиме процессора**. Часто в литературе можно встретить фразы вроде «процесс выполняется в режиме ядра», и мы тоже будем использовать это выражение; но пусть вас не обманывает такая терминология — на самом деле речь идёт о выполнении машинного кода *самого ядра*, а не процесса, просто этот код выполняется *в рамках процесса как единицы планирования* — то есть планировщик выделяет ему кванты времени так же, как и другим процессам, а сам этот код может «заснуть» (заблокироваться) в ожидании наступления какого-нибудь события. Такое «исполнение в режиме ядра» происходит, например, при обработке системного вызова.

Ядро операционной системы берёт на себя две основные функции: *управление пользовательскими задачами* и *управление внешними устройствами*. Первое включает запуск и останов процессов, планирование времени центрального процессора, распределение оперативной памяти, организацию взаимодействия между процессами и разграниче-

ние их полномочий; второе сводится к *абстрагированию* от особенностей конкретных устройств и *координации* запросов к ним (см. §5.1).

Отметим, что ядро не обязано быть монолитной программой. В некоторых операционных системах ядро представляет собой набор взаимодействующих программ (архитектуры с микроядром и экзоядром); почти во всех современных системах в ядро во время работы можно добавлять дополнительные модули. Некоторые свои части ядро может оформить в виде процессов, работающих наравне с пользовательскими задачами в ограниченном режиме.

8.1.1. Ядро как обработчик запросов

Интересно отметить, что, загрузившись в память после старта системы, **ядро ничего не делает по своей инициативе**. На первый взгляд это может показаться странным, но всё становится на свои места, если вспомнить, что компьютер в основном предназначен для решения проблем *конечных пользователей*, а эти проблемы решают прикладные программы, которые создаются и работают в виде пользовательских задач. При отсутствии в системе активных пользовательских задач ядру просто оказывается нечего делать; в этом случае оно может *остановить* центральный процессор в ожидании, пока не произойдёт что-нибудь интересное. В таких случаях говорят, что система перешла в ***состояние покоя*** (англ. *idle*). Отметим, что состояние покоя возможно и осмысленно, когда задачи в системе есть, но все они находятся в блокировке, то есть ждут наступления каких-то событий, чтобы после этого продолжить работу. Если в системе вообще исчезнут пользовательские задачи, ядру не останется ничего иного, кроме как прекратить работу — либо отправить компьютер на перезагрузку, либо вообще «повиснуть» за неимением лучших идей. Сообщение об этом — лаконичное «`System halted`» — часто снится системным администраторам в ночных кошмарах.

Пользовательским задачам для работы часто требуется что-то такое, что может сделать только ядро (чаще всего это ввод-вывод информации), и они через системные вызовы обращаются к ядру за услугами. В этом случае ядро, естественно, начинает активно работать, но, заметим, оно при этом работает не по своей инициативе, а в ответ на запросы пользовательских задач. Обращаясь к контроллерам устройств, ядро может получать ответы от них не сразу, а по мере готовности — через механизм аппаратных прерываний; через них же ядро узнаёт о наступлении внешнего события, требующего каких-то действий — например, о приходе пакета данных по локальной сети, о нажатии клавиши на клавиатуре и т. п., и в этом случае ядро тоже включается в работу, но инициатива и здесь, как видим, исходит не от него.

Во втором томе в части, посвящённой программированию на языке ассемблера, мы уже обсуждали термин «прерывание» и связанную с

ним путаницу (см. т. 2, §3.6.3). Напомним, что по инициативе внешних устройств у нас могут происходить собственно ***прерывания***; в результате ошибочных действий выполняющейся программы процессор может генерировать ***исключения***, называемые также ***внутренними прерываниями***. В обоих случаях режим центрального процессора становится привилегированным, а управление передаётся на процедуру-обработчик, адрес которой настроен заранее. Сама такая процедура, естественно, является частью ядра операционной системы. Частным случаем внутреннего прерывания можно считать ***программное прерывание***, которое задача инициирует не по ошибке, а намеренно, чтобы отдать управление ядру для выполнения ***системного вызова***. Мы также отмечали, что терминология с тремя разными видами прерываний не совсем удачна, поскольку реально что-то *прерывают* только «настоящие» прерывания, они же ***аппаратные*** или ***внешние***, а остальные — внутренние и программные — на самом деле ничего и никого не прерывают; при описании архитектур, отличных от Intel, термин «прерывание» обычно используется только для обозначения аппаратных прерываний, внутренние прерывания называют исключениями, а между системным вызовом и его реализацией не делают различия, то есть вместо фразы «программа выполнила внутреннее прерывание для осуществления системного вызова» говорят просто «программа выполнила системный вызов».

Вне зависимости от того, какой из стилей терминологии использовать, ядро операционной системы после окончания загрузки может получить управление только по запросу внешнего устройства, по запросу пользовательской задачи или в результате ошибочных действий пользовательской задачи. В принципе, это мы тоже обсуждали во втором томе, но не лишним будет напомнить ещё раз, что обработка аппаратных прерываний на уровне ядра существенно отличается от обработки событий, ставших результатом действий пользовательской задачи — неважно, случайных или намеренных. При получении управления по инициативе пользовательской задачи ядро выполняет свой обработчик в контексте этой задачи, то есть планировщик продолжает рассматривать такую задачу как единицу планирования; задача может заблокироваться, заснуть, отдать управление другим частям ядра, получить управление обратно и т. д.; всё это никак не влияет на функционирование системы в целом.

С аппаратными прерываниями всё гораздо сложнее. Исключения и системные вызовы — это своего рода продолжение выполнения задачи, они возникают в ходе обычного выполнения машинных команд; аппаратные прерывания, напротив, возникают в произвольные моменты времени, в том числе и в самые неподходящие. Уже на аппаратном уровне процессору приходится это учитывать: если запрос на прерывание пришёл в середине выполнения очередной машинной инструкции, то эту инструкцию приходится либо доводить до конца, либо откаты-

вать к началу. Когда обработчик аппаратного прерывания получает управление, никакого контекста пользовательской задачи у него нет, ведь прерывание может возникнуть в том числе и тогда, когда никакая пользовательская задача не выполнялась — либо когда работало ядро, либо когда система вообще находилась в состоянии покоя, не выполняя никаких программ.

Больше того, ничто не исключает возможности возникновения следующего прерывания сразу после предыдущего — например, запрос прерывания может прийти от другого устройства. Как правило, обработчик прерывания к этому не готов, поскольку он мог ещё не успеть выяснить, что́ же произошло и стало причиной прерывания, и при возникновении нового прерывания эта информация окажется потеряна; поэтому центральный процессор в момент получения прерывания *блокирует* другие прерывания. В этом режиме поступающие запросы прерываний накапливаются, но никакого эффекта не имеют; процессор как бы *откладывает* их обработку до тех пор, пока обработчик текущего прерывания не снимет блокировку. Надо отметить, что возможности хранения отложенных запросов у процессора весьма ограниченны: как правило, он может «помнить» не более чем об одном отложенном прерывании от каждой из нескольких групп внешних устройств. Если в режиме блокировки прерываний система пробудет достаточно долго, это может привести к потере запросов прерываний и в конечном итоге к сбоям в работе системы. Поэтому обработчик обязан как можно скорее достичь такого состояния, при котором вся информация о поступившем прерывании надлежащим образом сохранена, и разрешить (разблокировать) прерывания.

Из всех случаев обработки аппаратных прерываний можно выделить важный вариант, при котором обработчик успевает сделать всё необходимое за столь короткий период времени, что в разрешении аппаратных прерываний необходимости не возникает: управление практически сразу возвращается той программе, выполнение которой было прервано, при этом блокировка прерываний автоматически снимается. Самый простой пример такой ситуации — это прерывание таймера, поступившее, когда по тем или иным причинам вызывать планировщик не нужно: если нет других задач, готовых к выполнению, или если квант времени текущей задачи ещё не истёк. Обычно в ядре для таких ситуаций предусматриваются специальные флажки и счётчики, проверить и изменить которые можно за несколько десятков тактов центрального процессора, не сохраняя нигде большинство регистров; если обработчик прерывания таймера видит, что больше от него ничего не требуется, он сразу же завершается. **Такая обработка аппаратного прерывания, при которой все необходимые действия выполняются достаточно быстро, чтобы всё можно было сделать при заблокированных прерываниях, называется *коротким прерыванием*.**

Если аппаратное прерывание требует от ядра более сложных действий, картина резко усложняется. Прежде чем разрешить другие прерывания, обработчик должен позаботиться о том, чтобы неожиданный вызов другого (или, возможно, того же самого) обработчика ничего не испортил. Кроме того, обработчик прерывания, работая вне контекста какой-либо пользовательской задачи, не является единицей планирования, что накладывает серьёзные ограничения на ассортимент используемых средств. В частности, обработчик прерывания не может «заснуть» (заблокироваться), поскольку разблокировать его будет некому.

Если бы центральный процессор в системе был один, обработчику прерывания было бы вообще не нужно ничего ждать, да и *нельзя*, ведь при запрещённых прерываниях ничего «внешнего» по отношению к активной программе, собственно говоря, произойти не может; но в современных условиях процессоров в системе обычно больше одного, ведь с программной точки зрения «многоядерные»[1] процессоры представляют собой несколько независимых центральных процессоров, выполненных в виде одной микросхемы. В таких условиях неизбежно возникают *критические секции*[2] по данным, к которым могут обратиться компоненты ядра, выполняемые одновременно на разных процессорах, и эти критические секции требуют взаимоисключения; но поскольку блокировка здесь невозможна (ведь, напомним ещё раз, обработчик прерывания не является единицей планирования, в отличие от процесса), приходится применять активное ожидание, вхолостую расходуя процессорное время.

Ядро операционной системы обычно реализует целый ассортимент средств взаимоисключения для разных случаев; в частности, в ядре Linux их больше десятка. Программисту приходится тщательно следить, чтобы применяемый механизм соответствовал контексту; так, средства взимоисключения, предназначенные для обработчиков аппаратных прерываний, нельзя применять в контексте процесса, и наоборот.

Итогом обработки аппаратного прерывания может стать констатация необходимости смены текущей задачи, например, в силу истечения отведённого ей кванта времени, либо если событие, о котором сигнализирует полученное прерывание, становится причиной разблокировки задачи, имеющей приоритет более высокий, чем текущая. Может случиться и так, что система находилась в состоянии покоя, но происшедшее собы-

[1] Здесь имеется очередная терминологическая сложность. Студенты, не понявшие в должной мере, о чём идёт речь, иногда путаются со словом «ядро», которое используется в русском языке для обозначения, с одной стороны, основной программы операционной системы, а с другой — для отдельных независимых процессоров в составе одной «многоядерной» микросхемы. В английском такой путаницы нет: ядро операционной системы называется *kernel*, тогда как часть микросхемы называется *core*. Вообще-то слово *core* следовало бы переводить как «сердечник», а не «ядро», но словосочетание «многосердечниковый процессор», что называется, не звучит.

[2] Если словосочетание «критическая секция» вызывает какие-то сложности с пониманием, самое время вернуться к предыдущей части и перечитать § 7.1.2.

тие разблокировало какую-то задачу из ранее заблокированных. В этом случае всё сравнительно просто: ядро сохраняет контекст текущей задачи, если таковая была, затем восстанавливает контекст задачи, которую нужно поставить на выполнение, и дальнейшая подготовка этой задачи к выполнению происходит уже в её контексте.

В сравнительно редких случаях при получении прерывания оказывается нужно выполнить какие-то действия, которые невозможно отнести к выполнению той или иной задачи, но при этом они слишком сложны, чтобы их можно было выполнить вне контекста задачи. Для таких случаев ядра некоторых систем предусматривают специальные ***псевдопроцессы***, порождаемые ядром и выполняющиеся исключительно в режиме ядра, но имеющие свой контекст в качестве единиц планирования. Обработчик прерывания может поручить выполнение оставшейся части работы такому псевдопроцессу.

В частности, авторы ядра Linux используют для разных стадий реакции на прерывание термины ***верхняя половина*** и ***нижняя половина*** (англ. *top half*, *bottom half*). Под верхней половиной понимается обработчик аппаратного прерывания как таковой, то есть та часть кода ядра, на которую процессор передаёт управление при наступлении прерывания и которая выполняется при заблокированных аппаратных прерываниях. Очевидно, она должна завершиться как можно скорее; основную работу следует перепоручить нижней половине.

Для организации нижней половины обработки прерывания Linux предусматривает две принципиально различные сущности: ***тасклеты*** (*tasklets*) и «рабочие очереди» (*workqueues*). Тасклет представляет собой сравнительно короткую функцию, выполнение которой предполагается уже без блокировки аппаратных прерываний, то есть не столь жёстко критично по времени; с другой стороны, тасклет должен выполниться «за один приём», поскольку единицей планирования он не является и «заснуть», как следствие, не может. Обработчик аппаратного прерывания, решив возложить часть работы на тасклет, сообщает об этом планировщику, после чего завершается; планировщик выполняет тасклет всегда на том же процессоре, на котором его запланировали, так что он точно не может начать выполняться раньше, чем завершится запланировавший его обработчик. Выполнение тасклета может начаться немедленно, как только обработчик прерывания вернёт управление, если процессор, на котором это происходит, свободен; если же на процессоре выполняется пользовательская задача, то тасклет будет выполнен после прихода следующего прерывания таймера, временно вытеснив задачу с процессора.

Рабочая очередь отличается от тасклета тем, что при выполнении «работы», поставленной в очередь, сама эта «работа» обладает всеми преимуществами единицы планирования, то есть ей, как обычному процессу, выделяются кванты времени, она может «заснуть» — заблокироваться в ожидании события, и т. д. Рабочей очереди, в отличие от тасклета, можно поручить сколь угодно длинное задание; с другой стороны, никто не гарантирует (опять же в отличие от тасклета), что выполнение задания, поставленного в очередь, начнётся не позже какого-то момента. С точки зрения планировщика рабочая очередь — это почти обыкновенный процесс, у него даже есть свой `pid`; в выдаче команды `ps ax`

рабочие очереди хорошо заметны — их названия выдаются заключёнными в квадратные скобки, примерно так:

```
  8 ?        S<     0:00 [cpuset]
  9 ?        S<     0:00 [khelper]
 10 ?        S      0:00 [kdevtmpfs]
```

Как и в случае с тасклетом, обработчик прерывания («верхняя половина») может потребовать выполнить очередную «работу» в рамках заданной рабочей очереди, после чего с чистой совестью вернуть управление; всё, что осталось, сделает за него «работа».

Вернёмся к обсуждению пассивной роли ядра в системе; как мы отметили выше, если в системе нет пользовательских задач, ядру будет нечего делать, но если оно не будет ничего делать, то откуда возьмутся задачи? Выйти из порочного круга позволяет введение специфической пользовательской задачи, которая выполняется в течение всего времени работы системы, начиная от момента её загрузки и заканчивая моментом останова или перезагрузки. В ОС Unix такая задача называется ***процессом init***, который всегда имеет номер 1. По сути это обыкновенная программа, написанная, как правило, на языке Си и не имеющая принципиальных отличий от других программ. Единственная существенная особенность `init` состоит в том, что ему приходится выполнять обязанности предка для тех процессов, предки которых уже завершились; мы упоминали эту его особую роль при обсуждении процессов-зомби и системных вызовов семейства `wait` в § 5.4.6.

Настройки самого ядра или (чаще) программы-загрузчика определяют, какую программу ядро должно загрузить и запустить в качестве `init`; последним действием в ходе загрузки операционной системы ядро запускает `init`, после чего считает загрузку завершённой и более ничего не делает иначе как в ответ на запросы. Первоначально такие запросы поступают от самого `init`: он должен прочитать свои конфигурационные файлы и решить, какие ещё программы запустить. В итоге система оказывается «населена» пользовательскими задачами, которые запускают друг друга.

Следует отметить, что завершение процесса `init` означает завершение работы системы; ядро при этом считает, что от него больше ничего не требуется, и в зависимости от настроек либо останавливается, либо (реже) отправляет компьютер на перезагрузку.

8.1.2. Загрузка и жизненный цикл ОС UNIX

Сразу после включения компьютера центральному процессору нужно начать что-то делать, в противном случае ничего никогда не произойдёт и компьютер останется для нас бесполезен. Как мы знаем, процессор умеет только одно — *выполнять программы*; но, с другой стороны, процессор не знает и не может знать ничего о внешних устройствах — для

работы с ними нужны ***драйверы***, которые тоже представляют собой программы, причём довольно сложные.

Всё, что процессор умеет сам — это общаться через шину с оперативной памятью, но мы знаем, что оперативная память не сохраняет информацию при выключении питания. Состояние оперативной памяти сразу после включения питания *не определено*, то есть в каждой ячейке памяти может оказаться совершенно произвольное число. Очевидно, что никакой программы там быть не может — ей неоткуда взяться.

Итак, процессор получил электропитание и «ожил», оперативная память содержит заведомый мусор, про внешние устройства процессор ничего не знает; чтó же, в таком случае, ему выполнять?

Специально на этот случай в конструкции компьютера предусмотрено так называемое ***постоянное запоминающее устройство*** (ПЗУ; англ. *ROM, read-only memory*). Микросхемы, составляющие ПЗУ, с точки зрения шины и центрального процессора «выглядят» в точности как обыкновенная оперативная память, состоящая из ячеек, за исключением того, что они поддерживают только операцию чтения, а попытки записи в свои ячейки молча игнорируют. По своей конструкции ПЗУ резко отличается от оперативной памяти (ОЗУ): каждая из ячеек постоянной памяти содержит одно фиксированное, раз и навсегда определённое значение, которое и выдаёт в шину в ответ на запрос чтения.

В разное время для создания ячеек ПЗУ использовались разные технологии. Самая старая и «кондовая» из них состояла в том, что каждая ячейка выполнялась в виде восьми микроскопических перемычек; замкнутая перемычка обозначала единицу в соответствующем разряде ячейки. При записи информации в микросхему с помощью специального устройства, называемого *программатором*, некоторые из перемычек попросту выжигались — на них подавалось высокое напряжение, под воздействием которого тонкий медный проводок плавился и переставал существовать; выжженные перемычки, как можно догадаться, обозначали ноль. Микросхемы такого типа назывались ППЗУ — программируемое постоянное запоминающее устройство (англ. *PROM — programmable read-only memory*). Очевидно, что такая микросхема была одноразовой: возможность изменить значения, записанные в её ячейки, физически отсутствовала, при любой ошибке микросхему приходилось выбрасывать.

Позже появилась более «аккуратная», хотя и дорогая технология, в которой вместо перемычек использовались специальным образом оформленные полевые транзисторы. Состояние такого транзистора исходно соответствовало «единице», но, подав на него достаточно сильное напряжение, можно было перевести его в состояние «ноль» — практически так же, как и с микросхемами типа PROM, только напряжение требовалось более скромное; в этом состоянии транзистор оставался вне зависимости от наличия или отсутствия питания. Все транзисторы та-

кой микросхемы можно было вернуть в исходное состояние, подвергнув её облучению сильным источником ультрафиолета. Такие микросхемы получили название СППЗУ — стираемое программируемое постоянное запоминающее устройство (англ. *EPROM — erasable programmable read-only memory*).

Технология, используемая в наше время, ультрафиолетовой лампы для стирания уже не требует; соответствующие микросхемы по-английски называются *EEPROM — electrically erasable programmable read-only memory*; изредка можно встретить также русскую аббревиатуру ЭСППЗУ. Программатор такой памяти может быть встроен в общую схему компьютера или другого устройства, использующего EEPROM, что позволяет изменить содержимое ПЗУ, не имея для этого специальных устройств — например, в домашних условиях сменить версию программы, «зашитой» в ПЗУ.

Программа, которую процессор должен начать выполнять после включения питания, размещается в микросхемах ПЗУ; именно эта программа получает управление самой первой — как только процессор начнёт работу. Адрес её начала (точки входа) для конкретного типа процессора всегда один и тот же; например, i386 начинает выполнять инструкции с адреса $\texttt{FFFFFFF0}_{16}$, причём процессор стартует в так называемом «реальном режиме» (*real mode*), совместимом с предыдущими процессорами линейки. В этом режиме процессор выполняет команды точно так же, как это делали его 16-битные «предки», отсутствует защита памяти и деление команд на привилегированные и непривилегированные. Переключение процессора в «защищённый» режим, в котором работают современные операционные системы, производится гораздо позже, обычно это делает уже ядро операционной системы.

В силу исторических причин программу, зашитую в ПЗУ для выполнения при старте процессора, часто называют BIOS (*basic input-output system*). После выполнения некоторых действий по проверке оборудования эта программа определяет загрузочное устройство (загрузочный диск), считывает в память первый сектор этого диска, называемый также ***загрузочным сектором***, и передаёт управление машинному коду, прочитанному из загрузочного сектора.

Поскольку размер загрузочного сектора сравнительно невелик (512 байт, причём начальный сектор может содержать ещё данные помимо загрузочного кода), программа, записанная в загрузочный сектор, не может выполнить никаких сложных действий, она для этого слишком коротка. Поэтому её роль заключается в загрузке в память более сложной программы, записанной на диске в специальных областях; эти области различны в разных операционных системах и для разных версий загрузочных программ. Новая программа называется ***загрузчиком операционной системы*** и может быть уже сравнительно сложной. Так, некоторые загрузчики имеют собственную командную строку, поз-

воляющую выбрать, какой раздел считать корневым, из какого файла загружать ядро и т. п., при этом возможен даже просмотр каталогов на дисках. Классический загрузчик ОС Linux (LILO) не столь гибок в возможностях: он обладает способностью загружать альтернативные операционные системы, выбирать одно из предопределённых ядер для загрузки и передавать ядру параметры, но формат файловой системы не понимает и просматривать диски не позволяет. Ядро он загружает из фиксированного набора физических секторов диска. Какой конкретно загрузчик будет использоваться — зависит от дистрибутива; LILO в последние годы встречается довольно редко.

Загрузчик загружает в память выбранное ядро и передает управление его коду. Ядро инициализирует свои подсистемы, в том числе драйверы устройств, что включает, естественно, проверку наличия в системе соответствующего оборудования; при отсутствии нужного устройства драйвер обычно отказывается инициализироваться, после чего ядро удаляет его из памяти. Затем ядро монтирует файловую систему корневого дискового устройства в качестве корневого каталога системы. Обычно корневой каталог монтируется в режиме «только для чтения». После того как ядро готово к работе и смонтировало корневое устройство, оно формирует процесс с номером `0`. Этот процесс существует только на этапе загрузки, и единственная его роль — создать с помощью `fork` процесс с номером `1`. После этого нулевой процесс прекращает существование, так что в системе с этого момента есть только процессы, созданные с помощью вызова `fork`.

Процесс номер `1` выполняет вызов `execve`, чтобы загрузить в память программу `init`. Обычно это файл `/sbin/init`; ясно, что он должен находиться на корневом дисковом устройстве. Это, как мы уже говорили, обычная программа, написанная на Си или (теоретически) на любом другом компилируемом языке программирования. Как правило, загрузчик позволяет передать ядру специальный параметр, указывающий, какую программу следует загрузить вместо `init`; например, так можно запустить интерпретатор командной строки для выполнения действий по обслуживанию системы.

Процесс `init` работает в течение всего времени работы ОС; его завершение влечёт останов системы. Именно процесс `init` выполняет проверку дисков, перемонтирует корневой раздел в режим «чтение/запись» и монтирует остальные файловые системы. Затем процесс `init` должен инициализировать подсистемы ОС, например, сконфигурировать интерфейсы работы с локальной сетью, запустить системные процессы-демоны и, наконец, запустить на имеющихся терминальных линиях программы `getty`, отвечающие за запрос входного имени и пароля пользователей. Программа `getty` создаёт сеанс, связанный с её терминалом, и после успешной аутентификации пользователя запускает с помощью `exec` интерпретатор командной строки. Когда процесс

интерпретатора завершается, программа `init` снова запускает `getty` на освободившейся терминальной линии.

Функциональность программы `init`, как видим, оказывается достаточно сложной. В связи с этим выполнение большинства действий, связанных с инициализацией системы, обычно возлагается на ***скрипты системной инициализации*** (в зависимости от системы это может быть файл `/etc/rc`, `/etc/rc.d/rc` и т. п.). Программе `init` тогда достаточно указать через конфигурационный файл или в качестве параметра компиляции, где находится соответствующий скрипт. В классическом варианте скрипты обычно писали на языке стандартного командного интерпретатора (Bourne Shell).

Процедура корректного останова системы для перезагрузки или выключения компьютера также возлагается на `init`, который запускает для этого специально предназначенный скрипт. В этом скрипте содержатся команды по уничтожению работающих процессов (сначала с помощью сигнала `SIGTERM`, затем, после паузы, — сигналом `SIGKILL`), размонтирования всех файловых систем, кроме корневой (её размонтировать невозможно), перевод корневой системы в режим «только чтение», синхронизация корневой файловой системы, т. е. запись недозаписанных данных из буферов. После этого выполняется собственно останов — прекращение работы ядра.

