

# ОПЕРАЦИОННАЯ СИСТЕМА

# UNIX®

*Принципы организации, идеология и архитектура, пользовательский и программный интерфейсы — все, что объединяет различные версии операционной системы под общим названием UNIX.*

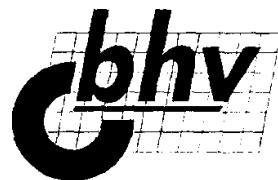
- Архитектура ядра UNIX: файловая подсистема, подсистема управления процессами и памятью, подсистема ввода/вывода.
- Работа пользователя в операционной системе: командный интерпретатор shell, основные команды и утилиты.
- Программный интерфейс UNIX: системные вызовы и основные библиотечные функции для работы с файлами и управления процессами.
- Сетевая поддержка в UNIX: описание и принципы работы протоколов семейства TCP/IP, архитектура сетевой подсистемы, программные интерфейсы сокетов и TLI.



Андрей Робачевский

Операционная система  
**UNIX**

Рекомендовано Министерством общего и профессионального образования Российской Федерации в качестве учебного пособия для студентов высших учебных заведений



Дюссельдорф Киев

Москва Санкт-Петербург

УДК 681.3.06

Книга посвящена семейству операционных систем UNIX и содержит информацию о принципах организации, идеологии и архитектуре, объединяющих различные версии этой операционной системы.

В книге рассматриваются: архитектура ядра UNIX (подсистемы ввода/вывода, управления памятью и процессами, а также файловая подсистема), программный интерфейс UNIX (системные вызовы и основные библиотечные функции), пользовательская среда (командный интерпретатор shell, основные команды и утилиты) и сетевая поддержка в UNIX (протоколов семейства TCP/IP, архитектура сетевой подсистемы, программные интерфейсы сокетов и TLI).

*Для широкого круга пользователей*

**Группа подготовки издания:**

Главный редактор	Екатерина Кондукова
Зав. редакцией	Наталья Таркова
Редактор	Татьяна Темкина
Корректор	Зинаида Дмитриева
Компьютерная верстка	Владислава Сорокина
Дизайн обложки	Дмитрия Солнцева
Зав. производством	Николай Тверских

Рукопись книги подготовлена в Республиканском научном центре компьютерных телекоммуникационных сетей высшей школы.

*Рецензенты:*

Зав. кафедрой "Вычислительная техника" Санкт-Петербургского государственного электротехнического университета д.т.н. профессор Д. В. Пузанков

Зав. кафедрой "Информационные и управляющие системы" Санкт-Петербургского государственного Технического университета д.т.н. профессор И. Г. Черноруцкий

**Робачевский А. М.**

Операционная система UNIX®. - СПб.: БХВ-Петербург, 2002. - 528 с.: ил.

ISBN 5-8206-0030-4

UNIX является зарегистрированным  
знаком консорциума The Open Group

© А. М. Робачевский, 1997

© Рисунки, К. Щукин, 1997

© Оформление, издательство "БХВ-Петербург", 1997

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 24.12.01.

Формат 70×100<sup>1/16</sup>. Печать офсетная. Усл. печ. л. 42,8.

Доп. тираж 5000 экз. Заказ 1383  
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар, № 77.99.1.953.П.950.3.99  
от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов  
в Академической типографии "Наука" РАН.  
199034, Санкт-Петербург, 9 линия, 12.