8.1.3. Эмуляция физического компьютера

Как мы знаем, при попытке процесса выполнить некорректную инструкцию, в том числе привилегированную, возникает исключение («внутреннее прерывание»), в результате которого управление отдаётся ядру системы. Это позволяет сымитировать действия физической машины таким образом, чтобы у программы, работающей в рамках пользовательского процесса, «создалось впечатление», что эта программа работает в привилегированном режиме на машине, на которой никого, кроме неё, нет. Действия, которые на физической машине осуществлял бы процессор в ответ на привилегированную команду, в режиме эмуляции выполняют обработчики прерываний по некорректной инструкции и нарушению защиты памяти.

В режиме такой эмуляции можно запустить в виде пользовательского процесса ядро другой операционной системы или даже второй экземпляр той же самой. Впервые такая эмуляция была реализована на мейнфреймах IBM/360 операционной системой CP/CMS в конце 1960-х годов. Под управлением этой системы можно было запустить несколько операционных систем OS/360, причём каждая из них была уверена, что весь мейнфрейм находится в её полном распоряжении. Под CP/CMS можно было даже загрузить в режиме эмуляции её саму. Бóльшую известность получила вышедшая в 1972 году система VM/370;

в литературе часто именно ей приписывают первенство в этом классе систем.

Подобные системы обычно называют ***гипервизорами***. Гипервизор может сам играть роль операционной системы, то есть загружаться на компьютере первым, а затем запускать «настоящие» операционные системы в качестве своих «задач»; кроме того, гипервизор может быть выполнен как подсистема обычной операционной системы — в этом случае такая система (работающая на физическом компьютере) называется *хостовой*[3], а системы, запущенные в виртуальных машинах под управлением гипервизора — *гостевыми*. Среди используемых в наши дни можно назвать «отдельно стоящий» гипервизор Xen[4], а также Kernel Virtual Machine (KVM) для Linux и *bhyve*[5] для FreeBSD, которые позволяют этим системам выполнять роль хостовых.

Технологию виртуализации, при которой гипервизор перехватывает исключения, возникающие при попытке исполнить привилегированные команды, и эмулирует их, так и называют — *trap and emulate*. При всей очевидности этого подхода отнюдь не любой процессор позволяет эту технологию использовать, и i386, который мы изучали во втором томе, как раз из тех, для которых такой вариант не годится. В частности, некоторые команды i386 не входят в число непривилегированных, так что их выполнение никаких исключений не вызывает, но работают они при этом по-разному в зависимости от установленного режима процессора. Чаще всего в качестве такого примера приводят команду `POPF`, которая извлекает из стека значение и записывает его в регистр флагов; её обычно применяют в паре с командой `PUSHF`, которая, наоборот, помещает значение регистра флагов в стек. Проблема в том, что некоторые флаги (но не все) считаются привилегированными; например, флаг `IF` (*interrupt flag*) указывает, разрешены ли в данный момент аппаратные прерывания, и изменять этот флаг в ограниченном режиме, естественно, нельзя. Команда `POPF`, будучи исполнена в привилегированном режиме, устанавливает *все* имеющиеся флаги в соответствии с извлечённым из стека значением, а в ограниченном режиме затрагивает только флаги, изменение которых допустимо для пользовательской задачи, «втихую» игнорируя при этом привилегированные флаги. Кроме того, анализируя значения в сегментных регистрах, выполняющаяся на i386 программа может узнать, в каком кольце защиты она работает, то есть, грубо говоря, ядро «гостевой» операционной системы имеет возможность понять, что оно в настоящий момент работает не на настоящей машине, а на виртуальной.

[3] С английским термином *host* мы уже встречались при обсуждении компьютерных сетей, см. сноску на стр. 171; напомним, что это слово переводится как *хозяин, принимающий гостей*.

[4] Как ни странно, это название произносится примерно как *зен*.

[5] Читается примерно как *бихáйв*.

Подход с эмуляцией привилегированных команд осложняется ещё и тем, что работа процессора может напрямую зависеть от структур данных, находящихся в обычной памяти. В основном это касается MMU, той части процессора, которая отвечает за преобразование виртуальных адресов в физические; немного позже мы увидим, что в этих преобразованиях участвует информация, которую операционная система должна сформировать в обычной оперативной памяти — так называемые страничные таблицы и дескрипторы сегментов. Изменение содержания этих страниц при работе на настоящем процессоре должно немедленно повлиять на то, к каким физическим адресам будет обращаться процессор; но когда «гостевая» система что-то записывает в свои таблицы, выглядит это как обычное обращение к памяти, то есть привилегированным действием не является. Для отслеживания таких обращений «хостовая» система вынуждена прибегать к хитростям — например, частично изменять машинный код «гостевого» ядра. Подробности о некоторых таких методах можно найти в статье [15].

Виртуализация может быть поддержана центральным процессором на аппаратном уровне; так, в 2006 году почти одновременно Intel и AMD включили в свои очередные процессоры линейки x86 расширения Intel VT-x и AMD-V, специально предназначенные для организации виртуальных машин.

Существуют и другие способы одновременного запуска нескольких систем на одной машине. Наиболее универсальным можно считать подход, предполагающий *интерпретацию машинного кода*, когда программа-эмулятор фактически выполняет (эмулирует) работу центрального процессора. Несомненным достоинством такого варианта является возможность эмулировать любой процессор на любой машине и отсутствие необходимости поддержки со стороны операционной системы; совершенно очевиден и фундаментальный недостаток такой схемы — чрезвычайно низкая эффективность: на обработку *одной* машинной команды эмулируемой программы уходят *десятки* реальных машинных команд. Несмотря на это, именно такой режим эмуляции находит применение, во-первых, для запуска программ, написанных для очень старых компьютеров — IBM PC-совместимых машин эпохи MS-DOS, игровых компьютеров восьмидесятых годов типа Atari, Commodore-64 и т. п.; современные машины превосходят их по быстродействию в сотни раз, так что потери на эмуляцию перестают быть серьёзной проблемой. Так работает, например, широко известный эмулятор DosBox.

Встречаются и промежуточные варианты. Так, известный эмулятор Wine, позволяющий запускать под управлением систем семейства Unix программы, написанные для Windows, на самом деле, строго говоря, не является эмулятором. Дело в том, что программы, написанные под Windows, за вводом-выводом и другими привилегированными услугами обращаются не напрямую к операционной системе, а к слою систем-

ных библиотек, вызывая их функции, составляющие так называемый WinAPI[6]; все эти функции системными вызовами не являются, а «настоящие» системные вызовы под Windows вообще не документированы, и прямое их использование в программах не предполагается. Wine подменяет библиотеки Windows своими, реализованными через системные вызовы той системы, на которой он работает; загрузка программы, предназначенной для Windows, в качестве процесса на unix-системах оказывается делом довольно сложным, поскольку адресное пространство процесса на Windows устроено не так, как на unix-системах, но с этой проблемой Wine ухитряется справляться, частично модифицируя код, прочитанный из исполняемого файла.

Ещё один известный эмулятор, `qemu`, применяет гибридный подход. `Qemu` способен имитировать работу разных процессоров, при этом сам исполняясь на разных архитектурах; при необходимости он применяет интерпретацию машинного кода, но когда эмулируемая система команд совпадает с той, на которой запущен сам `qemu`, он для увеличения скорости работы частично выполняет код непосредственно на процессоре, при этом используя поддержку виртуализации, встроенную в ядро операционной системы.

В современных условиях встречаются операционные системы, специально предназначенные для работы на виртуальных машинах; управлять физическим компьютером они не могут. Кроме того, широко распространены специальные модификации ядер систем общего назначения, таких как Linux и FreeBSD, предназначенные для более эффективной работы этих ядер в роли гостевых систем. Такой подход к виртуализации, предполагающий поддержку со стороны ядра гостевой системы, получил название ***паравиртуализации***. В некоторых случаях гостевая операционная система работает в виртуальной машине без модификаций — например, модифицировать ядро Windows не представляется возможным, поскольку его исходные тексты недоступны — но при этом в ней устанавливается набор драйверов устройств специально для работы под управлением конкретного гипервизора. Этот вариант также относят к паравиртуализации.

Рассказ о виртуальных машинах будет неполным без упоминания варианта, когда одно ядро обслуживает как хостовую, так и гостевые системы. Строго говоря, операционная система в этом случае одна, разделяются только пространства пользовательских процессов, причём процессы основной системы (её называют *hardware node*) могут видеть все другие процессы, в том числе и запущенные в рамках виртуальных систем — так называемых *контейнеров* (англ. *container*), или *виртуальных окружений* (англ. *virtual environment, VE*). Процессы, работающие внутри контейнера, видят только друг друга; ни процессы основной

[6]Напомним, что аббревиатура API означает *application programming interface*, т. е. *интерфейс прикладного программирования.*

системы, ни процессы других контейнеров с их точки зрения не существуют. Аналогичная ситуация создаётся также и с файлами: процессы основной системы видят всё дерево каталогов целиком, тогда как процессы внутри контейнера в качестве корневого каталога воспринимают некий каталог, специально созданный для данного контейнера, и видят только поддерево, начинающееся с этого каталога. При старте контейнера основная система запускает в нём его собственный процесс `init`, после чего контейнер работает практически как обычная unix-машина.

Такие виртуальные машины — контейнеры часто применяются хостинговыми провайдерами. Пользователю для запуска серверных программ предоставляется отдельный контейнер с администраторским доступом к нему, то есть в контейнере существует свой собственный пользователь `root`, не способный влиять на основную систему, но обладающий полной властью над системой внутри контейнера. Пользователь может установить любое нужное ему программное обеспечение (в основном обычно серверное, но ограничиваться этим не обязательно) и настроить его в соответствии со своими пожеланиями. Такой вид услуги называется VPS (*virtual private server*); стоимость VPS может быть в десятки раз ниже, чем аренда физического сервера. Иногда VPS создаётся с использованием гипервизоров (обычно это Linux KVM), но чаще — в виде контейнеров; это позволяет провайдеру хостинга более эффективно использовать аппаратуру своих серверов, ведь здесь не нужно запускать отдельное ядро ОС для каждой виртуальной машины. Наиболее популярная в наши дни реализация этого подхода называется OpenVZ и представляет собой модификацию ядра Linux. При описании самого ядра Linux механизм, изолирующий процессы отдельных контейнеров друг от друга, по-английски называют *namespaces*, что переводится как «пространства имён» (хотя о каких именах идёт речь, не вполне понятно). Полезно знать, что аналогичная возможность ядра FreeBSD называется *jail*[7].

8.1.4. Структура и основные подсистемы ядра

Ядра операционных систем обычно проектируются по «слоёной» схеме снизу, от непосредственного управления аппаратурой, вверх — к интерфейсу системных вызовов. Каждый следующий слой наращивает степень абстрагированности, то есть предоставляет следующему слою более обобщённые и удобные функции, чем те, на основе которых он реализован сам.

[7] Английское слово *jail* формально переводится как «тюрьма», но на самом деле это скорее изолятор, помещение, оборудованное для содержания узников в течение короткого времени, что-то вроде существующих в России СИЗО и спецприёмников; небезызвестные «обезьянники» в отделениях полиции тоже по-английски будут обозначаться как jails.

Самый «нижний» слой включает в себя драйверы физических внешних устройств, обработчики аппаратных прерываний и другие компоненты, предназначенные, чтобы справляться с особенностями конкретной аппаратной платформы, в том числе процессора и шины. Именно на этом слое обычно присутствует сравнительно небольшое, но всё же заметное количество программного кода, написанного на языке ассемблера.

Остальные слои не имеют столь же чётких границ; в зависимости от того, какую из подсистем мы обсуждаем, может меняться даже их общее количество. Например, если мы посмотрим на подсистему управления памятью, то в аппаратно-зависимом слое обнаружим функции, работающие с MMU конкретного процессора. «Этажом выше» мы увидим функции, отвечающие за учёт, выделение и освобождение кадров физической памяти; реализация этих функций почти не зависит от того, для какого процессора (аппаратной платформы) компилируется код ядра — в расчёт здесь берётся только то, что на разных платформах размер кадра может отличаться. Имея в своём распоряжении физические кадры, следующий слой реализует более удобную абстракцию *выделения физической памяти* — как для нужд самого ядра, так и для поддержки виртуальных адресных пространств процессов. Ещё выше располагается подсистема, отвечающая за отображение в память дисковых файлов (см. §5.3.7, системный вызов `mmap`), на которую вынуждены полагаться не только подсистема, отвечающая за управление виртуальной памятью, но и подсистема, обеспечивающая работу с файлами (так называемая ***виртуальная файловая система***, англ. *virtual file system, VFS*). Это уже «самый верхний этаж» ядра; соответствующие системные вызовы обращаются непосредственно к этим подсистемам. Надо сказать, что обе эти подсистемы — и подсистема управления виртуальной памятью, и виртуальная файловая система — обращаются также и к другим подсистемам, расположенным в более низких уровнях. Например, подсистеме виртуальной памяти нужны возможности подсистемы, отвечающей за откачку и подкачку (своппинг), а виртуальная файловая система должна, очевидно, обращаться к подсистемам, отвечающим за диски, но не только — для подключения «удалённых» или «сетевых» файловых систем ей нужно взаимодействовать с подсистемой работы с сетью, и т. д.

Если мы попытаемся предпринять аналогичную экскурсию по подсистеме внешних запоминающих устройств (англ. *storage*), то на нижнем слое увидим подпрограммы, обеспечивающие взаимодействие с конкретными типами дисковых контроллеров (так называемые ***драйверы*** физических устройств) и обработчики аппаратных прерываний от дисковых устройств, слоем выше — функции, создающие абстракцию знакомого нам по §5.3.5 *блочного устройства*; здесь же расположены функции, отвечающие за планирование дисковых операций. На блочное

устройство полагаются в своей работе уже знакомые нам виртуальная файловая система и подсистема своппинга. Что касается виртуальной файловой системы, то ей для работы требуются также реализации конкретных файловых систем; более подробный разговор о виртуальной файловой системе у нас ещё впереди.

Изучая часть ядра, обеспечивающую взаимодействие по компьютерным сетям, мы на нижнем её слое увидим драйверы (физических) сетевых карт и, как можно догадаться, обработчики их прерываний; уровнем выше реализуется единая абстракция «сетевого интерфейса», ещё выше расположены реализации конкретных сетевых протоколов, затем — модули, поддерживающие семейства протоколов, и, наконец, подсистема, реализующая хорошо знакомые нам *сокеты* (см. гл. 6.3). Ещё одна подобная «башня» из подсистем прослеживается в окрестностях планировщика времени центрального процессора: на нижнем слое имеется подсистема, отвечающая за программирование обработчиков аппаратных прерываний, и сами эти обработчики, в том числе обработчик небезызвестного прерывания таймера; выше расположен собственно планировщик, обслуживающий (на основе данных о приоритетах) так называемые *потоки ядра* (*kernel threads*), на основе которых реализуются в том числе и обычные процессы. Непосредственно над планировщиком располагаются функции, реализующие блокирующие примитивы взаимоисключения (мьютексы ядра и некоторые другие примитивы, предполагающие возможность блокировки), над ними — подсистема, отвечающая за единицы планирования (треды) в том виде, в котором они известны пользователю, а на самом верхнем уровне — подсистема, отвечающая за формирование и ликвидацию пользовательских процессов, включающая, в частности, системные вызовы `fork`, `execve`, `_exit` и `wait`.

Если говорить о неких неведомых «основных подсистемах» ядра, то обычно к ним относят, во-первых, аппаратно-зависимый слой, включающий, помимо прочего, обработчики прерываний и драйверы физических устройств; во-вторых, подсистему управления процессами, включая планирование времени ЦП; в-третьих, менеджер оперативной памяти; в-четвёртых, подсистему внешнего хранилища (*storage*), в которую входит ***виртуальная файловая система*** (*VFS*); в-пятых, сетевую подсистему. Этот список, естественно, можно продолжить, детализировать и т. д., но общее представление о ядре он уже даёт.

Более детально архитектуру ядра операционной системы имеет смысл рассматривать на примере какого-то конкретного ядра, поскольку различия между построением ядра Linux и ядра той же FreeBSD достаточно существенны, чтобы не позволять детальных рассуждений о «ядре вообще». Например, архитектура ядра Linux вплоть до роли отдельных функций разобрана в книге [9]. Столь глубокое проникновение в проблематику ядер операционных систем может быть весьма

интересно, но требует большой траты сил и времени; мы посвятим остаток этой части книги сравнительно краткому обсуждению общих принципов функционирования отдельных подсистем ядра, а читателю, заинтересовавшемуся «ядерным программированием», порекомендуем обратиться к специальной литературе и, конечно, к сети Интернет.

8.2. Управление процессами

8.2.1. Процесс как объект ядра системы

Как мы уже знаем, ***процесс*** — это некая сущность, создаваемая ядром операционной системы, чтобы выполнять пользовательскую программу. До сих пор мы рассматривали процессы с точки зрения программиста, пишущего пользовательские программы; процесс при этом воспринимается как нечто такое, что существует *вне* ядра: это прежде всего области памяти, содержащие машинный код программы, её глобальные структуры данных (в том числе кучу) и стек; иначе говоря, *программа и её состояние исполнения*.

В §5.4.1 мы упоминали, что процесс можно рассматривать совершенно иначе — не как нечто внешнее по отношению к ядру, а как *объект ядра*. В самом деле, ядро должно где-то хранить информацию о выполняющейся пользовательской задаче: список выделенных ей областей памяти, приписанные задаче полномочия, таблицу файловых дескрипторов и многие другие сведения. Кроме того, как мы уже хорошо знаем, процесс выполняется не всегда: он может быть блокирован или просто ждать своей очереди. Операционной системе нужна возможность снова продолжить исполнять этот процесс, для чего следует помнить значения всех регистров центрального процессора на момент снятия процесса с исполнения. Эту информацию называют ***контекстом выполнения процесса***, и её тоже надо где-то хранить. Естественно, в памяти ядра для всего этого создаётся структура данных, которая как раз и представляет собой процесс как объект.

Для запуска пользовательской программы операционная система должна выделить нужное количество памяти под код, под инициализированные данные, под неинициализированные данные и под стек, после чего первые две области памяти заполнить информацией, хранящейся в исполняемом файле программы, сформировать вышеупомянутый *контекст выполнения*, состоящий из начальных значений регистров. В частности, в этот контекст нужно записать адрес точки входа в программу — вспомните метку `_start` в наших ассемблерных программах — в качестве исходного значения «счётчика команд», а также начальное значение указателя стека; как правило, находятся и другие регистры, начальное значение которых по каким-то причинам важно. Существуют и другие подготовительные действия, зависящие от конкретной системы;

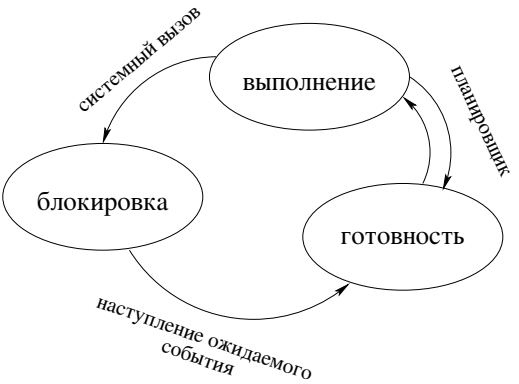

Рис. 8.1. Упрощённая диаграмма состояний процесса

например, ОС Unix должна расположить в памяти запускаемой программы слова, из которых состоит командная строка, и занести в стек указатели на эти слова, а также сделать ещё некоторые приготовления подобного рода.

Интересно, что все эти действия могут быть связаны с возникновением нового процесса, а могут и не быть; например, в ОС Unix, как мы знаем, процесс возникает как копия какого-то из существующих процессов, а затем в уже созданном процессе производится замена одной выполняемой программы на другую, то есть действия по запуску программы, перечисленные в предыдущем абзаце, производятся, когда процесс как таковой уже есть.

Вне зависимости от того, в какой момент создаётся процесс и происходит ли при этом загрузка новой программы, для нового процесса должна быть выделена и так или иначе заполнена память, нужно создать настройки ***MMU***[8] для преобразования его виртуальных адресов в физические и т. д.; результатом всей этой подготовительной работы становится полностью готовая к работе структура данных (объект) процесса. Затем новый процесс включается в очередь процессов, готовых к выполнению и ожидающих своей очереди, а управление процессу потом передаст планировщик — подсистема ядра, отвечающая за планирование времени центрального процессора.

В конкретный момент времени процесс может как исполняться, так и не исполняться, причём процесс, который сейчас не исполняется, может быть как готов к возобновлению исполнения, так и не готов: например, процесс может ожидать результатов операции ввода-вывода. Говорят, что процесс может находиться в одном из трёх состояний: *выполнение*, *блокировка* (в ожидании события) и *готовность* (см. рис. 8.1). Между

[8] *Memory management unit* — схема в составе центрального процессора, отвечающая за преобразование виртуальных адресов в физические. См. т. 2, §3.1.2.

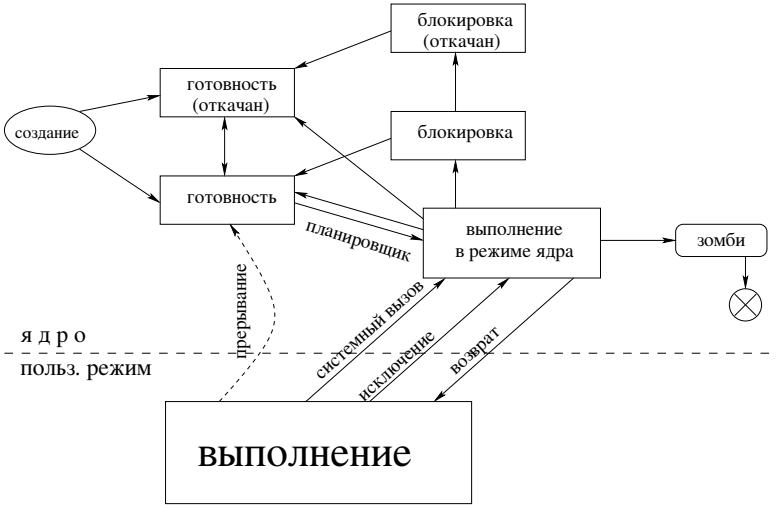

Рис. 8.2. Жизненный цикл процесса

состояниями выполнения и готовности процесс переходит при вмешательстве планировщика времени центрального процессора; один процесс может быть снят с выполнения, то есть переведён из состояния выполнения в состояние готовности, а другой при этом, наоборот, поставлен на выполнение. Отметим, что наша диаграмма соответствует положению вещей в системах разделения времени. В системах, реализующих пакетную мультизадачность, переход из состояния выполнения в состояние готовности никогда не производится, то есть аналогичная диаграмма для пакетного режима содержит на одну стрелку меньше.

Если не учитывать возможность откачки памяти процессов на диск (а наша упрощённая диаграмма этого не учитывает), то в состояние блокировки процесс может попасть только одним способом: выполнив такой системный вызов, после которого немедленное продолжение выполнения невозможно. Например, процесс может затребовать чтение данных с диска; в этом случае продолжать выполнение имеет смысл не раньше, чем данные будут прочитаны, что требует времени. Также процесс может в явном виде потребовать приостановить его выполнение на несколько секунд или до поступления внешнего сигнала, вызвав `sleep`, `nanosleep`, `pause` и т. п. Ранее нам встречались и другие случаи блокировки.

Когда компьютер, на котором запущена операционная система, обладает механизмом виртуальной памяти (то есть в процессоре есть MMU), а среди периферийной аппаратуры присутствует внешнее запоминающее устройство (попросту говоря, диск), обычно операционная система при нехватке оперативной памяти может временно ***откачать*** на диск

содержимое отдельных областей памяти, принадлежащих пользовательским задачам. Как правило, для этого выбираются такие области памяти, которые давно не использовались их владельцами. При этом у нас появляется ещё одна причина блокировки: если процесс попытался обратиться к области своей виртуальной памяти, которая в настоящий момент откачана, то система запланирует операцию обратной подкачки нужных областей, а сам процесс заблокирует до тех пор, пока подкачка не будет завершена. Как мы увидим несколько позже, виртуальное адресное пространство делится на ***страницы***, и каждая такая страница откачивается и подкачивается как единое целое; при обращении процесса к своей странице, которой нет в памяти, операционная система обычно подкачивает только одну эту страницу.

С учётом возможности откачки жизненный цикл процесса принимает вид, показанный на рис. 8.2. Под «откачанным» здесь понимается процесс, который не может продолжить работу, поскольку в оперативной памяти отсутствует хотя бы одна из страниц, нужных ему «прямо сейчас» — для выполнения той инструкции, на которую указывает счётчик команд. Это с равным успехом может быть страница из секции данных или стека, содержащая ячейки, к которым обратилась очередная машинная инструкция, а также страница из секции кода, содержащая саму очередную инструкцию. Если некоторые из страниц, принадлежащих процессу, откачаны, но он при этом может продолжать работу, весь процесс как целое откачанным не считается.

При постановке процесса на выполнение ядро сначала производит некоторую подготовительную работу; вместе с обработкой системных вызовов и исключений (внутренних прерываний) это составляет упоминавшееся ранее *выполнение процесса в режиме ядра*.

Процесс может оказаться снят с исполнения (перейти в состояние готовности), минуя стадию выполнения в режиме ядра. Это может произойти, если ядро примет решение о смене активного процесса во время обработки аппаратного прерывания (например, прерывания таймера). Возобновление выполнения процесса в любом случае требует подготовительных действий в режиме ядра.

Стоит обратить внимание на то, что процесс, попавший в режим блокировки (например, ожидающий результатов ввода-вывода), может как быть откачан, так и оставаться в памяти, если откачка системе не потребовалась; но если процесс всё же был откачан, обратная подкачка будет осуществлена не раньше, чем процесс окажется готов к выполнению (то есть исчезнет причина блокировки). Если говорить точнее, сначала исчезает основная причина блокировки — наступает событие, которого процесс ждал; после этого процесс остаётся заблокированным, но уже по причине отсутствия в памяти нужных ему страниц, и лишь тогда система принимает меры к их подкачке.

В предыдущей части книги мы обсуждали ***многопоточное программирование***, в основе которого лежит запуск ***легковесных процессов*** (***тредов***). Легковесный процесс представляет собой дополнительную единицу планирования в рамках одного обычного процесса; иначе говоря, обычный процесс в таких системах можно представить как группу легковесных процессов, работающих одновременно с одними и теми же кодом, данными и ресурсами. Как мы видели, с точки зрения программиста это выглядит как возможность запуска некоторых подпрограмм (функций) *параллельно* основной программе — одновременно с продолжением её выполнения. Для каждого легковесного процесса создаётся своя отдельная секция стека, чтобы хранить локальные переменные и параметры той функции, которая была запущена параллельно остальной работе, а также функций, вызываемых из неё. Все остальные секции у легковесных процессов общие с главным, и даже эти «личные» стеки никак друг от друга не защищены.

Когда в рамках процесса выполняются легковесные процессы, о состоянии (выполнение, готовность, блокировка) имеет смысл говорить в отношении каждого из легковесных процессов отдельно; вообще, каждый из них имеет свой собственный жизненный цикл, практически такой же, как у обычного процесса; единственное различие обнаруживается в порядке завершения легковесных процессов, ведь вызов `wait` для их окончательного снятия не требуется. Естественно, для каждого из них операционная система вынуждена хранить свой контекст выполнения (значения регистров ЦП); собственно говоря, для ядра системы каждый тред — это почти настоящий процесс; одно на всех пространство памяти — это очень заметный фактор с точки зрения программиста, работающего в пользовательском пространстве (пишущего обычные программы), но довольно незначительное обстоятельство при взгляде со стороны ядра.

8.2.2. Планирование времени процессора

Планировщик времени центрального процессора — это важнейшая часть любой многозадачной операционной системы; как мы знаем ещё из второго тома (см. §3.6.1), основные типы мультизадачных операционных систем различаются как раз по принципам планирования времени ЦП.

При описании планировщика иногда выделяют *три уровня планирования*: долгосрочное, среднесрочное и краткосрочное. На долгосрочном уровне принимается принципиальное решение о допуске задачи к выполнению; иначе говоря, долгосрочный планировщик решает, какие задачи будут выполняться в системе одновременно, а какие будут отложены до тех пор, пока не завершится часть уже выполняемых программ. Введение понятия долгосрочного планирования может показаться противоречащим нашему опыту, ведь при работе с ОС Unix все запускаемые

программы всегда начинали выполнение немедленно; это действительно так, в системах с разделением времени долгосрочное планирование либо не применяется вовсе, либо применяется, но исключительно для распределения процессов между несколькими процессорами. С другой стороны, долгосрочное планирование крайне важно в системах реального времени, где чрезмерная нагрузка на систему может привести к недостижению её главной цели — обеспечения успешного завершения определённых задач к указанному моменту. Кроме того, долгосрочное планирование применяется в системах пакетной мультизадачности (вроде суперкомпьютеров), где обычно присутствует входящая очередь заданий и нужно принимать решение о запуске очередного задания из этой очереди с учётом текущей загруженности системы.

О среднесрочном планировании обычно говорят при обсуждении откачки процессов. По мере исчерпания физической оперативной памяти ядро вытесняет (откачивает на диск) принадлежащие процессам виртуальные страницы, которые давно не использовались; но рано или поздно может сложиться ситуация, когда оперативную память потребуется освободить, а «ненужных» страниц не найдётся. В этом случае системе придётся принять решение об откачке страниц, без которых тот или иной процесс выполняться дальше не может; процесс при этом перейдёт в состояние «откачан» (см. рис. 8.2 на стр. 334). Принятие решения о том, какой именно из процессов будет откачан, а также о том, какой из откачанных процессов следует подкачать (точнее, подкачать те из его страниц, которые ему нужны для исполнения прямо сейчас), как раз и называется среднесрочным планированием. В системах без свопинга этот уровень планирования отсутствует.

Наконец, ***краткосрочное планирование*** или ***диспетчеризация процессов*** состоит в принятии решений о том, какие из выполняемых процессов должны быть принудительно переведены в режим готовности («вытеснены», англ. *preempted*), какие из готовых к выполнению процессов должны быть поставлены на выполнение при наличии возможности и какой длины кванты времени им следует выделить. Надо сказать, что в системах с разделением времени обычно задействуется только этот уровень планирования: долгосрочный планировщик отсутствует вовсе, а среднесрочный в системе, как правило, есть, но его вынужденное включение в работу свидетельствует о предаварийной перегруженности системы, чего стараются не допускать. В системах с пакетной мультизадачностью, напротив, краткосрочное планирование почти отсутствует — точнее, имеет рудиментарную форму. Задачи в пакетном режиме никогда не переводятся из состояния выполнения в состояние готовности, поскольку, напомним, снимаются с выполнения только при завершении и при уходе в блокировку; при наличии свободных процессоров диспетчер просто выбирает ту из готовых к выполнению задач, у которой

выше значение приоритета (если в системе вообще есть приоритеты), либо просто выбирает первую задачу, стоящую в очереди.

Вне зависимости от того, с какой системой мы имеем дело, задачи можно условно поделить на те, скорость выполнения которых в основном зависит от доступного процессорного времени, и те, которые, напротив, бóльшую часть времени проводят в блокировках (обычно на операциях ввода). Потребности последних в процессорном времени невелики. По-английски такие задачи называются соответственно *CPU-bound* и *I/O-bound*. Считается, что хороший планировщик должен учитывать разницу между этими задачами и их потребностями. Так, краткосрочный планировщик может заметно повысить общую производительность системы, если будет выделять часто блокирующимся задачам пусть и небольшие кванты времени, но делать это быстро, как только задача вышла из очередной блокировки. Видимое время отклика системы на внешние события это снизит, а на «вычислительные» задачи практически не повлияет. С другой стороны, если в системе присутствует долгосрочный планировщик, то в его задачу обычно входит соблюдение некоего баланса между «вычислительными» и «блокирующимися» задачами.

Алгоритмы краткосрочного планирования бывают очень разными, от совсем простых до весьма заковыристых, особенно в системах реального времени. Планирование в реальном времени — отдельная область научных исследований, этому вопросу посвящены целые книги — как популярные, так и монографии; мы эту проблематику затрагивать не будем. Что касается систем разделения времени, то самый простой вариант алгоритма планирования — ***циклический***[9], при котором диспетчер выстраивает все имеющиеся задачи в кольцевой список и выбирает каждый раз следующую задачу, готовую к выполнению, то есть блокированные задачи пропускает при просмотре списка, но не исключает из него; кванты времени выделяются одинаковые.

Чуть сложнее устроен алгоритм очереди (FIFO): диспетчер выстраивает готовые к выполнению задачи в очередь, выделяет задачам одинаковые кванты времени, выбирает для выполнения всегда первую задачу из очереди, а задачи, снятые с выполнения, как и задачи, вышедшие из состояния блокировки, ставит в конец очереди. Алгоритм можно модифицировать: на выходе из блокировки задачу ставить в начало очереди, а не в её конец, чтобы задаче, долго ждавшей какого-то события, не приходилось ждать ещё и своего первого кванта времени; при этом, впрочем, есть риск, что задачи специально будут обращаться к системе и блокироваться на короткое время, чтобы почти сразу получить следующий квант.

[9] Английский термин здесь — *round-robin*. Попытки перевести слово «циклический» обратно на английский — всевозможные *cyclic*, *loop* и прочее — ведут к полному непониманию.

Циклическое планирование и планирование на основе очереди не подразумевают никаких приоритетов для выполняемых задач; между тем в реальности часто требуется одним задачам назначить повышенный приоритет, чтобы, например, ускорить реакцию на действия пользователя (или клиента, если речь идёт о сервере), тогда как другим установить низкий приоритет, чтобы они выполнялись, когда системе больше «нечего делать» — например, долгие математические расчёты, результат которых требуется не срочно.

В §5.4.2 мы уже обсуждали, что процесс может обладать двумя составляющими его приоритета — статической, задаваемой извне, и динамической, пересчитываемой планировщиком по мере выполнения задач в системе. Простейший вариант планирования с двумя составляющими приоритета выглядит так: динамический приоритет может меняться от нуля до некоторого максимального значения; при постановке задачи на исполнение её динамический приоритет обнуляется, а динамический приоритет всех остальных готовых к выполнению задач увеличивается на единицу. Для постановки на выполнение планировщик выбирает задачу, имеющую максимальную сумму приоритетов — статического и динамического. Некоторые системы позволяли задавать процессам значения статических приоритетов, превышающие максимально возможное динамическое значение; в этом случае такие процессы могут захватить процессор и долгое время не давать выполняться процессам с более низким приоритетом.

В современных системах подобное тоже возможно, но в более гибком варианте. Например, в ОС Linux с помощью системного вызова `sched_setscheduler` для процесса можно задать тип используемого планировщика или, точнее, *режим планирования* (англ. *scheduling policy*): `SCHED_RR`, `SCHED_FIFO`, `SCHED_OTHER`, `SCHED_BATCH` и `SCHED_IDLE`. Обычные процессы используют `SCHED_OTHER`. Режимы `SCHED_RR` и `SCHED_FIFO` называются режимами «реального времени» и отличаются друг от друга применяемым алгоритмом планирования (Round-Robin или FIFO), но при этом имеют всегда заведомо более высокий приоритет, чем все остальные процессы; иначе говоря, при наличии в системе готового к выполнению процесса, имеющего режим `SCHED_RR` и `SCHED_FIFO`, любой другой процесс будет снят с исполнения, чтобы дать возможность работать процессам реального времени. Сами процессы реального времени различаются значением статического приоритета, который задаётся отдельным параметром. Динамическая составляющая приоритета для этих процессов отсутствует, так что процессы с более высоким приоритетом всегда вытесняют процессы, приоритет которых ниже.

Относительно процессов, находящихся в режиме `SCHED_BATCH`, ядро предполагает, что это «счётные» задачи (*CPU-bound*), которым нужно много процессорного времени, но для которых не критично время реакции. При принятии решения ядро отдаёт предпочтение процессам, имеющим режим `SCHED_OTHER`, предполагая, что они займут меньше времени; в то же время этот вариант предпочтения не является абсолютным, то есть рано или поздно процесс в режиме `SCHED_BATCH` всё же получит управление, даже если в системе имеются

готовые к выполнению процессы в режиме SCHED_OTHER. Наконец, процессы в режиме SCHED_IDLE выполняются лишь тогда, когда других задач нет; как только появляется готовый к выполнению процесс, имеющий любой другой режим, процесс в режиме SCHED_IDLE немедленно вытесняется.

Описанное выше решение на основе статической и динамической составляющих приоритета даёт возможность понять, как в общих чертах устроен планировщик, но при «лобовой» реализации такой планировщик оказывается неэффективен *сам по себе*, особенно когда в системе много готовых к выполнению процессов. В самом деле, планировщик получает управление, как мы знаем, по прерыванию таймера, причём довольно часто; если слова об увеличении динамического приоритета воспринять буквально, то планировщику придётся при каждом получении управления пройти по всему списку процессов, готовых к выполнению, увеличить каждому из них значение динамического приоритета, найти процесс с максимальной суммой приоритетов, и если она достаточно высока — поставить этот процесс на выполнение, вытеснив один из выполняющихся (либо единственный выполняющийся, если в системе только один процессор или планирование производится для каждого процессора отдельно). В принципе цикл по сравнительно небольшому (в пределах нескольких сотен элементов) списку занимает не так много времени, но поскольку его приходится прогонять, например, тысячу раз в секунду, потери могут оказаться заметными. Кроме того, не следует забывать, что всё это происходит в обработчике прерывания, то есть при запрещённых аппаратных прерываниях. Плюс к тому едва ли не во всех современных компьютерах имеется больше одного процессора, так что доступ к списку процессов приходится считать критической секцией и организовывать взаимоисключение, которое к тому же не может воспользоваться блокировкой; в самом деле, блокировки обеспечиваются планировщиком, но не может же он обеспечить их сам для себя, не говоря уже о том, что в обработчиках прерываний у нас нет контекста процесса. В целом приходится признать, что устройство планировщика нуждается в усовершенствовании: буквальным образом пересчитывать значение динамического приоритета для всех имеющихся процессов слишком долго.

Реально существующие в современных системах планировщики применяют алгоритмы, позволяющие производить любые пересчёты только при постановке задачи на выполнение и при снятии её с выполнения, а какие бы то ни было циклы по всем имеющимся задачам при этом полностью исключаются.

Например, в современных версиях ядра Linux используется так называемый *Completely Fair Scheduler* (CFS; название буквально переводится как «полностью справедливый планировщик»), основанный на принципе «справедливого деления» времени между процессами. Организуется такое «честное деление» довольно просто: для каждого процесса планировщик помнит, сколько времени тот успел отработать, и выбирает

для постановки на выполнение те задачи, которые на текущий момент получили времени меньше других. Чтобы объяснить, как этот планировщик работает, представим себе для начала, что все процессы в системе существуют с момента начала работы самой системы. Конечно, такое предположение не соответствует действительности ни в каком виде, но мы вскоре увидим, как от него избавиться.

Заведём для каждого процесса переменную, которая будет хранить время (*буквально* время, измеряемое в наносекундах), предоставленное данному процессу до настоящего момента. В коде ядра Linux эта переменная — если быть точным, поле структуры, связанной с процессом — называется `vruntime`. Продолжая считать, что все процессы стартовали вместе с системой, отметим, что исходно `vruntime` для всех процессов должна быть равна нулю, ведь они ещё не успели начать работу. Планировщик выбирает процесс с наименьшим значением `vruntime` и ставит его на выполнение, предоставив ему квант времени, длина которого, что важно, определяется значением приоритета — хорошо знакомым нам *nice value* (см. §5.4.8). По истечении этого кванта планировщик снимает процесс с выполнения и возвращает его в очередь, увеличив его `vruntime` (буквально!) на число наносекунд, соответствующее кванту времени, который только что был процессу предоставлен и им использован, после чего снова выбирает процесс с наименьшим значением `vruntime`.

Поскольку кванты времени, как мы уже поняли, могут различаться по своей длине, в очереди на выполнение очень быстро могут скопиться процессы с довольно большим разбросом значений отработанного времени, так что планировщику нужно содержать очередь в виде, упорядоченном по значению `vruntime`, и уметь эффективно вставлять в эту очередь элементы с сохранением упорядоченности. Планировщик CFS для этого использует структуру данных, известную как *красно-чёрное дерево* — разновидность самобалансирующегося двоичного дерева поиска.

Читатель может обратить внимание, что мы не разбирали работу со сложными структурами данных; в первом томе, рассказывая о двоичных деревьях поиска, мы обошли вниманием алгоритмы их балансировки, ограничившись замечанием, что такие существуют. Это было сделано вполне намеренно: тот уровень знаний и навыков, на который рассчитан текст первого тома, для освоения тех же красно-чёрных деревьев явно недостаточен.

Мы надеемся, что сейчас, подходя к концу третьего тома, читатель уже готов к восприятию хитросплетений сложных структур данных; удовлетворить своё любопытство относительно красно-чёрных деревьев и других методов построения сбалансированных деревьев поиска читатель может, воспользовавшись книгами [6], [7] и [8]. Все эти книги упоминались и в первом томе, но сейчас, возможно, у вас получится лучше. Впрочем, для понимания того, как именно работает планировщик CFS, не обязательно в деталях изучать красно-чёрные деревья, достаточно знать, что операции вставки, поиска и удаления элемента для такого дерева выполняются достаточно быстро.

В дереве поиска процессы, успевшие отработать меньше других, оказываются слева; планировщику остаётся только выбрать крайний левый элемент, изъять его из дерева и поставить соответствующий процесс на выполнение, а затем вернуть элемент в дерево уже с новым (увеличенным) значением `vruntime`, так что его новая позиция окажется где-то близко к правому краю дерева. Получается, что процессы в дереве постепенно продвигаются справа налево, так что рано или поздно каждый из готовых к выполнению процессов (а в дереве находятся только такие) получит свой квант времени.

Вернёмся теперь к нашему (довольно безумному) предположению, что все процессы стартовали вместе с системой. Мы прекрасно знаем, что такого быть не может, тем более что процессы приходится помещать в очередь на выполнение не только при их старте, но и при выходе из блокировки, и как раз с этим у описанного принципа планирования имеются явные проблемы. В самом деле, пусть наша система загрузилась час назад, и через небольшое время после загрузки в системе стартовал процесс, активно использующий процессорное время. К текущему моменту этот процесс мог отработать, к примеру, 1000 секунд процессорного времени, т. е. его значение `vruntime` будет равно 10^{12} (напомним, что единица измерения тут — наносекунды, то есть миллиардные доли секунды). Если теперь новый процесс, только что запущеный на выполнение, поставить в очередь с нулевым значением `vruntime` — что вроде бы логично на первый взгляд, ведь он ещё не успел поработать — то планировщик надолго забудет про старую задачу, виновную лишь в том, что она успела начать работу раньше. Серверным задачам, работающим годами, в такой системе вообще перестанет доставаться время.

Описанная проблема имеет очень простое решение: планировщик помнит *наименьшее* значение `vruntime` среди всех элементов очереди, и именно его (а не ноль!) присваивает полям `vruntime` для новых задач. Получается, что `vruntime` теперь не равна, строго говоря, времени, которое в действительности успел отработать процесс; про эту переменную можно сказать только то, что она *растёт* в соответствии с временем, выделяемым процессу, но в её текущей величине складываются отработанное время и некое начальное значение, определяемое моментом создания задачи.

Остаётся понять, что делать с процессами, выходящими из режима блокировки. Простейшим решением было бы поступать с ними так же, как поступают с новыми процессами: заносить в `vruntime` текущее минимальное значение и помещать процесс в начало очереди (левый край дерева), но это не совсем правильно, поскольку в блокировке процесс мог провести меньше времени, чем другие (готовые к выполнению) процессы проводят в ожидании. Поэтому планировщик поступает чуть хитрее: если при выходе из блокировки значение `vruntime` меньше текущего минимального, то присваивается минимальное значение (чтобы не

давать процессу несправедливого преимущества), в противном случае значение `vruntime` сохраняется без изменений. Есть и другие хитрости, связанные с планировщиком CFS; подробности можно узнать в книге [9], а также из посвящённого этому планировщику текста [16] из документации по ядру Linux.

8.2.3. Обработка сигналов

Сигналы, которые мы рассматривали в §5.5.2, при обсуждении устройства ядра интересны нетривиальностью реализации их доставки. Точнее, нетривиальностью отличается лишь один вариант ***диспозиции сигнала*** — выполнение функции-обработчика.

Обсуждая работу с сигналами, мы отметили, что функция-обработчик вызывается «в самый неожиданный момент», что требует определённой осторожности: из обработчиков сигналов нельзя вызывать функции, модифицирующие сколько-нибудь сложные структуры данных за пределами самого обработчика. С точки зрения работающего процесса момент вызова обработчика действительно наступает неожиданно, поскольку, как мы знаем, снятие с выполнения и обратная постановка на выполнение происходят для процесса (точнее, для его пользовательской части — той пользовательской программы, которая в нём выполняется) совершенно прозрачно, то есть процесс не может отследить эти моменты. Но, обсуждая устройство ядра системы, мы можем указать вполне определённый момент, когда (конкретно!) вызывается и выполняется обработчик сигнала.

В §5.5.2 мы отметили, что в определённом смысле обработчик сигнала вызывается прямо из ядра. На самом деле это, конечно же, не так. При прямом вызове функции из кода ядра она сама тоже будет выполняться в режиме ядра, т. е. в привилегированном режиме ЦП, но это, как мы знаем, категорически недопустимо для функции, находящейся в пользовательской программе. Поэтому ядро поступает хитрее.

Прежде всего отметим, что в терминах ядра можно определённо сказать, в какой момент будет вызван обработчик сигнала. Если процессу был (в любой момент времени) отправлен сигнал, диспозиция которого предполагает вызов обработчика, то этот обработчик сработает, когда процесс в очередной раз будет переходить из состояния выполнения в ядре в состояние выполнения в ограниченном режиме (см. рис. 8.2 на стр. 334). Таких ситуаций можно перечислить три: постановка процесса на выполнение после ожидания в режиме готовности, возврат из системного вызова и возврат к выполнению после возникновения исключительной ситуации (внутреннего прерывания), если соответствующий сигнал (`SIGSEGV`, `SIGFPE`, `SIGBUS` или `SIGILL`) процессом перехвачен или игнорируется.

Во всех случаях перед передачей управления в пользовательский режим ядро проверяет, нет ли для процесса поступивших сигналов,

диспозиция которых предполагает выполнение обработчика, и если такой сигнал найден, производится последовательность действий, лучше всего характеризуемых словосочетанием *black magic*: ядро напрямую модифицирует стек процесса так, чтобы на вершине стека возник стековый фрейм[10] для выполнения обработчика сигнала — такой же, как если бы обработчик был вызван самой программой, за исключением того, что адрес возврата в этом стековом фрейме тоже «хитрый» — он указывает на заранее размещённый ядром в виртуальном адресном пространстве процесса кусочек машинного кода, который обеспечивает одно простое действие: обращение к системному вызову `sigreturn`. Этот вызов приводит стек процесса в исходное состояние после окончания выполнения обработчика, а также восстанавливает прежнее состояние *маски сигналов* (см. 6.4.7) и производит некоторую другую работу по очистке последствий столь необычного вызова функции.

Сам по себе вызов обработчика делается совсем просто: в поле структуры состояния процесса, хранящее значение регистра `EIP` (на других платформах — в соответствующий регистр, играющий роль указателя на текущую машинную команду), заносится адрес входа в тело функции-обработчика, после чего производится возврат управления коду процесса — самым обычным образом; но после всех описанных нами танцев управление при этом получает функция-обработчик, отрабатывает, выполняет свою инструкцию `RET` («возврат управления»), что приводит, как мы уже сказали, к вызову `sigreturn`. Между прочим, возврат из `sigreturn` тоже представляет собой ситуацию перехода процесса «из ядра» к обычному выполнению, так что здесь ядро вполне может вызвать обработчик следующего сигнала, если он успел прийти.

Понимание того, как реализованы вызовы обработчиков сигналов, проливает свет на многие аспекты обработки сигналов, которые до сей поры могли оставаться непонятными. Так, мы упоминали ранее, что сигналы обладают свойством «склеиваться»: если отправить процессу подряд много сигналов с одним и тем же номером, процесс в итоге вполне может получить только один такой сигнал. Дело в том, что факт получения процессом сигнала ядро отмечает взведением флага, то есть занесением единицы в соответствующий разряд обычной четырёхбайтной целой переменной, связанной с процессом. Перед вызовом обработчика сигнала этот флаг сбрасывается в ноль. Очевидно, что если продолжать «бомбить» процесс сигналами с тем же номером на протяжении временно́го периода до ближайшего перехода процесса «из ядра в юзерспейс», то каждый такой сигнал приведёт к повторному взведению флага, который и так уже взведён.

Кроме того, становится понятно, почему обрабатываемые сигналы прерывают работу блокирующих системных вызовов: для обработки

[10] Если вы почувствовали какую-то неуверенность, увидев термин «стековый фрейм», вернитесь ко второму тому и перечитайте главу 3.3.

сигнала нужно вернуться в ограниченный режим, то есть *прекратить* выполнение системного вызова, вернуться из него. Впрочем, как мы знаем, современные ядра можно попросить автоматически возвращаться после этого к выполнению прерванных системных вызовов — такой возврат тоже организует вызов `sigreturn`; работает это не для всех блокирующих системных вызовов, что добавляет ещё больше путаницы.

Чтобы лучше показать всю глубину извращённости происходящего, попытаемся описать ещё одну возможность, связанную с обработчиками сигналов, но для этого сначала придётся рассказать о паре стандартных библиотечных функций, которые уместнее было бы описать во втором томе — в главах, посвящённых стандартной библиотеке языка Си. Это функции `setjmp` и `longjmp`:

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

На самом деле `setjmp` обычно макрос, а не функция, что же касается типа `jmp_buf`, то он в большинстве случаев представляет собой массив, что и позволяет передавать его в обе функции без применения операции взятия адреса, даже если это действительно функции. Вызвав `setjmp`, мы при этом «сохраняем» текущее *состояние выполнения* — попросту говоря, текущую позицию в программе и текущее положение указателя стека. Сама `setjmp` при этом возвращает `0`.