# СОДЕРЖАНИЕ

<b>О КНИГЕ "ОПЕРАЦИОННАЯ СИСТЕМА UNIX".....</b>	<b>1</b>
НАЗНАЧЕНИЕ книги.....	1
НА КОГО РАССЧИТАНА ЭТА КНИГА?.....	2
ПРИНЯТЫЕ ОБОЗНАЧЕНИЯ.....	2
<b>ВВЕДЕНИЕ.....</b>	<b>3</b>
ИСТОРИЯ СОЗДАНИЯ.....	3
Исследовательские версии UNIX.....	4
ГЕНЕАЛОГИЯ UNIX.....	6
System V UNIX.....	6
System V Release 4 (SVR4).....	7
UNIX компании Berkeley Software Distribution.....	7
OSF/1.....	8
Версии UNIX, использующие микроядро.....	8
Свободно распространяемая система UNIX.....	9
ОСНОВНЫЕ СТАНДАРТЫ.....	9
IEEE и POSIX.....	10
X/Ореп.....	10
SVID.....	11
ANSI.....	11
НЕКОТОРЫЕ ИЗВЕСТНЫЕ ВЕРСИИ UNIX.....	12
AIX.....	13
HP-UX.....	13
IRIX.....	13
Digital UNIX.....	13
SCO UNIX.....	13
Solaris.....	14
ПРИЧИНЫ ПОПУЛЯРНОСТИ UNIX.....	14
ОБЩИЙ взгляд на архитектуру UNIX.....	15
Ядро системы.....	16
Файловая подсистема.....	17
Подсистема управления процессами.....	18
Подсистема ввода/вывода.....	18
<b>ГЛАВА 1. РАБОТА В ОПЕРАЦИОННОЙ СИСТЕМЕ UNIX.....</b>	<b>19</b>
ФАЙЛЫ И ФАЙЛОВАЯ СИСТЕМА.....	20
Типы файлов.....	21
Сокеты.....	25
Структура файловой системы UNIX.....	26
Корневой каталог.....	26

/bin.....	27
/dev.....	27
/etc.....	27
/lib.....	27
/lost+found.....	27
/mnt.....	28
/u или /home.....	28
/usr.....	28
/var.....	28
/tmp.....	28
Владельцы файлов.....	28
Права доступа к файлу.....	30
Дополнительные атрибуты файла.....	35
ПРОЦЕССЫ.....	38
Программы и процессы.....	38
Типы процессов.....	39
Системные процессы.....	39
Демоны.....	40
Прикладные процессы.....	40
Атрибуты процесса.....	41
Идентификатор процесса Process ID (PID).....	41
Идентификатор родительского процесса Parent Process ID (PPID).....	41
Приоритет процесса (Nice Number).....	41
Терминальная линия (TTY).....	41
Реальный (RID) и эффективный (EUID) идентификаторы пользователя.....	41
Реальный (RGID) и эффективный (EGID) идентификаторы группы.....	42
Жизненный путь процесса.....	42
Сигналы.....	44
УСТРОЙСТВА.....	47
Файлы блочных устройств.....	47
Файлы символьных устройств.....	47
Мнемоника названий специальных файлов устройств	
в файловой системе UNIX.....	49
ПОЛЬЗОВАТЕЛИ СИСТЕМЫ.....	50
Атрибуты пользователя.....	51
Пароли.....	54
Стандартные пользователи и группы.....	55
ПОЛЬЗОВАТЕЛЬСКАЯ СРЕДА UNIX.....	56
Командный интерпретатор shell.....	56
Синтаксис языка Bourne shell.....	59
Общий синтаксис скрипта.....	59
Переменные.....	60
Встроенные переменные.....	64
Перенаправление ввода/вывода.....	66
Команды, функции и программы.....	68
Подстановки, выполняемые командным интерпретатором.....	71