Переменная `env` «действительна» до тех пор, пока продолжает существовать стековый фрейм, из которого вызвали `setjmp`, то есть пока вызвавшая её функция не вернёт управление. В течение всего этого времени можно — из той же функции или из любой функции, вызванной *из неё* прямо или косвенно — вернуться к состоянию выполнения, запомненному с помощью `setjmp`. Для этого вызывается функция `longjmp`; первым параметром она получает переменную, содержащую «законсервированное» состояние, а вторым — некое целочисленное значение, которое должно быть ненулевым. Выглядит «возврат в прошлое» так, как будто функция `setjmp` в том же самом месте снова вернула управление, только на этот раз она возвращает не ноль, а то значение, которое `longjmp` получила вторым параметром. Отметим, что, конечно, при этом восстанавливается только позиция выполнения и принудительно ликвидируются все более поздние стековые фреймы, но никакие побочные эффекты, в том числе присваивания глобальным переменным, назад не откатываются.

Примечательно сказанное в документации на эти функции для случая, если программисту придёт в голову вызвать `longjmp` для сохранённого состояния *после* того, как функция, в которой состояние было «упаковано», уже вернула управление. В английском оригинале это звучит лаконично и на удивление понятно: *total chaos is guaranteed*.

Реализация этих функций достаточно очевидна, хотя и требует применения ассемблерных вставок. Если говорить об изучавшейся нами системе регистров i386, то `setjmp` записывает в `EAX` значение `0`, записывает в `env` значения регистров `ESP` и `EIP` и выполняет `RET`; `longjmp` копирует свой параметр `val` в `EAX`, после чего восстанавливает значения `ESP` и `EIP` из параметра `val`, что приводит к повторному выполнению `RET` в теле `setjmp`, стек при этом находится в том же виде, в котором был на момент её первого вызова, но в `EAX` теперь не ноль, а новое значение — его-то функция `setjmp` и «возвращает» на сей раз.

Зачем нужны эти функции, читатель без труда может догадаться сам, если же с этим возникнут сложности — подождать выхода четвёртого тома, где, помимо прочего, будет описан механизм *обработки исключений* в языке Си++; для тех, кто знает, о чём речь, скажем, что `setjmp` и `longjmp` нужны примерно для того же самого. Дадим только один совет: если вы можете без них обойтись, лучше обойдитесь. Мы отнюдь не случайно решили не рассматривать их во втором томе.

Теперь, зная о существовании «длинных прыжков» (буквальный перевод слов *long jump*) и представляя, как они реализованы, отметим, что *длинный прыжок можно выполнить изнутри обработчика сигнала*. Автор вынужден признать, что испытал изрядный шок, когда много лет назад впервые узнал об этой возможности. Тем не менее, такая возможность не только есть, но для неё даже предусмотрены специальные варианты функций:

```
int sigsetjmp(sigjmp_buf env, int savesigs);
void siglongjmp(sigjmp_buf env, int val);
```

Ненулевое значение параметра `savesigs` предписывает сохранить текущую маску сигналов; если это было сделано, то `siglongjmp` перед выполнением «прыжка» восстанавливает сохранённую маску. Больше эти две функции от предыдущей пары ничем не отличаются; в частности, «выпрыгнув» из обработчика сигнала, мы *перепрыгнем через вызов `sigreturn`* (!). Впрочем, как легко догадаться, ничего страшного при этом не произойдёт: восстановление стека мы произвели сами, вызвав `siglongjmp`, он же восстановил маску сигналов, так что теперь мы спокойно обойдёмся без услуг ядра по очистке последствий «магического» вызова обработчика сигнала.

При всей чудовищности описанных инструментов автору известен как минимум один случай, когда весьма известная и часто используемая библиотека предполагает выпрыгивание из обработчика сигнала в качестве штатного способа досрочного завершения некоего действия. Речь идёт о библиотеке GNU Readline, которая во многих интерактивных программах вроде командных интерпретаторов или отладчика `gdb` обеспечивает редактирование вводимой команды, дополнение слов по нажатию `Tab`, хранение и поиск истории команд с помощью стрелок вверх/вниз и по нажатию `Ctrl-R`. Если читатель, вняв нашим рекомендациям, сделал командную строку своим основным рабочим инструментом (а мы надеемся, что это именно так, ведь в противном случае читатель вряд ли добрался бы почти до конца третьего тома), то он, несомненно, знает, что при нажатии `Ctrl-C` в тех же командных интерпретаторах вводимая строка сбрасывается и её ввод начинается сначала. Чтобы так сделать, нужно, находясь *где-то внутри* функции, собственно осуществляющей чтение строки с клавиатуры со всеми подобающими «примочками», заставить эту функцию завершиться. Столкнувшись с этим, автор попытался понять, как же сообщить библиотеке Readline, что активную функцию следует завершить прямо сейчас. Такого средства не нашлось, зато в документации обнаружилось подробное описание того, как можно перехватить сигнал `SIGINT` (напомним, именно он присылается по нажатию `Ctrl-C`) и как из него, а заодно и из пресловутой функции чтения строки выпрыгнуть с помощью `siglongjmp`.

Воистину, сон разума рождает чудовищ.

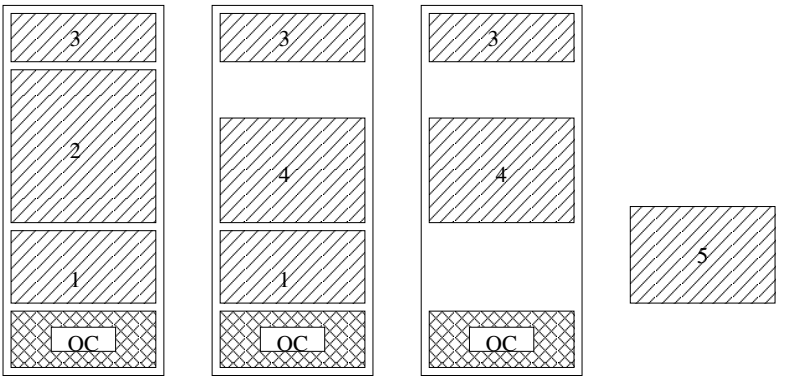

Рис. 8.3. Возникновение фрагментации памяти

8.3. Управление оперативной памятью

В ситуации, когда пользовательских задач запущено больше одной, возникает вопрос, как их расположить в памяти вычислительной машины и как разделить память между ними. Естественно, решение возлагается на операционную систему, в которой специально для этого создаётся подсистема, известная как *менеджер оперативной памяти*. Как мы уже знаем, для запуска полноценной мультизадачности требуется, чтобы аппаратура компьютера обладала определённым минимальным набором свойств, в который входит *защита памяти*. В реальности, как мы обсуждали ранее, центральный процессор, предназначенный для многозадачных компьютеров, обычно содержит схему, называемую *устройством управления памятью* (*Memory Management Unit*, MMU), и это устройство производит преобразование адресов перед обращением к физической памяти, создавая возможность для запуска программ в *виртуальном адресном пространстве* или просто в виртуальной памяти. Управление защитой памяти, настройка MMU для преобразования виртуальных адресов в физические — всё это возлагается на менеджер оперативной памяти.

8.3.1. Проблемы, решаемые менеджером памяти

Перечислим проблемы, с которыми сталкивается операционная система при управлении оперативной памятью:

- управление аппаратной защитой памяти;
- недостаток объёма памяти;
- дублирование данных;
- перемещение кода;
- фрагментация.

Остановимся на каждом из пунктов подробнее; начнём по порядку с управления защитой памяти. Целью здесь — в защите процессов друг от друга и операционной системы от процессов. В мультизадачной системе, как мы уже не раз отмечали, необходимы аппаратные механизмы защиты памяти. Управление ими возлагается, естественно, на операционную систему, ведь это требует привилегированных действий. На неё же ложится и обязанность по распределению доступной памяти между процессами.

Объёма оперативной памяти может не хватить для размещения ядра и всех процессов. В такой ситуации современные операционные системы высвобождают физическую память, сбрасывая (откачивая) давно не использовавшиеся данные на диск. Когда процессу снова требуются данные из откачанной области памяти, операционная система производит обратную операцию, так что процесс ничего не замечает — происходящее проявляется только в некотором замедлении работы.

Дублирование данных может возникнуть, например, при запуске нескольких копий одной программы: хотя при этом их данные могут различаться, содержимое сегментов кода будет одинаковым. Естественно, такого дублирования желательно избегать.

Код программ может быть привязан к конкретным значениям адресов памяти, в которые загружается программа; например, код может использовать переходы по абсолютным адресам. Вместе с тем, в мультизадачной ситуации заранее не известно, в какое конкретно место физической памяти придется загружать конкретную программу; если привязать машинный код к физическим адресам, именно это место в памяти может оказаться занято другой программой.

Наконец, при постоянном выделении и освобождении блоков памяти разного размера может возникнуть ситуация, при которой очередной блок не может быть выделен, хотя общее количество свободной памяти превышает его размер. Пример такой ситуации показан на рис. 8.3. В некоторый момент мы не можем разместить в памяти задачу №5, потому что нет подходящего свободного блока адресов, хотя общее количество свободной памяти превышает размер новой задачи. С проблемой фрагментации связана проблема увеличения размеров существующей задачи в случае, если ей потребовалась дополнительная память: может оказаться, что память за верхней границей задачи занята и расширять ее некуда.

Заметим, что большинство перечисленных проблем возникает лишь в мультизадачных системах. В самом деле, если система однозадачна, то защищать, вообще говоря, некого и не от чего, дублирование возникнуть не может (задача всего одна), проблемы с перемещением кода не возникают, так как все программы можно грузить в одну и ту же область памяти; по той же причине отсутствует и фрагментация. Остаётся только проблема объёма (для случая одной задачи, не умещающейся в

памяти целиком), но и эта проблема оказывается решаема с помощью оверлейных структур — частей кода, загружаемых и выгружаемых под контролем основной программы, — хотя это и усложняет программирование. В мультизадачной же системе управление оперативной памятью превращается в целый комплекс технических решений, требующих как аппаратной, так и программной поддержки.

8.3.2. Виртуальная память и подкачка

Мы уже рассказывали о виртуальной памяти в самом начале части, посвящённой языку ассемблера (см. т. 2, §3.1.2). Напомним основную идею виртуальной памяти: исполнительные адреса, фигурирующие в машинных командах, считаются не адресами физических ячеек памяти, а некими абстрактными ***виртуальными адресами***. Всё множество виртуальных адресов называется ***виртуальным адресным пространством***. Виртуальные адреса преобразуются центральным процессором — а точнее, уже упоминавшимся устройством MMU в составе процессора — в адреса ячеек памяти (***физические адреса***) по некоторым правилам, причём эти правила могут динамически изменяться.

Использование ***виртуальной памяти*** позволяет эффективно преодолевать проблему перемещения кода, а при использовании достаточно гибкой модели преобразования адресов — облегчает решение проблем дублирования данных, фрагментации и защиты. Обычно для каждой задачи задаются свои правила вычисления физических адресов по виртуальным; перед передачей управления коду задачи операционная система соответствующим образом настраивает MMU. В распоряжении задачи оказывается своё собственное виртуальное адресное пространство, при использовании которого можно никак не учитывать существование других задач. При необходимости задачу (а в некоторых случаях и отдельную её часть) можно перенести в другое место физической памяти, одновременно изменив для этой задачи правила преобразования адресов так, чтобы те же самые виртуальные адреса соответствовали новым физическим.

Некоторые виртуальные адреса могут не иметь соответствия физическим; при попытке задачи обратиться к таким адресам возникает исключение (внутреннее прерывание), в результате которого получает управление операционная система. Дальнейшие действия операционной системы зависят от конкретной ситуации. Если задача попыталась обратиться к памяти, которой у неё нет и никогда не было, это рассматривается как ошибка (нарушение защиты памяти); задача снимается как аварийная. В то же время обращение задачи на запись к памяти по адресу, непосредственно примыкающему к секции стека, указывает на необходимость увеличения этой секции. Наконец, операционная система может, если нужно, убрать часть информации, принадлежащей

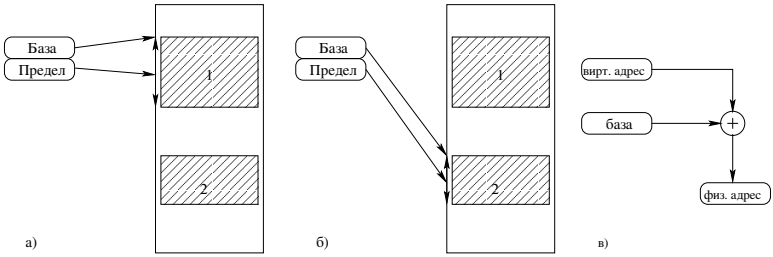

Рис. 8.4. Модель база/предел

задаче, из физической памяти, а при попытке задачи обратиться к этой информации — вернуть всё на место; это называется ***подкачкой***[11].

Подкачку операционные системы используют для решения проблемы нехватки физической памяти. Данные, которые временно не используются (например, принадлежат блокированным задачам), могут быть для временного хранения перенесены на диск («откачаны»), а при возникновении в них потребности — вновь загружены в оперативную память («подкачаны»). Память той или иной задачи может быть откачана на диск целиком, например, если задача блокирована. Кроме того, при использовании достаточно гибкой модели управления памятью возможна откачка отдельных частей памяти задачи; при этом задача может продолжать выполняться. При попытке задачи обратиться к области своей памяти, которая в настоящее время откачана, возникает внутреннее прерывание (в зависимости от используемого процессора это может быть прерывание по защите памяти или специальное ***страничное прерывание***); получив управление в результате этого прерывания, операционная система определяет, что задаче требуется откачанная на диск область виртуальной памяти, и подкачивает соответствующие данные в оперативную память, после чего продолжает выполнение прерванной задачи с того же места, так что задача ничего о факте откачки не знает.

8.3.3. Простейшая модель виртуальной памяти

Снабдим процессор двумя регистрами специального назначения, которые будем называть *базой* и *пределом* (соответствующие английские термины — *base* и *limit*). Для простоты будем считать, что в привилегированном режиме значения этих регистров игнорируются. После перехода процессора в ограниченный режим при выполнении любой команды процессор (на аппаратном уровне) к любому заданному командой исполнительному адресу прибавляет значение базы и уже

[11] Английский термин — *swapping*; данные, *откачанные на диск*, по-английски называются *swapped out*.

результат этого сложения использует в качестве адреса в физической памяти (рис. 8.4). Одновременно с этим процессор сравнивает значение исполнительного адреса с содержимым регистра «предел»; если обнаруживается превышение, процессор отрабатывает внутреннее прерывание «нарушение защиты памяти». Модификация базы и предела при работе процессора в ограниченном режиме запрещена.

Как видим, регистр «база» задаёт адрес, начиная с которого в памяти располагается текущая задача. Исполнительные адреса, задаваемые инструкциями в коде задачи, не совпадают с «настоящими» адресами ячеек памяти, к которым в итоге производится обращение, так что эти адреса можно считать виртуальными. Регистр «предел» задаёт размер блока памяти, доступного текущей задаче. Это позволяет защитить память, находящуюся вне области, выделенной данной задаче, от случайных обращений со стороны текущей задачи.

Адреса в коде задачи теперь формируются в предположении, что задача будет работать в адресном пространстве, начинающемся с нуля. Операционная система может загрузить задачу в любой свободный участок памяти: проблема адаптации программы к адресам решается установкой соответствующих значений базового и предельного регистров. Более того, при необходимости задачу можно переместить в другое место памяти — для этого достаточно скопировать содержимое её области памяти в память по новым (физическим) адресам и изменить соответствующим образом значения базы и предела. Операционная система для передачи управления задаче заполняет базовый и предельный регистры, после чего переключает режим исполнения в ограниченный и передаёт управление коду задачи. При возникновении прерывания[12] ограниченный режим снимается и управление вновь получает код операционной системы, который, в числе прочего, может принять решение о передаче управления другой задаче, для чего достаточно изменить содержимое базового и предельного регистров на соответствующие другой задаче и передать ей управление.

Ясно, что проблемы защиты и адаптации к адресам таким образом решены. С проблемой объёма памяти дела обстоят далеко не так гладко: на диск можно сбросить только целиком всю память той или иной задачи. Кроме того, если объёма физической памяти не хватает даже для одной задачи (то есть нашлась такая задача, потребности которой превышают объём физической памяти), описанная модель выйти из положения не позволяет. Проблема фрагментации решается только путём перемещения задач в физической памяти, что влечёт относительно дорогостоящие операции копирования существенных объёмов данных. Наконец, проблема дублирования не решена вообще.

[12] Имеется в виду прерывание любого из трёх видов; в другой терминологии это будут прерывания, исключения и системные вызовы.

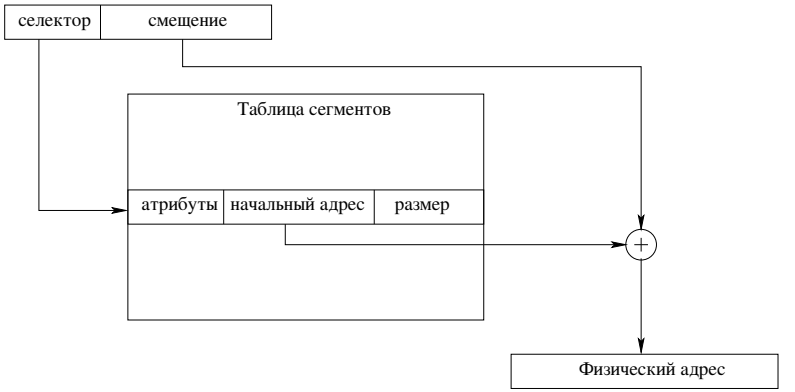

Рис. 8.5. Сегментная организация памяти

8.3.4. Сегментная организация памяти

Усовершенствуем модель «база-предел», для чего введём понятие ***сегмента***. Под сегментом будем понимать область физической памяти, имеющую начало (по аналогии с базой) и длину (по аналогии с пределом). В отличие от предыдущей модели, позволим каждой задаче иметь *несколько* пар база-предел, то есть несколько сегментов.

Конечно, такая смена модели существенно затронет как устройство процессора, так и программное обеспечение, причём если в модели «база-предел» потребовалась поддержка со стороны операционной системы, то в сегментной модели новую архитектуру придётся учитывать и при написании кода пользовательских программ — по крайней мере, если мы захотим писать на языке ассемблера; трансляторы языков высокого уровня все нужные моменты учтут автоматически при переводе нашей программы в исполняемый код.

Итак, первое и наиболее видимое новшество состоит в том, что в исполнительном адресе, то есть в адресе, формируемом тем или иным способом инструкциями процессора, появляется специфическая часть, называемая ***селектором сегмента***. Под него можно отвести несколько старших разрядов адреса или использовать отдельный специально предназначенный для этого регистр процессора. Уместно будет вспомнить о существовании в архитектуре i386 «загадочных» регистров CS, DS, ES, SS, FS и GS, которые мы во втором томе упомянули, но работать с ними не пытались; при использовании сегментной составляющей MMU процессора i386 именно эти (сегментные) регистры используются в роли селекторов сегментов.

Селектор сегмента содержит число, представляющее собой, упрощённо говоря, порядковый номер сегмента в ***таблице дескрипторов сегментов***. Для каждого сегмента эта таблица содержит физический адрес его начала и его длину (то есть базу и предел), плюс к этому

некоторые служебные параметры — например, флаги, разрешающие или запрещающие запись в этот сегмент, исполнение его содержимого в качестве кода и т. п.). Физический адрес ячейки памяти вычисляется путем прибавления адреса начала сегмента, взятого из строки таблицы, выбранной селектором, к смещению, взятому из исполнительного адреса (рис. 8.5).

Возможность завести несколько сегментов для одной задачи позволяет, например, сделать некоторый сегмент общим для двух и более задач, так что теперь мы можем решить проблему дублирования. Некоторые преимущества мы получаем и в отношении проблем объёма и фрагментации. Откачивать на диск по-прежнему требуется сегмент целиком, но при наличии у каждой задачи нескольких сегментов это всё же проще, чем откачивать всю задачу; точно так же при перемещении данных с целью дефрагментации можно перемещать не задачу целиком, а лишь некоторые из её сегментов, и, кстати, найти свободную область подходящего размера для размещения сегмента в среднем проще, чем для всей задачи; впрочем, эти преимущества чисто количественные и всерьёз на ситуацию не влияют.

Кроме того, сегменты в определённых случаях удобны сами по себе. Представьте задачу, требующую нескольких больших таблиц, каждая из которых может увеличиваться в размерах. Если использовать обычную (плоскую) модель памяти, не исключено, что одну из таблиц придётся перемещать, чтобы дать возможность расширить другую; при этом придётся не только проводить копирование, но и пересчитывать все указатели, содержавшие адреса элементов перемещённой таблицы. Всех этих трудностей можно избежать, если под каждую таблицу выделить отдельный сегмент.

Таблица дескрипторов сегментов хранится в оперативной памяти, так что, если не предпринять специальных мер, на каждое обращение активной программы к памяти потребовалось бы ещё одно — за информацией из этой таблицы, что снизило бы производительность системы практически вдвое. Поэтому разработчики архитектуры процессоров организуют хранение информации об используемых сегментах непосредственно в процессоре. Например, процессоры серии Intel загружают информацию из таблицы дескрипторов каждый раз при изменении содержимого сегментного регистра; информация из соответствующей строки таблицы дескрипторов загружается в «невидимую» часть сегментного регистра и хранится там до следующего его изменения.

8.3.5. Страничная организация памяти

Более эффективно решить проблемы объёма и фрагментации позволяет *страничная* организация памяти. Отметим сразу, что в этой модели обычно каждая задача имеет свое собственное пространство

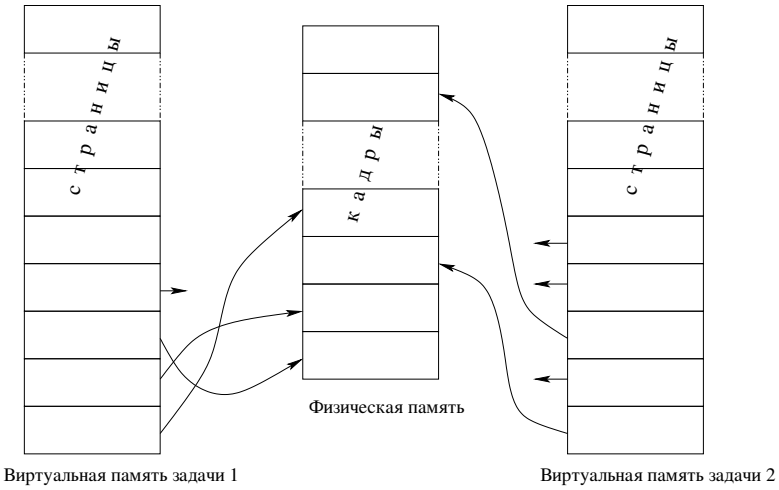

Рис. 8.6. Страничная организация памяти

виртуальных адресов и свои таблицы для перевода их в адреса физические.

Разделим физическую память на ***кадры***[13] фиксированного размера, а пространство виртуальных адресов — на ***страницы*** того же размера (рис. 8.6). Если, к примеру, наш процессор использует 32-битные виртуальные адреса, а физической памяти в компьютере установлено 8Gb (2^{33} байт), мы можем разделить физическую память на $2^{21} = 2097152$ кадров по $2^{12} = 4096$ байт, или 4Kb каждый; виртуальное адресное пространство тогда разделится на $2^{20} = 1048576$ страниц такого же размера. Размер страницы и кадра всегда составляет степень двойки (2^n; в нашем примере $n = 12$); младшие n разрядов адреса — как виртуального, так и физического — задают смещение относительно начала страницы или кадра, а остальные биты — номер страницы или кадра.

Остаётся каким-то образом для каждой задачи сопоставить *некоторым* (не всем!) её страницам номера физических кадров. Важно, что сопоставление производится в произвольном порядке, то есть двум соседним страницам могут соответствовать кадры из совершенно разных областей физической памяти. Обычно отображение страниц на физические кадры осуществляется через ***таблицу страниц***, принадлежащую активной задаче (рис. 8.7). Для каждой страницы таблица содержит запись, состоящую из номера кадра и служебных атрибутов. **В число атрибутов страницы обычно входит так называемый *признак***

[13] Английский оригинал термина «кадр» в данном случае — *frame*; некоторые русскоязычные авторы пользуются вместо слова «кадр» словом «фрейм», либо используют слово «страница» как для обозначения страниц виртуальных адресов, так и для обозначения кадров физической памяти.

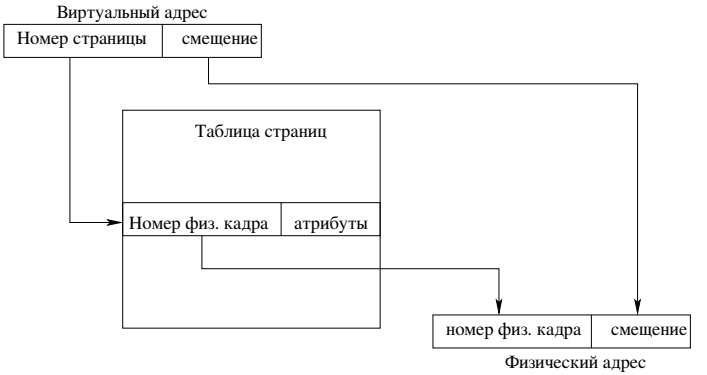

Рис. 8.7. Страничное преобразование адреса

***присутствия*, означающий, находится ли данная страница в настоящее время в оперативной памяти или нет. При попытке обращения к странице, для которой признак присутствия сброшен, процессор инициирует исключение (внутреннее прерывание), называемое *страничным*; получив управление, операционная система производит, если возможно, подкачку соответствующей страницы с диска**.

Существует проблема выбора размера страницы. Чем больше страница, тем больше памяти пропадает впустую в последних страницах задач. Так, если выбрать размер страницы 1 Mb, то задача, занимающая чуть больше 1 Mb, займёт два физических кадра, причём второй почти весь (то есть почти 1 Mb) не будет использоваться. С другой стороны, чем меньше размер страницы, тем больше количество самих страниц и тем, соответственно, больше размеры страничных таблиц. Процессоры i386 работали со страницами размером 4 Кб; более поздние процессоры линейки x86, начиная с Pentium, умеют работать со страницами размером 4 Kb, 2 Mb и 4 Mb. Размер страницы 4 Kb (2^{12} байт) можно считать наиболее популярным среди различных архитектур.

Обратим внимание, что при 32-битной адресации (4 Gb адресуемого пространства) количество страниц составит 2^{20}, то есть больше миллиона. Если учесть, что каждая строка в таблице страниц занимает обычно 4 байта, получим, что на таблицу страниц каждого процесса требуется 4 Mb памяти. Безусловно, это неприемлемо: большинство задач занимает в памяти существенно меньше одного мегабайта, так что на таблицы страниц придётся потратить больше памяти, чем на сами задачи. Одним из возможных решений могло бы стать искусственное ограничение количества страниц, но это решение далеко не лучшее, поскольку приводит к невозможности выполнения задач, требующих бóльшего количества памяти.

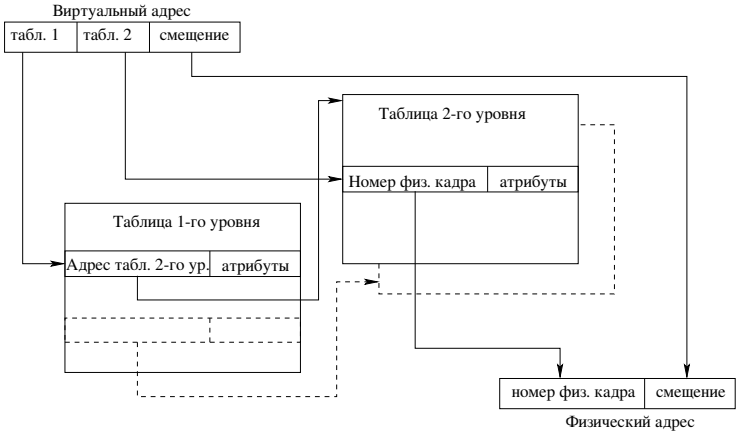

Рис. 8.8. Двухуровневая схема страничного преобразования

Более элегантное решение состоит в применении ***многоуровневых страничных таблиц***. В этом случае номер страницы делится на группы битов, задающих номер строки в таблицах соответствующих уровней; таблица первого уровня содержит адреса таблиц второго уровня, и т. д., а таблица последнего уровня — номера физических кадров (рис. 8.8). При описании двухуровневой схемы таблицу первого уровня часто называют ***каталогом страничных таблиц***, а таблицы второго уровня — просто страничными таблицами, без уточнения насчёт уровня[14]. Для каждой задачи при использовании такой схемы необходимо завести таблицу верхнего уровня и по одной таблице для всех остальных уровней; остальные таблицы будут добавляться по мере надобности. Поскольку схема, как правило, содержит не более трёх уровней[15], для небольшой задачи требуется хранить всего две или три сравнительно небольшие таблицы.

Рассмотрим для примера двухуровневую страничную схему для случая 32-битных адресов и размера страницы 4 Kb. Смещение при таком размере страницы занимает 12 бит, на номер страницы остаётся 20 бит, из которых 10 используется для выбора строки в таблице первого уровня, остальные 10 — для выбора строки в таблице второго уровня. Именно так обстоят дела в i386-совместимых процессорах; в документации обычно таблица первого уровня называется каталогом таблиц, а таблицы второго уровня — собственно таблицами страниц. Тогда таблицы обоих уровней могут содержать $2^{10} = 1024$ строки, что

[14] В литературе можно встретить противоположный подход к нумерации уровней: таблицами первого уровня называют таблицы, содержащие номера физических кадров, таблицами второго уровня — таблицы, содержащие адреса таблиц первого уровня и т. д. На самом деле это не более чем вопрос принятой терминологии.

[15] Обычно два уровня для 32-битных архитектур и три — для 64-битных.

при длине строки в 4 байта требует 4096 байт, то есть каждая таблица занимает ровно один кадр в памяти.

Как можно заметить, страничная организация памяти снимает проблемы объёма и фрагментации. Действительно, в рассмотренной модели физическая память выделяется кадрами, причём совершенно безразлично, будут ли кадры соседними или нет; любой свободный кадр может быть использован для любой задачи. Фрагментация здесь просто не может возникнуть, ей неоткуда взяться. Каждая страница независимо от других может быть в любой момент откачана на диск. При обращении к ней произойдёт страничное прерывание, по которому операционная система может подкачать страницу обратно в память, после чего продолжить выполнение задачи в точности с инструкции, вызвавшей прерывание. Возможно, страница будет подкачана в совершенно другой физический кадр, но пользовательская задача этого не заметит, как и вообще самого факта подкачки. Благодаря независимой обработке отдельных страниц можно выделить для отдельно взятой задачи больше памяти, чем физически есть на машине; в этом мы ограничены только объёмом виртуального адресного пространства и, конечно, дисковым пространством, выделенным под откачку (своппинг).

Отметим, что разрядности физического и виртуального адресов не обязаны совпадать, причём физический адрес может быть как больше, так и меньше адреса виртуального. Разрядность физического адреса совпадает с количеством дорожек на шине адресов (см. т. 1, § 1.3.1) и определяет максимально возможное количество физической памяти, которое можно к этой шине (и к этому процессору) подключить. В зависимости от процессора виртуальное адресное пространство может быть как меньше, так и больше этого предельного количества физической памяти.

К недостаткам страничной модели можно отнести, во-первых, сравнительно большое количество неиспользуемой памяти в конце последней страницы каждой задачи, особенно при больших размерах страниц, и, во-вторых, большие объёмы служебной информации — страничных таблиц. Остановимся на этом подробнее. Если не принять специальных мер, при работе по двухуровневой табличной схеме каждое обращение к памяти за командой или данными потребует ещё двух обращений: к таблице первого уровня и к таблице второго уровня. Ситуация, в общем, аналогична возникшей в сегментной модели, за исключением того, что страница — не сегмент, её объём существенно меньше, а количество страниц — гораздо больше; программа может работать с сотнями, тысячами, десятками тысяч страниц. Ясно, что хранить информацию о страничном соответствии в регистровой памяти не получится из-за большого объёма.

Решить проблему позволяет встроенное в процессор специальное устройство, называемое ***ассоциативной памятью***[16]; оно представляет собой таблицу из двух полей — номер виртуальной страницы и информация о соответствующем ей кадре (номер кадра и его атрибуты). Электронная схема ассоциативной памяти устроена так, что запрос к ней формируется не по номеру строки таблицы, а по *значению первого поля*, то есть по ключу. Сличение предъявленного значения со значениями ключевого поля производится одновременно во всех строках таблицы (параллельными схемами). Если в одной из строк значение совпадает, в качестве результата выдаётся второе поле той же строки, то есть номер кадра и его атрибуты.

При обращении к странице, о которой в ассоциативной памяти нет информации, фиксируется так называемый *промах* и производится преобразование через страничные таблицы (то есть к страницам всё-таки приходится обратиться), но после этого информация, полученная из таблиц, записывается в ассоциативную память. Если теперь программа обратится к той же самой странице, читать содержимое таблиц уже не потребуется. Когда все строки таблицы оказываются заполнены, очередное обращение к неизвестной странице приводит к тому, что информация об одной из известных страниц из ассоциативной памяти удаляется. Количество строк TLB варьируется в достаточно широких пределах: можно встретить процессоры с TLB на восемь строк и на несколько тысяч. В любом случае разработчики процессоров стараются сделать TLB достаточно большим, чтобы удерживать вероятность промаха в пределах долей процента.

Следует отметить, что информация об отображении страниц локальна для каждой задачи, так что при смене активной задачи содержимое ассоциативной памяти приходится сбрасывать, и некоторое время новая задача тратит на заполнение ассоциативной памяти своими данными. Это ещё сильнее увеличивает и без того высокую стоимость переключения контекста, но выбора у нас нет.

Отметим также, что управление ассоциативной памятью можно возложить на операционную систему. В этом случае процессор вообще не делает никаких предположений о том, как устроены структуры данных, отвечающие за отображение страниц; это полностью отдано на откуп операционной системе. В случае, если в ассоциативной памяти отсутствует строка для требуемого номера страницы, процессор генерирует исключение (внутреннее прерывание), в ответ на которое операционная система должна программно произвести поиск или иное вычисление соответствующей информации и занести эту информацию в ассоциативную память, после чего возобновить исполнение активной задачи. Достоинством такого подхода можно считать очевидную гиб-

[16] В англоязычной литературе обычно используется термин «*Translation Lookaside Buffer*», TLB, а в русскоязычной можно встретить термин «буфер быстрого преобразования адреса».

кость применяемой модели виртуальной памяти: операционная система может применять для хранения сведений о виртуальных страницах любые структуры данных, какие сочтёт нужным; с другой стороны, цена «промаха» оказывается много выше из-за вынужденного переключения контекста при передаче управления в ядро ОС.

8.3.6. Сегментно-страничная организация памяти

Как говорилось ранее, сегментная модель организации памяти имеет свои специфические достоинства, делая адресное пространство задачи многомерным. Сегментно-страничная модель представляет собой попытку объединить достоинства сегментной и страничной организации памяти. В этой модели виртуальный исполнительный адрес претерпевает двойное преобразование: сначала из адреса выделяется сегментная часть, которая с использованием таблицы дескрипторов сегментов превращается в «плоский» (но всё ещё виртуальный) адрес; полученный адрес подвергается страничному преобразованию, результатом которого становится физический адрес. Иначе говоря, в сегментно-страничной модели виртуальной памяти каждая задача имеет своё виртуальное страничное адресное пространство, внутри которого могут быть организованы сегменты.

Сегментно-страничная организация реализована на i386-совместимых процессорах, но больше, собственно говоря, нигде не встречается. Основной недостаток этой модели — её чрезмерная сложность; большинство операционных систем этой моделью не пользуется. Свою роль здесь играет и то обстоятельство, что операционную систему, которая задействовала бы сегментно-страничную модель виртуальной памяти во всём её великолепии, нельзя было бы перенести на другие процессоры, где страничное преобразование адресов присутствует (оно в наше время присутствует практически на всех процессорах, имеющих MMU), а сегментное отсутствует. Изучая программирование на языке ассемблера, мы видели, что пользовательские задачи под управлением ОС Linux и FreeBSD «живут» в плоской модели памяти; операционная система создаёт несколько «сегментов», «накрывающих» всё страничное пространство целиком, и дескрипторы этих сегментов заносит в сегментные регистры, на чём и успокаивается. Трогать содержимое сегментных регистров задаче не рекомендуется, ни к чему хорошему это не приведёт.

8.4. Управление аппаратурой; ввод-вывод

8.4.1. Две точки зрения на ввод-вывод

Схематически ввод-вывод (или, говоря шире, ***управление устройствами***) показан на рис. 8.9. Центральный процессор и оперативная

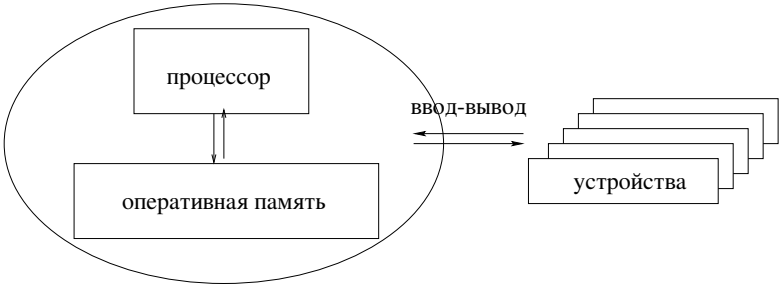

Рис. 8.9. Ввод-вывод с точки зрения аппаратуры

память занимают в этой схеме несколько особое место: несмотря на то, что и процессор, и память, несомненно, являются техническими устройствами, они не входят в число устройств, об управлении которыми идёт речь. Иногда во избежание путаницы говорят об управлении ***внешними устройствами***, при этом подразумевается, что процессор и память — устройства «внутренние», хотя так их и не называют. Считается, что взаимодействие между процессором и памятью не входит в понятие ввода-вывода; с точки зрения, принятой при обсуждении аппаратного обеспечения (в том числе среди программистов, создающих ядра операционных систем), ЦП и память представляют собой единый конструктив, а вводом-выводом считается обмен информацией между этим конструктивом и всем остальным миром.

На то, что считать, а что не считать вводом-выводом, существует и иная точка зрения, которая принята среди прикладных программистов. Эта точка зрения могла бы полностью совпадать с предыдущей, если бы не тот факт, что одна и та же программа, выполняя одни и те же действия, в зависимости от обстоятельств, в которых её запустили, может как осуществлять аппаратный ввод-вывод, так и не осуществлять его[17]. Рассмотрим для примера хорошо знакомую нам программу «Hello, world». Если вызвать её командой

```
avst@host:~$ ./hello > file1
```

то строка `"Hello, world!"` будет выведена в файл `file1` на диске, то есть произойдёт аппаратный ввод-вывод. Если же *та же самая программа* будет запущена командой

```
avst@host:~$ ./hello | ./prog2
```

то никакого ввода-вывода (с аппаратной точки зрения) не последует: строка, выведенная программой `hello`, будет подана на вход программе `prog2`, так что всё взаимодействие, скорее всего, останется в рамках системы «процессор-память».

[17] Имеется в виду, естественно, ввод-вывод через системные вызовы.

Если использовать «аппаратную» терминологию, программистам придётся в каждой такой ситуации уточнять, что речь идёт не о вводе-выводе, а о его *возможности* или *высокой вероятности* (или низкой, зависит от задачи), и такое уточнение потребуется едва ли не в каждом месте, где речь идёт о вводе-выводе. Очевидно, что такой вариант терминологии оказывается неудобен достаточно, чтобы его никто не использовал. Поэтому прикладные программисты обычно под вводом-выводом понимают любой обмен информацией с внешним миром через потоки ввода-вывода, то есть через вызовы `read`, `write`, `send`/`recv` и т. п. Конечно, такое «определение» ввода-вывода очевидным образом порочно, ведь оно фактически сводится к тому, что «ввод-вывод — это то, что делается через вызовы для ввода-вывода»; с почти таким же успехом можно было бы заявить, что ввод-вывод — это ввод-вывод.

Можно было бы сказать, что вводом-выводом с программистской точки зрения считается вообще любой обмен информацией между процессом и внешним миром, но это не так: процесс может как принимать, так и получать информацию, например, через сигналы или через разделяемую память, и программисты традиционно не рассматривают всё это как ввод-вывод. Остаётся лишь констатировать, что техническая предметная область в очередной раз демонстрирует своё нежелание соответствовать каким бы то ни было определениям.

8.4.2. Взаимодействие ОС с аппаратурой

Мы уже обсуждали (см. т. 1, §1.3.3), что центральный процессор взаимодействует с остальными частями компьютера через общую шину, а для подключения к этой шине разных внешних устройств используются ***контроллеры***[18]. Необходимость в контроллере как своего рода посреднике между процессором и устройством обусловлена потребностью в унификации способов подключения периферийных устройств к компьютеру. Очевидно, что никакой из выводов («ножек») процессора не может соответствовать конкретным действиям с внешними устройствами, таким как перемещение читающих головок, запуск или останов моторов и т. п.: во-первых, различных действий такого рода слишком много, а выводов у процессора — ограниченное количество, и, во-вторых, такое построение архитектуры ограничило бы ассортимент устройств, которыми может управлять данный конкретный процессор — например, к процессору нельзя было бы подключить никакое устройство, созданное позже данного процессора и, как следствие, не предусмотренное его конструкцией. Контроллеры эту проблему решают: благодаря им от процессора оказываются скрыты схематические особенности различных

[18] Здесь будет уместно напомнить, что основной перевод английского глагола *to control* на русский язык — *управлять*, а вовсе не «контролировать», как часто ошибочно полагают.

устройств, то есть с точки зрения процессора устройства различаются между собой разве что номерами.

Контроллер представляет собой электронную логическую схему, подключаемую с одной стороны к общей шине компьютера, что позволяет обмениваться данными с процессором, а с другой стороны — к физическому устройству, которым нужно управлять. Поскольку контроллер создается всегда для управления конкретным устройством, для него не составляет проблемы генерировать именно такие электрические сигналы, которые нужны для управления данным устройством. С другой стороны, поскольку любой контроллер взаимодействует с процессором путём приёма и передачи определённых числовых значений по шине, процессор может быть запрограммирован на работу с любым контроллером, который можно физически подключить к общей шине данного компьютера. Благодаря этому процессор может работать и с устройствами, которых на момент выпуска данного процессора ещё не существовало.

Поскольку к общей шине может быть подключено одновременно большое количество различных контроллеров, нужно каким-то образом различать, кому предназначены передаваемые данные или, наоборот, кто должен ответить на запрос данных. Для этого вводится понятие ***порта ввода-вывода***. Порт ввода-вывода представляет собой абстракцию, имеющую адрес, представимый на данной шине. Для каждого порта возможны (с точки зрения процессора) операции *записи* и *чтения*, то есть, соответственно, передачи значения по заданному адресу и запроса значения с заданного адреса. Диапазон возможных значений, как и диапазон адресов портов, зависит от архитектуры шины (определяется разрядностью соответственно шины адресов и шины данных). За каждым контроллером числится один или несколько (а иногда целая область) портов ввода-вывода, причём возможно, что некоторые из них могут быть только прочитаны, а некоторые — только записаны.

Можно заметить, что порт ввода-вывода похож на обыкновенную ячейку памяти. Более того, в некоторых архитектурах, как мы увидим позже, порты ввода-вывода располагаются в том же адресном пространстве, что и обычная оперативная память, и читаются/записываются теми же командами процессора. Тем не менее, порт ввода-вывода ячейкой памяти не является, поскольку в большинстве случаев ничего не хранит, а значения, читаемые из порта, обычно отличаются от значений, туда заносимых, не говоря уже о том, что многие порты доступны только для чтения или, наоборот, только для записи. Значение, записываемое в порт, может представлять собой, например, код команды включения мотора жёсткого диска, а значение, из того же порта читаемое, — код, по которому можно определить, завершена ли последняя операция чтения.

Кроме портов, некоторые контроллеры имеют ещё и *буферы ввода-вывода*, представляющие собой оперативную память, конструктивно

интегрированную в контроллер и предназначенную для обмена массивами информации между контроллером и центральным процессором. Например, данные, предназначенные для записи на диск, нужно скопировать в буфер контроллера этого диска, а затем дать через порт вывода команду на запись; наоборот, при чтении информации с диска она помещается в буфер контроллера, откуда её можно потом скопировать в основную память. Другим примером буфера ввода-вывода можно считать видеопамять, то есть память, в которой хранится изображение, видимое на экране дисплея.

На некоторых процессорах порты ввода-вывода имеют отдельное адресное пространство (как правило, меньшей разрядности, чем пространство адресов оперативной памяти). В этом случае для работы с портами используются отдельные инструкции процессора (например, `IN` и `OUT` вместо `MOV`).

Альтернативное решение — разместить контроллеры устройств в том же адресном пространстве, что и оперативную память. В этом случае процессору не нужны отдельные инструкции для работы с портами, все делается обычной командой `MOV`. Такая схема была, например, реализована на PDP-11. Одно из достоинств такой схемы — возможность сопоставить области портов ввода-вывода виртуальным адресам некоторых процессов, предоставив им возможность взаимодействия с устройствами напрямую, без посредства операционной системы. Это позволяет вынести драйверы отдельных устройств из ядра в пользовательские процессы. Отсутствие специальных инструкций для взаимодействия с портами делает возможным написание драйверов на языках высокого уровня (таких как Си++) без применения вставок на языке ассемблера.

К недостаткам единого адресного пространства можно отнести, например, тот факт, что на порты расходуется основное адресное пространство, которого может быть не так много. На PDP-11 эта проблема была достаточно актуальна, т. к. адреса на этой машине были 16-разрядные. Кроме того, на современных процессорах применяется кеш-память, так что участки адресного пространства, соответствующие буферам и портам, из процесса кеширования приходится искусственно исключать. Наконец, оперативная память и контроллеры устройств представляют собой сущности весьма различные, и поместить их на одну общую шину может оказаться затруднительно, а построение двух разных шин при использовании одного общего пространства адресов приводит к необходимости на том или ином уровне принимать решение, по какой шине следует работать с конкретным адресом.

Ещё один возможный подход, применяемый в хорошо знакомой нам архитектуре i386 — комбинированный. Порты ввода-вывода в этом случае размещаются в отдельном адресном пространстве, а буферы — в общем.

8.4.3. Драйверы

Под ***драйвером*** исходно понималось некое *программное средство*, берущее на себя заботу об особенностях управления определённым внешним устройством. Напомним, что в задачи операционной системы входит *управление аппаратурой*, которое подразумевает *абстрагирование* и *координацию*; можно считать, что абстрагирование как раз и обеспечивается драйверами — они предоставляют некий унифицированный набор программных функций, позволяющих управлять внешним устройством, не зная, как это на самом деле делается: соответствующее знание разработчики драйвера вложили в реализацию его функций.

Стоит обратить внимание, что мы назвали драйвер каким-то «программным средством», а не просто «программой». Этому есть вполне внятная причина: строго говоря, драйвер не является программой, он представляет собой набор подпрограмм (функций), пусть даже связанных между собой, но именно подпрограмм, рассчитанных на то, что вызывать их будет кто-то ещё — в большинстве случаев имеются в виду другие подсистемы ядра ОС. В этом плане драйвер своей структурой напоминает скорее библиотеку, чем программу, хотя по назначению драйвер, конечно, далеко не библиотека. Если говорить о *программе*, то программой, работающей с внешними устройствами, как можно догадаться, является ядро ОС, ну а драйвер с момента его загрузки становится (обычно) частью ядра.

Зная, как нужно работать с данным конкретным аппаратным устройством, драйвер скрывает особенности этого устройства от всей остальной системы. Другие части ядра обращаются к драйверу, вызывая его внешние функции — так называемые ***точки входа***, которые имеют одинаковую структуру для всех драйверов определённой категории. Так, обращение к драйверу жёсткого диска с целью записать сектор №55 и такое же обращение к драйверу flash-брелка будет (с точки зрения любой подсистемы ядра, кроме самих этих драйверов) выглядеть одинаково, несмотря на то, что эти устройства на аппаратном уровне управляются совершенно по-разному. Детали процесса управления конкретными устройствами оказываются локализованы в коде драйвера. При написании остальных частей операционной системы, а также при создании пользовательского программного обеспечения становится возможно их не учитывать; именно так и достигается искомое *абстрагирование*.

В большинстве случаев драйвер рассчитан на выполнение в привилегированном режиме, то есть должен быть частью ядра; привилегированные команды нужны драйверу для обращения к контроллеру (то есть к портам ввода-вывода) через шину. Теоретически возможно выполнение драйвера и в пользовательском режиме в виде обычного процесса; при этом драйвер должен сообщать операционной системе, какое число в какой порт следует отправить и из какого порта прочитать ответ, а ядро уже выполняет эти действия и сообщает драйверу о результатах.

Необходимость переключения контекста между процессом драйвера и ядром может существенно сказаться на эффективности работы, так что применяется такой подход достаточно редко. Отметим, что при этом драйвер всё-таки превращается в обычную программу, хотя и имеющую специфическое назначение.

Вернёмся к наиболее распространённой ситуации, когда драйвер работает в виде части ядра операционной системы. Можно выделить три основных способа помещения драйвера в ядро: включение кода драйвера в качестве модуля на этапе сборки ядра, подключение драйвера на этапе загрузки операционной системы и динамическая загрузка модулей в ядро.

Если код драйвера включается в ядро в качестве модуля прямо при его сборке, то добавление или удаление драйвера требует пересборки (перекомпиляции) ядра и перезагрузки операционной системы. При этом пользователю должны быть доступны исходные тексты ядра либо, как минимум, его объектные модули и соответствующие им интерфейсные файлы. Такой подход считается традиционным для операционных систем семейства Unix, хотя в последние 10-15 лет он едва ли не полностью вытеснен из реальной практики динамической загрузкой модулей.

Подключение драйвера на этапе загрузки операционной системы предполагает, что в системе имеется файл, содержащий список драйверов, и сами драйверы в виде отдельных файлов. При загрузке ядро анализирует список и подключает драйверы, после чего начинает работу. В этом случае для подключения дополнительного драйвера достаточно перезагрузить систему; перекомпиляция ядра не требуется. Традиционно этот подход использовали системы семейства Windows, а ранее — и MS-DOS.