## СОДЕРЖАНИЕ

Запуск команд	73
Условные выражения	74
Команда <i>test</i>	75
Циклы	77
Селекторы	78
Ввод	79
Система управления заданиями	80
Основные утилиты UNIX	82
Утилиты для работы с файлами	82
Утилиты для управления процессами	86
Об администрировании UNIX	88
Ситуация 1. Нехватка дискового пространства	89
Ситуация 2. Избыточная загрузка процессора	89
Ситуация 3. Регистрация новых пользователей	90
Ситуация 4. Авария загрузочного диска	90
Ситуация 5. Слабая производительность сети	91
Ситуация 6. "Глупые" вопросы пользователей	91
Ситуация 7. Установка новой версии операционной системы	91
Ситуация 8. Пользователям необходима электронная телефонная книга	92
ЗАКЛЮЧЕНИЕ	92
<b>ГЛАВА 2. СРЕДА ПРОГРАММИРОВАНИЯ UNIX</b>	<b>93</b>
ПРОГРАММНЫЙ ИНТЕРФЕЙС UNIX	93
Системные вызовы и функции стандартных библиотек	93
Обработка ошибок	95
СОЗДАНИЕ ПРОГРАММЫ	100
Исходный текст	100
Заголовки	101
Компиляция	105
Форматы исполняемых файлов	107
Формат ELF	108
Формат COFF	112
ВЫПОЛНЕНИЕ ПРОГРАММЫ В ОПЕРАЦИОННОЙ СИСТЕМЕ UNIX	114
Запуск С-программы	114
Завершение С-программы	118
РАБОТА с ФАЙЛАМИ	121
Основные системные функции для работы с файлами	121
Функция <i>open(2)</i>	122
Функция <i>creat(2)</i>	124
Функция <i>close(2)</i>	125
Функции <i>dup(2)</i> и <i>dup2(2)</i>	125
Функция <i>lseek(2)</i>	126
Функция <i>read(2)</i> и <i>readv(2)</i>	126
Функции <i>write(2)</i> и <i>writev(2)</i>	127

Функция <i>pipe(2)</i> .....	128
Функция <i>fcntl(2)</i> .....	129
Стандартная библиотека ввода/вывода.....	130
Связи.....	133
Файлы, отображаемые в памяти.....	137
Владение файлами.....	140
Права доступа.....	140
Перемещение по файловой системе.....	142
Метаданные файла.....	144
<b>ПРОЦЕССЫ.....</b>	<b>146</b>
Идентификаторы процесса.....	147
Выделение памяти.....	150
Создание и управление процессами.....	154
Сигналы.....	160
Надежные сигналы.....	166
Группы и сеансы.....	173
Текущие и фоновые группы процессов.....	175
Ограничения.....	177
<b>ПРИМЕРЫ ПРОГРАММ.....</b>	<b>180</b>
Демон.....	180
Командный интерпретатор.....	184
<b>ЗАКЛЮЧЕНИЕ.....</b>	<b>186</b>

<b>ГЛАВА 3. ПОДСИСТЕМА УПРАВЛЕНИЯ ПРОЦЕССАМИ.....</b>	<b>187</b>
ОСНОВЫ УПРАВЛЕНИЯ ПРОЦЕССОМ.....	188
Структуры данных процесса.....	190
Состояния процесса.....	191
ПРИНЦИПЫ УПРАВЛЕНИЯ ПАМЯТЬЮ.....	195
Виртуальная и физическая память.....	197
Сегменты.....	199
Страницочный механизм.....	202
Адресное пространство процесса.....	204
УПРАВЛЕНИЕ ПАМЯТЬЮ ПРОЦЕССА.....	206
Области.....	207
Замещение страниц.....	210
ПЛАНИРОВАНИЕ ВЫПОЛНЕНИЯ ПРОЦЕССОВ.....	216
Обработка прерываний таймера.....	217
Отложенные вызовы.....	218
Алармы.....	219
Контекст процесса.....	221
Принципы планирования процессов.....	222
СОЗДАНИЕ ПРОЦЕССА.....	226
ЗАПУСК новой ПРОГРАММЫ.....	230
ВЫПОЛНЕНИЕ в РЕЖИМЕ ЯДРА.....	233
СОН И ПРОБУЖДЕНИЕ.....	234