Динамическая загрузка модулей в ядро позволяет добавлять драйверы в уже работающую систему без её перезагрузки. Для этого достаточно подготовить файл драйвера, соблюдающий определенные соглашения об именах, и выдать ядру соответствующий системный вызов, после чего модуль становится частью ядра. Ненужные модули обычно можно из ядра изъять. Такая возможность присутствует практически на всех современных Unix-системах, включая Linux и FreeBSD. Стоит отметить, что при высокой гибкости эта модель считается небезопасной, т. к. фактически позволяет при наличии прав системного администратора запустить произвольный код в привилегированном режиме.

Конкретный набор «внешних функций» (точек входа), которые драйвер предоставляет остальной системе, составляет ***интерфейс драйвера***. Абстрагирование достигается тем, что драйверы совершенно разных устройств могут реализовывать один и тот же интерфейс, то есть иметь (и предоставлять остальной системе) внешне полностью одинаковые наборы точек входа. Остальным подсистемам ядра при этом становится всё равно, с каким устройством в действительности происходит работа,

ведь эта работа для разных устройств требует вызова одинаковых функций. Обращение к точкам входа обычно производится через указатели на функции (см. т. 2, §4.13.2). Драйвер *экспортирует* свои точки входа, то есть сообщает остальному ядру об их существовании, предоставляя некую таблицу их адресов; это может быть или массив указателей на функции, или (чаще) некая структура с полями-указателями.

Конечно, всё многообразие внешних устройств не позволяет придумать единый интерфейс драйвера, который подходил бы вообще для любого устройства. Обычно операционная система поддерживает несколько ***типов устройств***; каждый такой тип есть не что иное, как фиксированный набор точек входа, которые должен предоставлять соответствующий драйвер.

Наличие у драйверов унифицированного интерфейса — фиксированного набора точек входа — открывает одну интересную возможность. Если в систему поместить драйвер, реализующий все положенные точки входа, то с точки зрения всего остального ядра, а также, естественно, с точки зрения пользовательских программ в системе появится новое устройство; как именно оно будет реализовано — никого, кроме собственно драйвера, не касается. Это позволяет заводить в системе ***виртуальные устройства*** — такие, за которыми на самом деле вообще нет ничего аппаратного; их функциональность полностью реализуется драйвером без обращений к каким-либо реальным устройствам. Классическим примером такого устройства можно считать ***виртуальный диск*** (англ. *virtual disk* или, чаще, *RAM disk*). Драйвер такого «диска» при старте выделяет себе определённый объём оперативной памяти и эмулирует операцию записи сектора путём размещения записываемой информации в этой памяти, а операцию чтения сектора — путём копирования информации из памяти. Виртуальный диск со всех возможных точек зрения выглядит как обычный диск, его можно отформатировать (то есть создать на нём файловую систему) точно так же, как это делается с обычными дисками, и смонтировать на выбранную точку монтирования.

Виртуальные устройства не следует путать с ***логическими устройствами***, несмотря на несомненную схожесть этих двух понятий. Как виртуальные, так и логические устройства представляют собой реализованную программно абстракцию «устройства, которого на самом деле нет»; но если виртуальное устройство — это имитация физического устройства программными средствами (с помощью драйвера, который на самом деле не управляет никаким физическим устройством, а вместо этого занимается программной эмуляцией), то логические устройства — это дополнительный уровень абстрагирования, реализованный *поверх* «настоящих» устройств вместе с их драйверами. Например, физический диск может быть поделен (*размечен*) на несколько ***разделов*** (англ. *partitions*), каждый из которых будет представлен

в системе как отдельный *логический* диск; в то же самое время физический диск можно не делить на разделы, используя его как единое целое, и в этом случае можно будет сказать, что логическое представление диска совпадает с его физическими параметрами.

При эксплуатации серверных компьютерных систем часто используют так называемые RAID-массивы[19]; в этом случае ситуация противоположна — один логический диск оказывается реализован на основе (*поверх*) двух или более (вплоть до нескольких десятков) физических дисков. Наконец, многие системы (включая Linux и FreeBSD) позволяют представить в виде дискового устройства обыкновенный файл; обычно эта возможность называется *loopback device*. Здесь тоже речь идёт о логическом диске, а не о виртуальном, поскольку нельзя сказать, что такого диска «на самом деле нет»: он есть, просто он на самом деле не диск, но при этом программный слой, отвечающий за логические устройства, представил его в виде диска.

Отметим, что *все* устройства, видимые пользовательским программам, следует считать именно логическими. Даже если созданная ядром системы абстракция полностью отвечает физическим свойствам «настоящего» устройства, всё равно через системные вызовы нам доступна именно логическая абстракция, которая может быть (в очень редких случаях) «прозрачной», то есть проявлять ровно такие свойства, которыми обладает реальное физическое устройство.

Некоторую путаницу создаёт тот факт, что термин «драйвер устройства» применяют как к драйверам физических устройств (их мы уже обсуждали), так и к компонентам «логического» слоя ядра, которые отвечают за формирование абстракций логических устройств. Драйверы логических устройств по своей сути очень похожи на драйверы устройств физических: это тоже некие наборы функций, включающие унифицированные наборы точек входа. Мы уже обсуждали (см. §5.3.5), что файлы устройств в системах семейства Unix делятся на два типа: ***символьные*** (или ***потоковые***) и ***блочные***; эти два типа файлов устройств как раз и соответствуют двум «классическим» типам драйверов *логических устройств* в Unix. Наборы точек входа у драйверов символьных устройств и у драйверов блочных устройств различны; при этом любой драйвер символьного устройства должен иметь такой набор точек входа, какой предусмотрен в данной системе для символьных устройств; то же самое будет верно для блочных устройств и вообще для драйверов устройств того или иного типа.

Помимо символьных и блочных устройств, система может поддерживать и другие типы драйверов логических устройств. Так, в ОС Linux поддерживается третий тип логического устройства — сетевые интерфейсы. Как мы знаем, сетевые карты в Linux не имеют представления

[19] Название образовано как сокращение от *Redundant Array of Independent Disks* — избыточный массив независимых дисков.

в виде файлов, так что к ним нельзя обратиться с помощью обычных системных вызовов, таких как `read`, `write` или `ioctl`; естественно, интерфейс драйвера сетевой карты или другого сетевого интерфейса «выглядит» с точки зрения всей остальной системы совершенно не так, как диск или простой коммуникационный порт, то есть набор точек входа — функций, вызываемых из других частей ядра — для драйвера сетевого интерфейса не такой, как для драйверов других типов.

Интересно отметить, что создатели ядра FreeBSD в какой-то момент решили отказаться от поддержки блочных устройств, исходя из несколько неожиданных соображений. При обсуждении файлов устройств в §5.3.5 мы назвали основным отличием между символьными и блочными устройствами возможность позиционирования, то есть применимость вызова `lseek`. Авторы ядра FreeBSD к этому вопросу подходят иначе: с их точки зрения основной особенностью блочных устройств следует считать их буферизацию в памяти ядра, а это, в частности, означает, что данные, полученные ядром системы через вызов `write` для записи на блочное устройство, могут неопределённо долго находиться в памяти ядра, не попадая при этом на физический диск, и этот аспект никак не контролируется системными вызовами. К обсуждению буферизации мы вернёмся позже. Так или иначе, в ядре FreeBSD блочные устройства как отдельная сущность отсутствуют, а для драйверов символьных устройств предусмотрена опциональная возможность отработки вызова `lseek` — естественно, не для всех существующих устройств. Создатели Linux пошли другим путём, предусмотрев при открытии файла устройства дополнительный флаг для вызова `open` — `O_DIRECT`; этот флаг позволяет обойти слой буферизации.

Программный слой, создающий логические абстракции, не всегда представляет их в форме «устройства». Так, ***виртуальная файловая система***, который мы позже посвятим отдельный параграф, позволяет монтировать файловые системы, не связанные ни с каким устройством — ни физическим, ни виртуальным, ни даже логическим. Дело в том, что реализация файловой системы в ядре тоже представляет собой фиксированный набор функций, подобных точкам входа драйвера; обычные файловые системы реализуют свои функции через обращения к структурам данных на диске, но никто не мешает реализовать их каким-то иным способом.

Поскольку виртуальные устройства представляют собой имитацию физических устройств, их место в архитектуре системы расположено рядом с физическими устройствами; это позволяет выстраивать логические устройства и другие логические абстракции как поверх физических, так и поверх виртуальных устройств.

Выше мы приводили примеры, связанные с дисками; системы семейства Unix поддерживают также и потоковые (символьные) устройства, не имеющие под собой физического содержания. С некоторыми из них мы уже знакомы (см. §5.3.5, стр. 63): это `/dev/null`, `/dev/zero`, `/dev/full` и `/dev/random`. Вопрос, считать ли эти устройства «логическими» или «виртуальными», относится скорее к области философии; в литературе встречаются оба термина. Если следовать терминологии,

Рис. 8.10. Ввод-вывод на разных уровнях вычислительной системы

введённой нами выше, перечисленные устройства правильнее будет отнести к логическим, поскольку они реализуются в том слое ядра системы, который отвечает за логические устройства.

8.4.4. Ввод-вывод на разных уровнях ВС

Как известно, под ***вычислительной системой*** понимается компьютер вместе со всеми программами, которые на нём выполняются. Припомнив «слоёное» описание ядра операционной системы, начатое в §8.1.4, мы можем заметить, что аналогичным образом на отдельные слои можно разделить не только ядро, но и всю вычислительную систему — и аппаратуру, и пользовательские программы. Конечно, любое такое деление остаётся условным, поскольку реальность обычно ни в какие абстрактные схемы не укладывается; но если некая абстракция позволяет нам лучше понять происходящее, то она полезна хотя бы этим, несмотря на неполное соответствие реальности.

Аппаратуру мы делить на слои не будем, хотя такое деление, безусловно, тоже возможно. Для нужд нашего изложения важнее структура программного обеспечения, как внутри ядра системы, так и за его пределами. Из пользовательского кода мы выделим в отдельный слой библиотеки — ту их часть, которая включает функции-обёртки системных вызовов, а также функции, так или иначе к системным вызовам обращающиеся. Поскольку мы намерены рассматривать ввод-вывод, внутри ядра отметим ***виртуальную файловую систему***, которая, помимо прочего, поддерживает *потоки ввода-вывода* как объекты ядра; кроме того, нас, естественно, заинтересует уровень, создающий абстракции логических устройств, и уровень драйверов физических устройств.

Компоненты каждого слоя, получив «сверху» соответствующее обращение, переводят его в термины следующего уровня и передают ниже, а полученный ответ переводят, наоборот, в термины уровня предыдущего и отправляют наверх. По мере движения от аппаратуры к пользователю нарастает уровень абстрагирования, а сложность описания падает.

Рассмотрим для примера запуск программы с выводом информации в файл (рис. 8.10). Пользователь подаёт пользовательской программе

(в данном случае — командному интерпретатору) команду на понятном пользователю языке. Программа использует для вывода библиотечную функцию высокого уровня. Библиотека функций переводит полученный запрос на язык, понятный более низкому уровню (ядру операционной системы); таким языком будет интерфейс системных вызовов, а переведённый запрос становится системным вызовом (например, `write`) с соответствующими параметрами.

Со стороны операционной системы вызов `write` обрабатывает виртуальная файловая система, поскольку объект открытого файла находится именно в ней. Параметры системного вызова `write` здесь преобразуются в последовательность действий, необходимых для записи полученной информации в файл, на открытый дескриптор которого сослался вызвавший процесс. Говоря конкретнее, модуль файловой системы переводит запрос в последовательность операций по модификации секторов логического дискового устройства.

Драйвер логического диска, в свою очередь, должен выполнить последовательность операций над секторами диска физического. Соответствующую последовательность запросов он передаёт низкоуровневым подпрограммам ядра, включающим драйвер физического диска. Этот драйвер уже осуществляет действия, нужные для выполнения поступившего запроса с учётом особенностей конкретного физического дискового устройства, то есть переводит полученные запросы на язык команд контроллера.

Современные контроллеры периферийных устройств сами представляют собой достаточно сложные устройства, позволяющие в ряде случаев абстрагироваться от некоторых особенностей аппаратуры, что снимает часть нагрузки с драйверов. Так, большинство современных жёстких дисков на самом деле имеет геометрию (расположение секторов), отличающуюся от видимой для операционной системы. Например, на дорожках, расположенных ближе к центру диска, для сохранения более-менее постоянной плотности записи данных размещают меньше секторов, чем на внешних дорожках (рис. 8.11); но с точки зрения всей остальной системы геометрия диска состоит из N_h сторон, N_c дорожек на каждой стороне (цилиндров), N_s секторов на каждой дорожке, причём все три параметра постоянны и не зависят от значения других. Функцию отображения виртуальной геометрии на физическую берёт на себя контроллер диска. Таким образом, внутри слоя, обозначенного нами как «аппаратура», также имеются различные уровни абстрагирования.

Если говорить совсем строго, то в современных условиях контроллер дисков, подключённый к шине компьютера, на самом деле работает не с диском, а с неким абстрагированным аппаратным интерфейсом; читатель наверняка не раз встречал обозначения таких интерфейсов — SATA, SCSI или уже практически вышедший из употребления IDE. Каждый из них тоже представляет собой шину, «по ту сторону» которой находится очередной контроллер. Физически этот

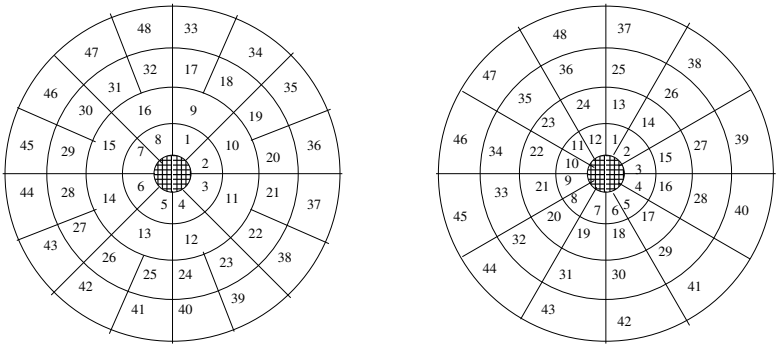

Рис. 8.11. Физическая и виртуальная геометрия диска

контроллер размещён прямо на самом диске — это электронные печатные платы, расположенные на нижней стороне его корпуса.

8.4.5. О роли аппаратных прерываний

Чтобы понять, как функционируют драйверы физических устройств, важно обратить внимание на то, как организовано *ожидание момента, когда устройство завершает требуемую операцию*. Допустим, драйвер устройства выполнил последовательность команд записи в порты, инициировав тем самым некоторую операцию ввода-вывода. Поскольку скорость работы внешних устройств конечна и в большинстве случаев сравнительно невысока, до завершения операции (то есть до момента, когда можно будет воспользоваться результатом) должно пройти некоторое время. Во втором томе мы обсуждали (см. §3.6.2) два подхода к ожиданию этого момента: через циклический опрос порта (активное ожидание) и через обработчик аппаратного прерывания. В первом случае, отправив контроллеру запрос на выполнение какого-то действия, драйвер в цикле выполняет чтение из порта контроллера, чтобы узнать, завершило ли устройство выполнять запрошенное действие. Всё время от начала работы внешнего устройства до её окончания центральный процессор не занят ничем полезным, то есть вычислительное время попросту теряется.

Во втором случае драйвер ничего не ждёт; вместо этого он настраивает обработчик соответствующего аппаратного прерывания и на этом завершает работу, освобождая процессор для более важных дел. Другие подсистемы ядра, получив управление, возвращённое драйвером, в свою очередь при необходимости переводят процесс, затребовавший (прямо или косвенно) данную операцию, в режим блокировки и ставят на выполнение другой процесс, если готовые к выполнению процессы в системе есть, если же их нет — переводят отдельно взятый процессор или всю систему в «состояние покоя» (*idle state*), в котором аппаратура,

Рис. 8.12. Уровни организации ввода-вывода

отвечающая за вычисления (прежде всего центральный процессор) может потреблять меньше электричества. Устройство, завершив заданную драйвером операцию, выдаёт запрос прерывания, в результате которого драйвер устройства снова получает управление (на сей раз — от обработчика прерывания) и выполняет действия, нужные для завершения операции и получения ее результатов. О результатах драйвер сообщает вышестоящей подсистеме. При этом возможно, что процесс, ожидавший результата операции, будет переведён из режима блокировки в режим готовности.

В некоторых случаях оказывается выгоднее (с точки зрения затрат процессорного времени) применить активное ожидание, а не обработку прерывания; обычно так бывает, если ожидаемое время отклика устройства столь незначительно, что потери времени центрального процессора на активное ожидание оказываются меньше, чем потенциальные затраты времени на настройку обработчика прерывания, а затем на обработку самого прерывания. Именно такова ситуация, если драйверу потребовалось узнать некую информацию о текущем состоянии устройства, причём заведомо известно, что ответ на запрос не требует от устройства выполнения каких-либо действий: например, это может быть запрос к сетевой карте относительно того, подключён к ней (физически) провод для передачи данных или нет.

Место обработчиков прерываний в системе можно проиллюстрировать, выделив их в отдельный слой (рис. 8.12). Разумеется, такое деление остаётся во многом условным. Слой аппаратно-независимых компонентов здесь символизирует всё, что есть в операционной системе выше уровня драйверов физических устройств, включая и реализацию логических дисков, и виртуальную файловую систему. Слой обработчиков прерываний не является слоем в полном смысле слова. Драйверы физических устройств отдают распоряжения устройствам о проведении тех или иных операций напрямую, и так же напрямую могут извлекать результаты; в обоих случаях драйвер, являясь программой, выполняется на центральном процессоре и использует команды чтения/записи портов ввода-вывода, взаимодействуя с контроллером устройства. Однако момент, когда следует начать извлечение результатов операции

из контроллера, может выбираться с помощью обработчиков прерываний, т. к. именно определённое прерывание сигнализирует о том, что данные для этого готовы. Обработка прерываний может сыграть свою роль и при выборе момента для инициирования операции ввода-вывода. Действительно, в момент, когда драйвер получил на вход запрос на проведение операции, устройство, которым управляет этот драйвер, может быть ещё занято обработкой предыдущей операции. В этом случае драйверу придётся подождать её завершения, о котором сообщит прерывание; после извлечения результатов можно будет приступить к началу следующей операции с устройством.

8.4.6. Буферизация ввода-вывода

При выполнении операции ввода-вывода бывает так, что соответствующую процедуру на физическом уровне начать в данный момент невозможно. Допустим, процесс A инициировал запись в дисковый файл; в результате цепочки вызовов запрос на эту операцию трансформировался в запрос на запись определённого сектора диска и был передан контроллеру в виде инструкции на позиционирование головки, ожидание нужной фазы поворота диска и запись сектора. Эти физические операции требуют времени, так что процесс A переводится в режим блокировки, а на выполнение запускается процесс B. Допустим теперь, что процесс B тоже потребовал записи в файл, причём этот файл находится на том же физическом диске. Операционная система может трансформировать и его запрос в набор физических операций, однако передать их контроллеру диска не представляется возможным, ведь контроллер всё ещё занят выполнением заказа процесса A.

Логично было бы блокировать процесс B до освобождения контроллера, а затем уже приступить к выполнению его операции. Представим теперь следующую ситуацию. Контроллер по заданию процесса A занят выполнением операции в области последних цилиндров диска. В это время операционной системе пришлось сначала блокировать процесс B, требующий операции в области первых цилиндров, а затем — процесс C, требующий снова операции с последними цилиндрами. Если рассматривать соответствующие запросы в порядке поступления, контроллеру придётся сначала перевести головку из конечной в начальную позицию, а затем снова в конечную. Если процессы A, B и C будут продолжать активно использовать диск, такие переводы головки туда и обратно могут весьма негативно сказаться на общем быстродействии.

Попытки оптимизации (сначала разбудить процесс C, затем уже B) потребуют от планировщика знаний о том, как следует оптимизировать последовательности запросов к данному конкретному устройству. Давать планировщику такие знания нежелательно, ведь они специфичны для разных типов устройств, а такую специфичную информацию не хотелось бы выпускать за пределы драйверов устройств. Кроме того,

процессу в его дальнейшей работе, возможно, и не требуется дожидаться результатов операции вывода, в противоположность операции ввода, результат которой, скорее всего, необходим в вычислениях. Поэтому блокирование процесса в ожидании доступности контроллера для операции может оказаться идеей неудачной.

Рассмотрим теперь операцию ввода данных. Пусть процесс A запросил чтение данных из файла и был заблокирован в ожидании поступления этих данных. Через некоторое время обработчик прерывания сообщил драйверу устройства о готовности запрошенных данных. Теперь драйверу нужно скопировать данные из буфера контроллера куда-то в память. Логично было бы копировать непосредственно в пространство процесса A, однако здесь можно столкнуться ещё с одной проблемой: соответствующая область памяти может оказаться откачана на диск. Таким образом, для разгрузки буфера контроллера от поступивших данных драйверу придётся сначала инициировать загрузку в память нужных страниц из области процесса A. Чтобы понять недопустимость такого варианта, достаточно представить себе, что область подкачки располагается на том же физическом диске, что и прочитанные данные. В этой ситуации драйвер загонит систему в порочный круг: чтобы освободить контроллер, необходимо сначала подкачать в память страницы процесса A, но чтобы это сделать, надо сначала освободить контроллер.

Как видим, ядру часто желательно, а иногда и просто необходимо при проведении операций ввода-вывода сохранять данные в неких областях памяти для промежуточного хранения. Такие области памяти называются ***буферами ввода-вывода***; их не следует, разумеется, путать с буферами контроллеров устройств.

Для дисковых устройств буферы организуются как набор порций данных, соответствующих тому, что должно находиться (но, возможно, пока не находится) в секторах физического диска. В итоге при операции записи вместо непосредственного обращения к драйверу устройства верхний слой ядра просто заносит переданную ему информацию в буфер. Драйвер устройства, имеющий доступ к буферам, самостоятельно определит очерёдность, в которой содержимое буферов будет реально записано на диск; например, при наличии нескольких буферов, соответствующих смежным областям диска, драйвер может записать их все, прежде чем переходить к другим буферам, вне зависимости от того, сколько времени назад соответствующие буферы были созданы.

Как можно догадаться, буферизация часто приводит к уменьшению числа физических операций. Так, если сначала процесс A потребовал записи некоторого сектора, затем процесс B потребовал чтения того же сектора, модифицировал полученные данные и тоже потребовал записи, то физически операция с диском, возможно, произойдёт всего одна. Действительно, информация от процесса A окажется сохранена в буфере; операция чтения, запрошенная процессом B, ни к каким

физическим действиям не приведёт: ему просто выдадут информацию из буфера. Наконец, запрос процесса B на запись модифицирует содержимое уже существующего буфера, и, если к этому времени операция записи всё ещё не была осуществлена, вместо двух операций теперь потребуется только одна: запись последней версии информации. Такие ситуации действительно нередки: системные области диска, отвечающие за глобальные параметры файловой системы, могут подвергаться модификации очень часто, и экономия может достигать тысяч логических операций на одну физическую.

Как мы знаем, внешние устройства бывают не только дисковые (блочные); обмен данными с устройствами других видов обычно организуется в виде потоков — последовательностей байтов. Именно такие потоки используются при передаче данных на печать, при передаче информации по локальной сети или по модемному каналу и т. п. Здесь также возникают определённые трудности, в основном обусловленные ограничением скорости обмена с периферийными устройствами; частично справиться с ними позволяет опять-таки буферная память.

Буферизация потокового вывода позволяет процессам не ждать результатов выполнения операции вывода. Что касается буферизации ввода, то с её помощью можно накапливать информацию, полученную от внешнего источника (например, модема), чтобы выдать её читающему процессу в один приём (за один системный вызов). Поскольку системный вызов, как мы знаем, представляет собой операцию дорогостоящую в плане использования процессорного времени, а информация от внешнего устройства может приходить небольшими порциями и даже отдельными символами (байтами), буферизация здесь также приводит к экономии ресурсов системы. Кроме того, в некоторых случаях буферизация последовательного ввода оказывается необходима по тем же причинам, что и буферизация ввода дискового: необходимость очистки контроллера может возникнуть в тот момент, когда процесс, для которого предназначены полученные данные, либо находится в откачанном состоянии, либо занят другими действиями и не выполняет вызов чтения.

Поскольку буфер может понадобиться в самый неподходящий момент и времени на его подкачку не будет, буферы ввода-вывода всегда размещаются в памяти ядра, которая не подлежит откачке и подкачке. Каждый дополнительный буфер уменьшает количество доступной физической памяти. Ясно, что общий объём буферов в системе оказывается ограничен. При достижении предельного совокупного объёма дисковых буферов некоторые из них могут быть ликвидированы, чтобы освободить память. Ликвидировать, однако, можно не всякий буфер: так, если в буфере содержатся данные, подлежащие записи на диск, но ещё на диск не записанные, уничтожать такой буфер нельзя.

Если предельный объём достигнут, а буферов, допускающих ликвидацию, нет, очередная операция дискового вывода будет заблокирована до тех пор, пока драйвер не запишет на диск информацию из некоторых существующих буферов и не освободит буферную память. Аналогично происходит и работа с буферами последовательного вывода. При их переполнении очередная операция вывода блокируется до тех пор, пока в буфере не освободится место. Наиболее тяжёлая ситуация складывается при переполнении буфера последовательного ввода. Такое может произойти, например, если по каналу связи поступают данные, а процесс, отвечающий за их получение и обработку, по каким-то причинам чтения данных не производит — например, просто не успевает. В этом случае возможна потеря входящих данных, разрыв соединения по каналу связи и т. п. Ясно, что таких ситуаций следует по возможности избегать.

Отдельного внимания заслуживает вопрос, на какую из подсистем ядра следует возложить буферизацию. Традиционный подход — назначить держателями буферов драйверы логических устройств, как символьных (потоковых), так и блочных. С потоковыми устройствами всё до сих пор обычно так и есть, что же касается дискового ввода-вывода, то методы его буферизации изрядно развились. Так, в современных ядрах Linux присутствует подсистема, именуемая Page Cache (страничный кеш), которая изначально была предназначена вовсе не для буферизации ввода-вывода, а для отслеживания статуса имеющихся страниц физической памяти, в том числе их использования процессами, когда физические страницы соответствуют виртуальным страницам процессов. Учитывая, что система поддерживает вызов `mmap` (см. §5.3.7), подсистема Page Cache была вынуждена также хранить информацию о том, что та или иная физическая страница отображает данные, хранимые на диске (в файле), так что вносимые в неё изменения следует рано или поздно отразить на диске. До некоторых пор в Linux существовал отдельный от страничного кеша «буферный кеш», применявшийся при работе с файлами через вызовы `read` и `write` и поддерживавшийся драйверами блочных устройств; одни и те же данные — точнее, один и тот же фрагмент какого-нибудь файла — могли оказаться одновременно отражены и в страничном, и в буферном кеше, что порождало дополнительные сложности. Начиная с версий 2.4.*, в ядре Linux от этого дуализма начали избавляться, а в современных версиях ядра от буферного кеша никаких следов не осталось. Подсистема страничного кеша отвечает за распределение физической памяти, буферизацию дискового ввода-вывода и откачку/подкачку (свопинг), которая тоже, естественно, связана с дисковым вводом-выводом. Применение единой системы для учёта использования физических страниц, свопинга и буферизации файловых операций имеет ряд преимуществ; в частности, поскольку этой подсистеме известно количество свободных страниц, она может задействовать их для временного хранения информации,

прочитанной с диска, а как только эти страницы кому-то потребуются, отдавать их; общий размер дисковых буферов, таким образом, становится динамическим — он всегда равен количеству свободной физической памяти, что избавляет нас от необходимости принятия решения о его размере.

Создатели FreeBSD пошли другим путём. Мы уже упоминали (см. стр. 368), что от блочных устройств в ядре FreeBSD отказались; буферизация файлового ввода-вывода там возложена на виртуальную файловую систему.

Наличие в ядре буферной памяти логично приводит нас к вопросу о том, в какой момент следует вернуть управление процессу, запросившему операцию вывода. Благодаря буферизации управление можно вернуть *сразу же*, записав данные в буфер и не дожидаясь каких-либо результатов. Можно, напротив, записать информацию в буфер, но управление пользовательскому процессу не возвращать, пока операция не будет физически завершена. Эти два подхода, которые мы уже упоминали в §5.3.3, называются соответственно ***асинхронным*** и ***синхронным***. Асинхронный подход более эффективен, поскольку позволяет процессу продолжать работу, не дожидаясь окончания медленной физической операции. С другой стороны, синхронный подход более надёжен, т. к. если во время выполнения физических действий с устройством произойдет ошибка, об этом можно будет сразу же сообщить процессу, тогда как при асинхронном построении вывода процесс может завершиться до того, как результаты операции станут известны. Выбор подхода зависит от стоящих перед нами задач. К примеру, дисковые файловые системы в Unix можно использовать как в синхронном, так и в асинхронном режиме; это указывается при монтировании файловой системы. Иногда используется подход, при котором диски, постоянно установленные в компьютер, работают в асинхронном режиме, а съёмные (дискеты, флеш-карты и пр.) — в синхронном.

Отметим, что **именно применением асинхронного режима обусловлена крайняя нежелательность выключения питания компьютера без подготовки системы к такому выключению.** Результатом неожиданного отключения питания может стать, как известно, нарушение целостности файловой системы и потеря данных, а иногда и полное разрушение файловой системы на логическом уровне, несмотря на то, что физически аппаратура может при этом нисколько не пострадать.

Иногда нужно точно удостовериться, что информация, переданная в операции вывода, была физически записана на диск. В качестве простейшего примера можно назвать операцию вывода на съёмный диск (флеш-брелок, внешний жёсткий диск, старотипную дискету и т. п) перед физическим удалением этого диска из системы. Можно также привести пример с банкоматом, выдающим деньги: прежде чем выдать

пачку банкнот, необходимо удостовериться, что эта операция реально зафиксирована в долговременной памяти, то есть что со счёта клиента соответствующая сумма будет списана. При работе в асинхронном режиме операционная система обычно предоставляет возможность принудительного сброса содержимого буферов на диск. Это называется ***принудительной синхронизацией***. В §5.3.3 мы обсуждали системные вызовы, позволяющие её затребовать — `fsync`, `fdatasync` и `sync`, а в §5.3.7 упоминали также вызов `msync`, работающий с отображениями, созданными с помощью `mmap`.

Отметим ещё один довольно печальный момент. Никакие системные вызовы не могут на 100% гарантировать, что записанные данные достигли поверхности диска. Дело в том, что контроллеры современных жёстких дисков, которые, как уже говорилось, расположены прямо на корпусе устройства и представляют с ним единое целое, обладают своей собственной кеш-памятью, которая может быть не подконтрольна программному обеспечению.

8.4.7. Планирование дисковых обменов

Потребность чтения с диска или записи на диск возникает у системы в двух основных случаях: когда соответствующее действие запрошено пользовательской задачей и когда пользовательский процесс обращается к странице виртуальной памяти, которая в настоящий момент откачана. Если в момент возникновения потребности в чтении или записи контроллер диска свободен, то система, в принципе, может начать нужную операцию немедленно; но контроллер с таким же успехом может быть занят выполнением предыдущей операции. В этом случае сформированный запрос на дисковую операцию помещается в очередь. К тому моменту, когда контроллер завершит выполнение текущего запроса и вновь окажется готов к работе, в очереди может находиться больше одного запроса на работу с тем же диском; на загруженных системах очередь запросов может вообще никогда не пустеть.

На первый взгляд наиболее естественным кажется решение выбирать из очереди те запросы, которые находятся в ней дольше всего, то есть считать очередь действительно очередью, работающей по принципу «кто первый пришёл, того первым обслужили» (*first come first served, FCFS*). Для виртуальных и сетевых дисков, а также дисковых устройств, не являющихся в физическом смысле дисками (например, flash-брелков и накопителей SSD) всё именно так и происходит: поступившие запросы драйвер обрабатывает в порядке поступления.

С устройствами, которые в буквальном смысле физически представляют собой диски, так тоже можно поступить, но если немного изменить подход к выбору следующего запроса из очереди, можно получить заметное повышение производительности системы. К примеру, если для обработки текущего запроса головку пришлось поставить на дорожку с

номером 735, а после завершения обработки этого запроса в очереди обнаруживаются два других запроса, причём первый из них требует чтения с дорожки номер 114, а второй — записи на дорожку с номером 738, то обработка этих запросов в «естественном» порядке потребует сначала переместить головку с дорожки 735 на дорожку 114, а потом обратно и чуть дальше — на дорожку 738; но если запросы поменять местами, длинный путь головка проделает лишь один раз.

Принцип, в соответствии с которыми драйвер выбирает следующий запрос из очереди, называется ***стратегией планирования*** или — не совсем корректно — ***алгоритмом планирования***. Таких стратегий известно довольно много; одна из простейших, но эффективных — на каждом шаге выбирать запрос, выполнение которого требует наименьших затрат времени на перемещение головки. Английское название этой стратегии — *shortest seek time first, SSTF*; она относится к числу так называемых «жадных алгоритмов» (*greedy algorithms*), которые на каждом шаге из всех возможных действий выбирают позволяющее достичь наибольшей выгоды. Применив SSTF, можно добиться высокой производительности, но у этой стратегии есть фатальный недостаток — возможность уже знакомого нам ***ресурсного голодания*** (*starvation*). Если два или три процесса будут активно работать с файлами, находящимися в одной области диска, то процесс, чьи потребности относятся к другой области диска, может никогда не дождаться своей очереди, ведь до дорожек, нужных первым процессам, всегда будет ближе.

Другая широко известная стратегия получила название ***лифтового алгоритма*** (англ. *elevator algorithm*). В этом случае планировщик перемещает головку в одном направлении — например, от меньших номеров к бóльшим, по пути обслуживая все имеющиеся в очереди запросы, относящиеся к дорожкам, которые он проходит; дойдя до края — например, до запроса с наибольшим номером дорожки среди всех, которые были в очереди, и обслужив этот запрос, планировщик меняет направление движения головки; теперь она перемещается в сторону меньших номеров дорожек, опять-таки обслуживая все запросы, попадающиеся по пути, пока не будет обслужен запрос с наименьшим (среди имеющихся запросов) номером дорожки, и так далее. Название стратегии объясняется тем, что автоматические лифты работают примерно так же: кабина движется вверх, пока не достигнет наибольшего запрошенного этажа, затем движется вниз, пока не достигнет наименьшего этажа среди запрошенных. Проблема ресурсного голодания здесь не возникает, рано или поздно любой запрос будет обслужен.

В англоязычных источниках часто рассматривают в качестве двух разных стратегий нечто, именуемое SCAN и LOOK. Тот вариант, который только что ввели мы — это как раз LOOK; что касается SCAN, то он отличается тем, что головка перемещается в заданном направлении, пока не достигнет *края диска* (а не дорожки с максимальным/минимальным номером среди требуемых имеющимися запросами). В действительности такой вариант никогда не применяется,

поскольку он заведомо хуже; кто первым придумал рассматривать SCAN как отдельную стратегию, непонятно, зато на этом примере мы можем наглядно видеть, сколь охотно люди готовы копировать ошибки других людей.

В качестве проблемы лифтового алгоритма обычно называют сравнительно большой разброс времени ожидания для поступившего запроса. Например, если на диске есть 1000 дорожек, головка начала движение в направлении уменьшения номеров и только что обслужила запрос на чтение или запись на дорожке 995, а в это время поступил запрос на работу с дорожкой 996, но в очереди также присутствует запрос на дорожку с номером 12, то время ожидания для запроса 996 составит почти двойное время перемещения головки диска от края до края. Отметим, что такое возможно только для дорожек, расположенных близко к краям диска; для дорожек в середине диска разброс времени ожидания оказывается вдвое меньше. Вероятность того, что очередной запрос будет вынужден долго ждать, существенно повышается при движении головки от конца диска к началу, ведь при чтении и записи больших объёмов данных возникает последовательность запросов, затрагивающих области диска, идущие подряд — естественно, в направлении возрастания номеров.

Простая модификация лифтового алгоритма позволяет сократить и выровнять разброс времени ожидания, хотя среднее время ожидания при этом не меняется. В изменённом варианте головка движется от меньших номеров дорожек к бóльшим, по пути обслуживая запросы; исполнив запрос с максимальным номером дорожки, головка возвращается к началу диска и снова проходит диск в направлении от меньших номеров к бóльшим. Эта стратегия, называемая ***круговым лифтовым алгоритмом***, может быть почти столь же эффективной (по среднему времени обслуживания запроса), как и обычный лифтовый алгоритм, благодаря тому, что большинство физических дисков способно перемещать головку на край диска с бóльшей скоростью, чем когда требуется найти конкретную дорожку. Возврат головки в начало диска с позиции в его конце происходит почти так же быстро, как перемещение на соседнюю дорожку. Разброс времени ожидания для этой стратегии вдвое меньше и не зависит от номера дорожки.

Существует множество модификаций описанных стратегий. Например, можно применять «жадную» стратегию не более чем для N запросов, после чего, даже если есть ещё запросы на близлежащие дорожки, очередной запрос выбирать с помощью лифтового алгоритма. В таком варианте эффективность близка к показателям чистой «жадной» стратегии, но при этом имеется очевидная гарантия недопущения ресурсного голодания.

Довольно интересен так называемый ***предчувствующий алгоритм*** (англ. *anticipation algorithm*). Если после обслуживания очередного запроса требуется большое перемещение головки, то прежде чем приступить к нему, планировщик делает небольшую паузу в надежде,

что за это время поступит ещё запрос на ту же или следующую дорожку, и это во многих случаях действительно происходит (отсюда название).

Планирование обменов в реально существующих ядрах операционных систем в наши дни производится по ещё более изощрённым стратегиям. Так, в ядре Linux используется так называемый *Completely Fair Queuing (CFQ)*, который разделяет запросы на синхронные и асинхронные, отдавая предпочтение синхронным (например, запросам на чтение), поскольку до их окончания соответствующий процесс не может продолжать выполнение; каждому процессу выделяется некое *время*, в течение которого планировщик дискового обмена готов выполнять его запросы, не обращая внимание на запросы, поступающие от других процессов. Подробности об устройстве этого планировщика читатель без труда найдёт в Интернете.

В §8.4.4 мы обсуждали виртуализацию геометрии физических дисков на уровне контроллера самого диска; с этим связано очевидное ограничение применимости (и осмысленности) изощрённых стратегий планирования дисковых обменов, ведь те воображаемые «головки», которыми позволяет «управлять» контроллер диска, теперь напоминают реально существующие механические головки только издали; попросту говоря, операционная система не знает и не может знать, какое расстояние в действительности придётся преодолеть физическим головкам диска, она даже не знает, *сколько* этих головок у диска реально есть. Создателям жёстких дисков приходится подбирать такое отображение физической геометрии на виртуальную, чтобы минимизация пути виртуальных головок в большинстве случаев уменьшала также и путь головок реальных; надо сказать, что они с этой задачей справляются, несмотря на её нетривиальность.

8.4.8. Виртуальная файловая система

Мы уже не раз упоминали ***виртуальную файловую систему*** — подсистему ядра, отвечающую за поддержку файлов как абстрактных объектов, обладающих определёнными свойствами. Как мы знаем, файл можно *открыть* (на запись, чтение или и то и другое), можно читать из него или записывать в него информацию, если это позволяет установленный режим работы, можно закрыть его, когда работа окончена. К открытому файлу можно, хотя и не всегда, применить отображение в память (вызов `mmap`), позиционирование (`lseek`), есть и другие операции. Файл в системах семейства Unix обладает информацией о владельце и группе, правами доступа, датой создания, модификации и последнего доступа, можно узнать количество установленных на него ***жёстких ссылок*** — то есть, попросту говоря, файловых имён, которые с данным файлом связаны. Более того, мы обсуждали, что вся служебная информация о файле, за исключением его имени, сосредоточена в ***индексном дескрипторе***, а для некоторых типов файлов

только индексный дескриптор и есть, то есть файл может не занимать на диске никакого места, кроме одного дескриптора в массиве индексных дескрипторов. Коль скоро мы вспомнили про типы файлов, можно заодно отметить, что в unix-системах их поддерживается семь: обычные файлы, директории (каталоги), символические ссылки, каналы (FIFO), сокеты и два типа файлов устройств — потоковые и блочные. Имена файлов хранятся в директориях, один файл может иметь больше одного имени (жёсткой ссылки), причём как в разных директориях, так и в одной; и так далее. Свойства файлов в unix-системах и операции, которые над ними можно производить, мы уже обсуждали (см. главу 5.3). Все эти свойства и операции как раз и образуют ***абстракцию файла***, характерную именно для систем семейства Unix.

Напомним, что термин ***файловая система*** может обозначать две совершенно разные, хотя и связанные между собой сущности: структуру данных на диске и набор программных функций, которые с этой структурой данных умеют работать. Программные функции, которые в случае Unix обычно находятся в ядре системы (хотя и не всегда), определяют то, *как* должны быть организованы данные на диске. Надо сказать, что unix-системы работают с довольно разными файловыми системами; для одного только Linux характерны системы `ext2`, `ext3`, `ext4`, `ReiserFS`, а кроме них Linux поддерживает также `minix`, `xia`, `jfs`, `xfs` и другие. Для FreeBSD основной файловой системой считается `ffs` (она же `ufs`), несколько реже используется `zfs`, пришедшая из системы Solaris. Для обеих этих файловых систем есть поддержка и в Linux, а FreeBSD, в принципе, можно научить работать с `ext2-4`, `ReiserFS` и другими файловыми системами, используемыми в Linux. Все перечисленные файловые системы поддерживают общий набор понятий, характерных для файловых систем семейства Unix. Файлы в них основаны на индексных дескрипторах, имеются файлы семи известных нам типов, в том числе, что особенно важно, директории и символические ссылки; индексные дескрипторы содержат привычные для Unix права доступа, идентификаторы владельца и группы, информацию о датах и прочее.

Работающая операционная система использует один из своих дисков в роли *корневого*; организованная на нём файловая система считается ***корневой файловой системой***, её корневой каталог совпадает с общесистемным — каталогом «`/`». Корневые каталоги остальных файловых систем отображаются на общее дерево каталогов в так называемых ***точках монтирования*** (англ. *mount points*) — директориях, которые на момент подключения (***монтирования***) новой файловой системы уже есть в общем дереве. Как правило, в роли точек монтирования используют пустые директории; если там ранее содержались файлы, то они никуда не денутся, но видно их уже не будет — до тех пор, пока новая система не будет размонтирована.

Общее дерево каталогов операционной системы может состоять из файловых систем разных типов, так что ядру придётся задействовать несколько разных наборов функций, причём делать это прозрачно для пользователя в том смысле, что работа с файлами с точки зрения пользовательских программ протекает совершенно одинаково вне зависимости от того, какой тип имеет файловая система.

Дело резко осложняется тем, что ядро вынуждено в некоторых случаях поддерживать диски, организованные по правилам других операционных систем. Как Linux, так и FreeBSD (и другие unix-системы) без особых проблем справляются с носителями, отформатированными на системах линейки Windows, для чего их ядра включают поддержку таких файловых систем, как `msdos`/`fat`, `fat32`, `vfat` и `NTFS`. Обе операционные системы поддерживают также файловую систему `iso9660`, используемую для оптических дисков (CD и DVD). Сложность здесь в том, что во всех этих файловых системах нет никаких индексных дескрипторов, права доступа либо отсутствуют, как в `fat`, либо организованы совершенно не так, как в unix-системах (`NTFS`).

Картина становится ещё более интересной, если упомянуть файловые системы, не предполагающие использования диска. Речь в данном случае идёт не о том, что файловая система создаётся на виртуальном диске в памяти, как это обсуждалось в §8.4.3; в этом случае всё как раз достаточно просто, ведь с точки зрения аппаратно-независимых компонентов ядра виртуальный диск почти ничем от реального не отличается. Более хитро устроены файловые системы, вообще не обращающиеся ни к каким дискам — ни реальным, ни виртуальным; можно сказать, что такая файловая система существует только в одной своей ипостаси — программной, а дисковых структур данных нет за неимением диска. Тем не менее, на такой файловой системе располагаются файлы, их можно открывать, читать, записывать и т. п.

Именно такова, например, файловая система `tmpfs`, поддерживаемая большинством современных unix-систем. Надо сказать, что в использовании она существенно удобнее, нежели обычная файловая система на виртуальном диске. Дело в том, что при создании файловой системы на виртуальном диске нужно указать и зафиксировать его размер, поскольку структуры данных практически любой дисковой файловой системы существенно зависят от размера диска. Файловая система `tmpfs` этого ограничения лишена, на неё можно записывать файлы до тех пор, пока в системе не исчерпается виртуальная память (то есть вся физическая память и все возможности своппинга). При удалении файлов с такой файловой системы память немедленно освобождается и может быть использована ядром для других целей — например, выделена какому-нибудь процессу.

Ещё больше путаницы возникает благодаря файловым системам, которые мы будем называть ***искусственными***, хотя этот термин и не

вполне удачен. В английском оригинале их обычно называют *pseudo file systems*, но прямой перевод на русский оказывается невозможен из-за того, что «псевдо» в русском языке — приставка, а не отдельное слово, и применить её не к слову, а к словосочетанию (в данном случае — к термину «файловая система») не получается. Несколько реже в англоязычных источниках та же самая сущность обозначается термином *synthetic file system*, чем мы и воспользуемся, переведя слово *synthetic* как «искусственный».

Искусственные файловые системы не имеют отношения не только к дискам, но и вообще к хранению информации. Выглядят они более-менее как обычные файловые системы с деревом каталогов и файлами, но все эти каталоги и файлы на самом деле представляют собой отражение состояния каких-то подсистем внутри ядра. Читая эти файлы, мы можем узнать, что творится в ядре; запись в некоторые из них позволяет менять настройки ядра и режимы работы отдельных его подсистем. Иначе говоря, такая файловая система — это *интерфейс* для управления ядром, который только выглядит как файловое поддерево. Такой стиль построения интерфейсов к подсистемам ядра имеет несомненное достоинство: не нужно добавлять лишние системные вызовы, которых и так очень много.

Классическим примером искусственной файловой системы может служить `procfs`, которая обычно монтируется на директорию `/proc`. Для каждого процесса, существующего в системе, появляется поддиректория, соответствующая его `pid`'у; например, для процесса `2715` это будет директория `/proc/2715`. Она содержит несколько десятков «файлов», отражающих различные аспекты работы процесса. Так, `/proc/2715/exe` — это символическая ссылка на исполняемый файл, `/proc/2715/cwd` — на текущую директорию (*current working directory*) процесса `2715`, а поддиректория `/proc/2715/fd` содержит символические ссылки на имеющиеся у процесса открытые потоки ввода-вывода; файл `/proc/2715/cmdline` «содержит» аргументы его командной строки, разделённые нулевым байтом; `/proc/2715/maps`, если его прочитать (например, выдать на экран с помощью `cat`), выглядит как текстовый файл, содержащий информацию об областях виртуальной памяти процесса. Между прочим, настоятельно рекомендуем поэкспериментировать с этими файлами; особенно интересным может оказаться анализ выполнения своей собственной программы, написанной на языке ассемблера или на Си.

Помимо информации о процессах, `/proc` содержит также целый ряд других псевдофайлов. Например, файл `/proc/sys/net/ipv4/ip_forward` может «содержать» `0` или `1`; `0` означает, что ядро не передаёт пакеты между сетевыми интерфейсами, то есть система не работает в роли маршрутизатора, `1` показывает, что передача пакетов

включена. Более того, чтобы её включить или выключить, надо занести в этот файл соответствующее значение, как в обычный файл, например:

```
root@gw2:~# echo "1" > /proc/sys/net/ipv4/ip_forward
```

В ОС Linux традиционно многие системные программы опираются на информацию из `/proc`; так, команда `ps` всю нужную ей информацию получает именно оттуда. В мире FreeBSD отношение к этой искусственной файловой системе иное: для работы собственных утилит FreeBSD она не нужна, так что не на каждом компьютере её монтируют, но поддержка для неё в ядре есть — на случай запуска программ, пришедших из Linux.

Ещё один пример искусственной файловой системы в Linux — `sysfs`, которая обычно монтируется на директорию `/sys`. Ввели её сравнительно недавно, чтобы разгрузить `/proc`, переполненную информацией, не имеющей отношения к процессам. Исследуя «содержимое» `/sys`, можно узнать, какие устройства подключены к вашему компьютеру (в том числе, например, к портам USB), какие модули загружены в ядро и многое другое. В ОС FreeBSD аналога этой системы нет.

Наконец, файловая система `devfs` позволяет сэкономить индексные дескрипторы на физических дисках, не создавая файлы устройств в директории `/dev`. Вместо этого `/dev` используется как точка монтирования для `devfs`, которая уже делает вид, что содержит файлы для всех устройств, поддерживаемых ядром. Создатели многих современных дистрибутивов Linux предпочитают идти другим путём: на `/dev` монтируется обычная `tmpfs`, которую наполняет файлами работающий в системе демон `udev` или его аналог; в этом случае обычно на `/dev/pts` монтируется другая искусственная файловая система, `devpts`, содержащая только файлы для главных и подчинённых устройств псевдотерминалов.

Рассказ о файловых системах, не подразумевающих использования дискового устройства, был бы неполным без упоминания `NFS` (*network file system*) и `smbfs`, которые представляют в виде локальных файловых деревьев диски, расположенные на других компьютерах. `NFS` работает по протоколу, который так и называется NFS и изначально появился в системах семейства Unix, тогда как `smbfs` использует протокол SMB/CIFS, пришедший из Windows, что означает неизбежное применение «чужой» абстракции файла (другое устройство прав доступа и т. п.).

Всё это великолепие должно обслуживаться единым набором системных вызовов, большинство которых нам уже знакомо. Для файловых систем, характерных для семейства Unix, всё ещё сравнительно просто, ведь они различаются в основном только форматом представления информации о файлах на диске, а сущности вроде индексного дескриптора, суперблока (дискового блока, содержащего «глобальную» информацию

о файловой системе), директории и прочего остаются общими для них всех. Для «чужих» файловых систем, таких как `vfat`, `NTFS` или `smbfs`, абстракции, ожидаемые unix-системой, приходится эмулировать; для искусственных файловых систем эмулировать приходится вообще всё, что в них есть. За обработку особенностей каждой файловой системы отвечает соответствующий модуль ядра, который, как мы уже неоднократно отмечали, тоже называется файловой системой; эти модули, как обычно в подобных случаях, экспортируют некий фиксированный набор интерфейсных функций (точек входа) — примерно так же, как это делают драйверы.

Обязанности координатора всех файловых систем выполняет виртуальная файловая система; она отвечает за монтирование и размонтирование отдельных файловых систем, за формирование и обслуживание общего дерева каталогов, за выбор подходящего модуля для поддержки каждой конкретной файловой системы и за перевод запросов с языка системных вызовов на язык точек входа модулей файловых систем. С другой стороны, именно виртуальная файловая система обрабатывает системные вызовы файлового ввода-вывода, создаёт и поддерживает потоки ввода-вывода (как объекты ядра).

8.4.9. Файловая система на диске

Структуры данных, создаваемые на диске для поддержки файловой абстракции, могут быть очень разными. Даже файловые системы, выдержанные в стиле Unix (то есть основанные на индексных дескрипторах), друг от друга сильно отличаются; но выделить некие общие принципы всё же можно.

Для построения файловой системы физические сектора диска (имеющие обычно размер 512 байт) группируются в ***блоки***, размер которых зависит от файловой системы; более того, один формат файловой системы может допускать разные размеры блоков, в этом случае размер задаётся при форматировании диска. Каждый блок имеет свой номер. Обычно самый первый блок на диске содержит глобальную информацию обо всей файловой системе (так называемый ***суперблок***); как правило, в нескольких местах диска располагают копии суперблока на случай его потери.

Часть блоков отводится под ***области индексных дескрипторов***; таких областей может быть от нескольких десятков до нескольких тысяч. В каждый блок области помещается несколько индексных дескрипторов (всегда степень двойки); в рамках одной файловой системы дескрипторы имеют сквозную нумерацию. Например, системы `ext2`–`ext4`, применяемые в ОС Linux, делят блоки на группы блоков, каждая такая группа имеет свою копию суперблока, свой массив дескрипторов и свою карту свободных блоков — массив данных, занимающий ровно один блок,

в котором каждый бит соответствует блоку из этой же группы, ноль означает, что блок свободен, единица — что он занят.

Размер индексного дескриптора тоже различается от системы к системе; в наши дни типичный размер — 256 байт. Как мы знаем, индексный дескриптор хранит всю информацию о файле, кроме его имени и содержимого; имена файлов хранятся в директориях (напомним, что это файлы специального типа), а содержимое файлов располагается в блоках, не занятых служебной информацией. Если файлу принадлежат блоки диска, индексный дескриптор содержит информацию о размещении, то есть о номерах этих блоков. Обычно структура индексного дескриптора предусматривает массив для хранения номеров блоков, причём первые N элементов массива содержат непосредственно номера блоков, в которых размещено содержимое файла; в конце массива три элемента содержат номера ***косвенных блоков*** (англ. *indirect block*) — таких, которые сами содержат номера блоков, причём косвенные блоки различаются по уровням: косвенный блок первого уровня содержит номера обычных блоков, косвенный блок второго уровня — номера косвенных блоков первого уровня, косвенный блок третьего уровня — номера косвенных блоков второго уровня. Элементы массива в индексном дескрипторе содержат номер одного косвенного блока каждого уровня.

Например, в системе `ext2` в индексном дескрипторе массив номеров блоков имеет 15 элементов, из которых первые 12 содержат номера простых блоков; если размер блока составляет 4 Кб (это самый типичный случай), то файлы до 48 Кб (12×4) обходятся без косвенных блоков. Для файлов бóльшего размера приходится завести косвенный блок первого уровня, который (при размере номера блока в 4 байта) может содержать до 1024 номеров блоков; это позволяет организовать файлы размером до 1036 блоков, т. е. до 4144 Кб. Когда не хватает и этого, система заводит косвенный блок второго уровня; это позволяет применить ещё 1024 косвенных блока первого уровня, а общее число блоков, занимаемых файлом (не считая косвенных), увеличить до $1024 \times 1024 + 1024 + 12 = 1049612$; сам файл при этом может увеличиться до 4 Гб с небольшим. Если не хватит и этого, в бой будет брошена тяжёлая артиллерия — косвенный блок третьего уровня; после этого резервов больше не остаётся, но максимальный возможный размер файла вырастает до $(1024^3 + 1024^2 + 1024 + 12) \times 4096$ байт, т. е. свыше 4 Тб (терабайт). Впрочем, конкретно `ext2` имеет другое ограничение — 32-разрядное число в структуре индексного дескриптора, означающее размер в 512-байтных секторах, так что вырастить отдельно взятый файл больше чем до 2 Тб не получится.

8.4.10. Шина, кеш и DMA

Материал этого параграфа не имеет прямого отношения к операционным системам, речь здесь пойдёт о возможностях аппаратуры. При соблюдении «справочной» логики изложения уместнее было бы рассказать об этом в первом томе — в главе 1.3, где шёл разговор об основах устройства компьютера, или во втором томе — в части III, где говорилось о возможностях процессора; но наша книга — не справочник. Во вводной части первого тома объяснять тонкости, связанные с работой шины, было явно рано, ведь она написана в расчёте на читателя, пока что не знающего о компьютерах вообще ничего. В части, посвящённой процессору, мы намеренно не полезли в архитектурные дебри. Если там рассказывать о кеш-памяти, DMA и прочих изысках из этой области, то пришлось бы перетащить туда же, например, MMU с моделями виртуальной памяти; но второй том и без того получился довольно объёмным, к тому же хотелось дать читателю возможность быстрее проскочить ассемблер и приступить к изучению Си.

Кеш-память поэтому лишь вскользь упомянута в первом томе; прямой доступ контроллеров устройств к оперативной памяти (*direct memory access, DMA*) мы пока не упоминали вообще. Такое положение вещей вполне могло нас устраивать до поры до времени, но не сейчас, когда в своих изысканиях мы спустились на уровень взаимодействия драйверов с аппаратурой. В любых (на что-то годных) источниках на эту тему, которые читатель пожелает изучить в дополнение к нашей книге, обе сущности наверняка будут постоянно упоминаться, причём, скорее всего, без пояснений — в предположении, что читатель знает, о чём идёт речь.

Углубляться в тонкости устройства кеш-памяти и DMA мы не будем, для этого есть специальная (в основном справочная) литература, к тому же в этой области всё довольно быстро меняется и сильно зависит от конкретной аппаратной платформы. В частности, мы не станем рассматривать алгоритмы вытеснения страниц кеша и принципы поддержки когерентности кешей в многопроцессорных машинах — это тоже очень долгий разговор, без знания этих тонкостей вполне можно обойтись, если же вдруг станет интересно — то ни в коем случае не отказывайте себе в знаниях, тем более что с поиском информации в наше время никаких проблем нет, надо только точно знать, что искать.

Приблизительно до середины 1980-х годов процессоры были довольно медленными, так что шина и микросхемы оперативной памяти, будучи устройствами гораздо более простыми, легко «поспевали» за темпом работы процессора. По мере развития технологий наметился определённый дисбаланс в скоростях; дело в том, что скорость работы цифровой электроники невозможно наращивать бесконечно, этому мешает скорость света. Если сравнить линейные размеры кристалла микропроцессора, диагональ которого редко превышает 3 см, с длиной дорожек шины, мы обнаружим разницу более чем на порядок; приняв во внимание, что даже непосредственно на кристалле схема самого процессора (точнее, вычислительного ядра[20], которое нас сейчас и ин-

[20] См. сноску 1 на стр. 319.

тересует) обычно занимает лишь малую часть площади, мы получим ещё один порядок разницы.

Если принять для ровного счёта длину шины равной 25 см, получится, что за один такт её работы электрический сигнал должен преодолеть 0,5 м, чтобы успеть дойти от запрашивающего к запрашиваемому и обратно, как это и происходит при операции чтения — неважно, из памяти или порта ввода. Скорость света, как мы знаем, составляет приблизительно 300 тысяч км/с, или 300 миллионов м/с, или 600 миллионов «полуметров» в секунду; следовательно, для шины длиной 25 см тактовая частота 600 МГц представляет абсолютный теоретический предел. В реальной жизни такт должен длиться дольше, оставляя запас времени, чтобы запрашиваемый успел среагировать на запрос, а запрашивающий — «снять» с шины пришедшую информацию. С учётом всего этого можно заметить, что шины современных материнских плат, работающие на частотах около 450 МГц, «разгонять» совершенно некуда. Тактовая частота современных процессоров достигает 3 ГГц, что почти в семь раз больше. Конечно, разгонять их ещё больше мешает всё та же скорость света, но и имеющаяся разница достаточна, чтобы процессор, коль скоро ему придётся всё время взаимодействовать с оперативной памятью, бóльшую часть времени простаивал.

Если рассматривать хорошо знакомую нам линейку процессоров x86 (см. т. 2, §3.1.3), то для процессоров, предшествовавших 80286, проблема дисбаланса скоростей не проявлялась вообще. На 80286 эта проблема уже возникла: выполняя операцию чтения из памяти, процессор был вынужден пропускать такт, давая микросхемам памяти и шине достаточно времени, чтобы они успели сработать. Если бы всё по-прежнему работало так, как в те времена, современным процессорам приходилось бы пропускать больше десяти тактов. Появившуюся проблему нужно было как-то решать, поэтому на некоторых (хотя и не на всех) материнских платах, предназначенных для i386, появились специальные микросхемы памяти, расположенные рядом с процессором. Скорость их работы ненамного превосходила скорость обычной памяти, но для обращения к этим микросхемам не нужно было задействовать шину.

В общих чертах работа с кешем протекает так. Вся физическая память делится на блоки одинакового размера. В кеш-памяти содержатся *копии* некоторых блоков физической памяти вместе с *метками*, показывающими, копия какого блока сохранена в этой области кеша, а при некоторых способах организации кеша — ещё и информация о том, когда в последний раз к этому блоку происходило обращение. Процессор, прежде чем обратиться к памяти, должен проверить, не содержится ли в кеше копия нужного блока; если нет — весь блок целиком копируется в кеш, при этом из кеша может быть удалён какой-то другой блок, чтобы расчистить место. Обмен с памятью по шине теперь протекает

в основном целыми блоками, что позволяет его ускорить, увеличив разрядность шины данных.

К выполнению операций записи есть два подхода. Можно записывать информацию в ячейки кеша, помечая соответствующий блок как «грязный», и при вытеснении этого блока из кеша записывать его в память целиком; можно, с другой стороны, каждую операцию записи выполнять немедленно, то есть записывать сразу и в кеш, и в память, благо процессору совершенно не нужно дожидаться окончания операции записи. Эти подходы называются ***отложенная запись*** и ***сквозная запись*** (англ. *write-back* и *write-through*); надо сказать, что такие же термины часто применяются при обсуждении буферизации ввода-вывода.

Отдельного внимания заслуживает вопрос, как реализовать метки, как искать в кеше нужный блок и как принимать решение, какой из блоков должен быть заменён, когда места в кеше больше нет. Подходов к этой проблеме существует достаточно много, подробное их рассмотрение оставим за рамками книги; отметим только, что самый простой и довольно остроумный способ состоит в том, чтобы каждый блок физической памяти мог отображаться только на один из блоков кеша: если кеш может содержать одновременно M блоков, то блок памяти с номером n может быть отображён только в блок кеша с номером, равным остатку от деления n на M. Поскольку число M обычно выбирается равным 2^k, деление с остатком сводится к выделению k младших битов. Такой подход может показаться неэффективным, но стоит учесть, что при этом не нужно ни запоминать, к каким блокам производились обращения, ни искать что бы то ни было где бы то ни было (поскольку блок памяти либо находится в «своей» части кеша, либо нет).

Начиная с процессоров i486, кеш стали размещать на одном кристалле с процессором. Конечно, этот кристалл не резиновый, поэтому много кеш-памяти на нём поместиться не может; процессоры i486 имели 8 Кб кеш-памяти на одном кристалле с самим процессором, и, кроме этого, некоторое количество кеша (чаще всего 256 Кб) предусматривалось на материнской плате рядом с процессором. Такая организация кеша называется многоуровневой; первый уровень кеша (L1) располагается ближе всего к процессору и имеет самую высокую скорость работы, но при этом обладает незначительным объёмом, по мере роста номера уровня объём кеша растёт, скорость падает. Кеш последнего уровня (в наши дни — обычно четвёртого, L4) представляет собой отдельную микросхему, кеши остальных уровней располагаются на одном кристалле с процессором. Многоядерные процессоры обычно предусматривают отдельный кеш L1 для каждого ядра и общий для всех ядер кеш L3, а L2 может быть как общий, так и свой у каждого ядра — это зависит от модели процессора. Например, четырёхъядерный Intel Core i7 имеет на каждое ядро по 32 Кб кеша L1 отдельно для команд и для данных,

смешанный (для инструкций и данных) кеш L2 размером 256 Кб (тоже свой для каждого из ядер) и общий для всех ядер кеш L3 на 8 Мб.

Применение кеш-памяти имеет любопытный побочный эффект: если всё происходит правильно, то процессор будет сравнительно редко обращаться к общей шине. Время, когда шина не нужна процессору, можно использовать, если вспомнить, что достаточно большие объёмы данных часто приходится перекачивать между памятью и контроллерами внешних устройств. Раньше этим приходилось заниматься центральному процессору, и мы описывали именно этот вариант взаимодействия между драйвером и контроллером; например, при чтении с диска драйвер задавал контроллеру выполнение операции и на этом заканчивал работу, освобождая процессор до тех пор, пока не произойдёт прерывание от контроллера; вновь получив управление, на сей раз от обработчика прерываний, драйвер выполнял копирование считанных с диска данных из буфера контроллера в оперативную память. Но благодаря использованию кеш-памяти шина бóльшую часть времени всё равно свободна, а копирование данных — операция достаточно примитивная, и задействовать для неё центральный процессор явно не обязательно. Из этого логично вытекает идея возложить копирование данных на кого-то ещё, причём так, чтобы это происходило, когда центральному процессору шина не требуется.

Позволить управлять шиной кому-то кроме процессора пытались и раньше, но по противоположной причине: ранние процессоры работали не быстрее, а, наоборот, медленнее шины. По своему устройству и шина, и микросхемы памяти гораздо проще процессора, а до ограничений из-за скорости света в те времена было ещё очень далеко, так что скорость работы более простых компонентов системы росла быстрее.

Уже в составе IBM PC, основанного на процессоре Intel 8088, присутствовал ***контроллер прямого доступа к памяти*** (контроллера DMA) — микросхема Intel 8237. Это было сравнительно простое устройство, подключённое к общей шине и имеющее свои собственные порты ввода, позволявшие сформулировать ему задание: откуда копировать информацию, куда и в каком количестве. Пусть нам, к примеру, нужно выдать некие данные в потоковое устройство, то есть записать порцию информации, расположенную в памяти, в однобайтовый порт вывода. Мы можем с помощью обычного цикла последовательно извлекать содержимое из каждой ячейки (например, командой `MOV`) и записывать полученное значение в порт (командой `OUT`), после чего увеличивать адрес ячейки, уменьшать счётчик, и если он всё ещё больше нуля, возвращаться к началу цикла. В исполнении ранних процессоров всё это происходило, как мы уже отметили, довольно медленно, и к тому же требовало двух тактов шины на каждый передаваемый байт.

Наличие контроллера DMA позволяло драйверу устройства поступить иначе: занести в порты контроллера DMA адрес начала области

памяти, содержащей данные, и количество этих данных, после чего дать ему команду на проведение операции вывода. Шина ISA, использовавшаяся в компьютерах линейки IBM PC вплоть до некоторых компьютеров на процессоре Pentium-1, предусматривала (в составе шины управления) специальные дорожки для управления прямым доступом к памяти, составлявшие так называемые *каналы DMA*, по две дорожки на каждый канал (запрос DMA — *DRQ, DMA request* и подтверждение DMA — *DACK, DMA acknowledged*). В ранних версиях шины ISA таких каналов было три[21], в более поздних — семь. Контроллеры внешних устройств, поддерживавшие прямой доступ к памяти, выдавали очередную порцию (байт) данных в шину (или, наоборот, считывали порцию данных) по сигналу «подтверждение DMA» с тем номером, который им был приписан. Это позволяло контроллеру DMA не обрабатывать данные самому: он давал порту ввода сигнал «чтение» через канал DMA, а ячейке памяти — обычный сигнал «запись», сопровождаемый выставлением её адреса на шине адресов, в результате чего порт устанавливал шину данных в состояние, соответствующее читаемым из него данным, а ячейка памяти снимала эту информацию с шины данных и запоминала, и всё это за один такт шины; естественно, можно было передать данные и в обратном направлении. Такой режим передачи данных по-английски называется *fly-by mode*. Всё, что требуется при этом от контроллера DMA — это после каждого такта увеличивать адрес ячейки памяти, в нужные моменты выдавать управляющие сигналы в шину управления и подсчитывать, сколько ещё осталось передать байтов[22], чтобы вовремя остановиться.

Передача блока данных между контроллером периферийного устройства и памятью в таком режиме происходила в несколько раз быстрее, чем если то же самое делать силами центрального процессора, что уже само по себе оправдывало существование контроллера DMA; но, кроме передачи блоков данных за один раз, контроллер DMA умел работать и по-другому. Например, внешнее устройство могло отдавать данные через порт со скоростью, меньшей скорости шины; в этом случае по мере готовности очередных байтов данных контроллер внешнего устройства взводил на шине сигнал DRQ; получив этот сигнал, контроллер DMA отвечал сигналом DACK и производил один цикл передачи. Благодаря наличию независимых каналов DMA в таком режиме можно было

[21] Хотя контроллер DMA поддерживал четыре канала, один из них — нулевой — в совсем ранних версиях архитектуры использовался для обновления содержимого микросхем памяти; в расширенной версии шины ISA он стал использоваться для каскадирования со вторым контроллером DMA.

[22] На IBM PC это действительно были байты, а на IBM PC AT, построенной на основе процессора 80286, за один приём передавалось 16 бит, несмотря на то, что контроллер DMA остался тот же самый, и он был восьмибитным (хотя их стало два ради увеличения числа каналов DMA). Для программирования самого контроллера DMA этих восьми бит хватало, а разрядность проходящих по шине данных его никоим образом не трогала.

обслуживать несколько устройств одновременно, позволяя при этом работать и центральному процессору тоже — если шина была занята контроллером DMA, а процессору нужно было выполнить операцию с шиной, он попросту пропускал нужное количество тактов или, наоборот, заставлял контроллер DMA подождать.

Кроме режима *fly-by* контроллер DMA Intel 8237 поддерживал также и режим «получить-записать», требовавший двух тактов на пересылку одного байта. В этом режиме его можно было использовать в том числе и для копирования данных между двумя областями памяти. В компьютерах линейки x86 этот режим никогда не применялся.

С появлением новой шины, получившей обозначение PCI, подход к прямому доступу к памяти изменился. Контроллер DMA уступил место другой микросхеме — ***арбитру шины***, который не участвует в передаче данных; его функция сводится к тому, чтобы устройства, подключённые к шине, могли договориться, кто из них сейчас играет роль «главного» (*bus master*). К этому времени процессоры обзавелись кеш-памятью, так что доступ к шине стал им требоваться реже. Что касается обмена данными между памятью и внешними устройствами, то управление таким обменом в режиме прямого доступа (DMA) взяли на себя контроллеры внешних устройств. Теперь, например, драйвер жёсткого диска сообщает контроллеру, в какую область памяти нужно записать данные, считанные с диска, или из какой области памяти нужно взять данные, чтобы затем записать их на диск, и на этом успокаивается. Контроллер жёсткого диска, получив шину (через арбитра) в своё единоличное распоряжение, сам производит запись данных в память или чтение их из памяти — точно так же, как это умеет делать центральный процессор.

Довольно интересную проблему порождает здесь использование виртуальной адресации памяти. Если с диска нужно будет прочитать, скажем, мегабайт данных, то операционной системе, скорее всего, не удастся найти 256 свободных страниц, идущих *подряд*; между тем контроллеры периферийных устройств не знают никаких адресов, кроме физических, так что область оперативной памяти, участвующая в работе DMA, должна быть непрерывной. Это несколько ограничивает возможности DMA и вынуждает ядра операционных систем по возможности избегать фрагментации физической памяти.

Литература

[1] А. М. Робачевский. Операционная система Unix. Изд-во «BHV—Санкт-Петербург», Санкт-Петербург, 1997.

[2] Уильям Стивенс. UNIX: Взаимодействие процессов. СПб.: Питер, 2002.

[3] У. Р. Стивенс. UNIX: Разработка сетевых приложений. СПб.: Питер, 2004.

[4] Эрик С. Реймонд. Искусство программирования для Unix. М.: изд-во Вильямс, 2005. *В оригинале на английском языке книга доступна в сети Интернет по адресу* `http://www.faqs.org/docs/artu/`

[5] Э. Танненбаум. Современные операционные системы. 2-е издание. СПб.: Питер, 2002.

[6] Вирт Н. Алгоритмы и структуры данных. Пер. с англ. 2-е изд. СПб.: Невский Диалект, 2001. 352 с. ISBN 5-7940-0065-1

[7] Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦНМО, 2000. 960 с. ISBN 5-900916-37-5.

[8] Кнут Д. Искусство программирования, т. 3. Сортировка и поиск, 2-е изд. Пер. с англ. М.: Издательский дом «Вильямс», 2000. 832 с. ISBN 5-8459-0082-4

[9] Wolfgang Mauerer. Professional Linux Kernel Architecture. Wiley Publishing, Inc., Indianapolis, 2008. *Полный текст книги доступен в сети Интернет; используйте поисковые машины.*

[10] Linus Åkesson. The TTY demystified. 25-Jul-2008, `http://www.linusakesson.net/programming/tty/`

[11] Bert Hubert. The ultimate `SO_LINGER` page, or: why is my tcp not reliable // Bert Hubert finally blogs,

18-Jan-2009. httpt://blog.netherlabs.nl/articles/2009/01/18/the-ultimate-so_linger-page-or-why-is-my-tcp-not-reliable

[12] Edward A. Lee. The Problem with Threads. UC Berkeley, January 10, 2006, Technical Report No. UCB/EECS-2006-1 http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html

[13] Ulrich Drepper. Futexes Are Tricky. November 5, 2011. http://www.akkadia.org/drepper/futex.pdf

[14] Andy Walker. Mandatory File Locking For The Linux Operating System. 15 April 1996 (Updated September 2007). https://www.kernel.org/doc/Documentation/filesystems/mandatory-locking.txt

[15] Keith Adams, Ole Agesen. Comparison of Software and Hardware Techniques for x86 Virtualization // ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

[16] CFS Scheduler // Linux kernel documentation, file scheduler/sched-design-CFS.txt. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

Предметный указатель

ГЛАВНЫЕ СПОНСОРЫ ПРОЕКТА

список наиболее крупных пожертвований

I: 114999 (5000+10000+99999), **Nikolay Ksenev**

II: 45763 (19972+25791), *анонимно*

III: 41500, *анонимно*

IV: 25000, **unDEFER**

V: 21600, *анонимно*

VI: 21048 (4×5262), **os80**

VII: 17216, *анонимно*

VIII: 15000 (3000+7000+5000), *анонимно*

IX: 13000 (2000+8000+3000), **Сергей Сетченков**

X: 12120 (3333+5699+3088), **Антон Хван**

XI: 12000 (2000+10000), **Masutacu**

XII: 10000, **Аня «canja» Ф.**

XIII: 10000 (2×1500+2000+5000), **Георгий Мошкин**

XIV: 9496 (2×1500+2048+4448), **Шер Арсений Владимирович**

XV: 8080, **Дергачёв Борис Николаевич**

XVI: 8072 (5053+3019), *анонимно*

XVII: 8000, **Смирнов Денис**

XVIII: 8000 (4000+4000), **Татьяна 'Vikora' Алпатова**

XIX: 8000 (5000+3000), **Катерина Галкина**

СТОЛЯРОВ Андрей Викторович

ПРОГРАММИРОВАНИЕ: ВВЕДЕНИЕ В ПРОФЕССИЮ
III: СИСТЕМЫ И СЕТИ
Учебно-методическое издание

Рисунок и дизайн обложки Елены Доменновой
Корректор Екатерина Ясиницкая

Напечатано с готового оригинал-макета

Подписано в печать 14.07.2017 г.
Формат 60x90 1/16. Усл.печ.л. 25. Тираж 316 экз. Изд.№ 180.

Издательство ООО “МАКС Пресс”
Лицензия ИД № 00510 от 01.12.99 г.

119992 ГСП-2, Москва, Ленинские горы,
МГУ им. М.В.Ломоносова, 2-й учебный корпус, 527 к.
Тел. 939-3890, 939-3891. Тел./Факс 939-3891

Отпечатано в ППП «Типография «Наука»
121099, Москва, Шубинский пер., 6
Заказ №865

Андрей Викторович Столяров (род. 1974) — кандидат физико-математических наук, кандидат философских наук, доцент; работает на кафедре алгоритмических языков факультета вычислительной математики и кибернетики Московского государственного университета имени М. В. Ломоносова.

Третий том серии посвящён операционным системам как явлению: какие услуги они предоставляют пользовательским задачам и как устроены они сами.

http://www.stolyarov.info