## СОДЕРЖАНИЕ

ЗАВЕРШЕНИЕ ВЫПОЛНЕНИЯ ПРОЦЕССА.....	235
СИГНАЛЫ.....	236
Группы и сеансы.....	236
Управление сигналами.....	237
Отправление сигнала.....	237
Доставка и обработка сигнала.....	238
ВЗАИМОДЕЙСТВИЕ МЕЖДУ ПРОЦЕССАМИ.....	240
Каналы.....	242
FIFO.....	243
Идентификаторы и имена в IPC.....	245
Сообщения.....	248
Семафоры.....	253
Разделяемая память.....	258
Межпроцессное взаимодействие в BSD UNIX. Сокеты.....	264
Программный интерфейс сокетов.....	265
Пример использования сокетов.....	274
Сравнение различных систем межпроцессного взаимодействия.....	277
ЗАКЛЮЧЕНИЕ.....	278
 <b>ГЛАВА 4. ФАЙЛОВАЯ ПОДСИСТЕМА.....</b>	<b>279</b>
БАЗОВАЯ ФАЙЛОВАЯ СИСТЕМА SYSTEM V.....	280
Суперблок.....	281
Индексные дескрипторы.....	282
Имена файлов.....	285
Недостатки и ограничения.....	287
ФАЙЛОВАЯ СИСТЕМА BSD UNIX.....	288
Каталоги.....	291
АРХИТЕКТУРА ВИРТУАЛЬНОЙ ФАЙЛОВОЙ СИСТЕМЫ.....	292
Виртуальные индексные дескрипторы.....	293
Монтирование файловой системы.....	296
Трансляция имен.....	303
ДОСТУП К ФАЙЛОВОЙ СИСТЕМЕ.....	304
Файловые дескрипторы.....	306
Файловая таблица.....	307
Блокирование доступа к файлу.....	309
БУФЕРНЫЙ кэш.....	311
Внутренняя структура буферного кэша.....	313
Операции ввода/вывода.....	314
Кэширование в SVR4.....	317
ЦЕЛОСТНОСТЬ ФАЙЛОВОЙ СИСТЕМЫ.....	317
ЗАКЛЮЧЕНИЕ.....	321
 <b>ГЛАВА 5. ПОДСИСТЕМА ВВОДА/ВЫВОДА.....</b>	<b>322</b>
ДРАЙВЕРЫ УСТРОЙСТВ.....	323

Типы драйверов.....	323
Базовая архитектура драйверов.....	325
Файловый интерфейс.....	333
Клоны.....	335
Встраивание драйверов в ядро.....	338
БЛОЧНЫЕ УСТРОЙСТВА.....	340
СИМВОЛЬНЫЕ УСТРОЙСТВА.....	342
Интерфейс доступа низкого уровня.....	343
Буферизация.....	344
АРХИТЕКТУРА ТЕРМИНАЛЬНОГО ДОСТУПА.....	346
Псевдотерминалы.....	348
ПОДСИСТЕМА STREAMS.....	350
Архитектура STREAMS.....	352
Модули.....	356
Сообщения.....	357
Типы сообщений.....	361
Передача данных.....	362
Управление передачей данных.....	364
Драйвер.....	368
Головной модуль.....	369
Доступ к потоку.....	371
Создание потока.....	372
Управление потоком.....	375
Мультиплексирование.....	377
ЗАКЛЮЧЕНИЕ.....	380
<b>ГЛАВА 6. ПОДДЕРЖКА СЕТИ В ОПЕРАЦИОННОЙ СИСТЕМЕ UNIX.....</b>	<b>382</b>
СЕМЕЙСТВО ПРОТОКОЛОВ TCP/IP.....	383
Краткая история TCP/IP.....	384
Архитектура TCP/IP.....	386
Общая модель сетевого взаимодействия OSI.....	391
ПРОТОКОЛ IP.....	393
Адресация.....	398
ПРОТОКОЛЫ ТРАНСПОРТНОГО УРОВНЯ.....	400
User Datagram Protocol (UDP).....	402
Transmission Control Protocol (TCP).....	404
Состояния TCP-сессии.....	406
Передача данных.....	410
Стратегии реализации TCP.....	413
Синдром "глупого окна".....	414
Медленный старт.....	416
Устранение затора.....	417
Повторная передача.....	419
ПРОГРАММНЫЕ ИНТЕРФЕЙСЫ.....	420
Программный интерфейс сокетов.....	420

## СОДЕРЖАНИЕ

Программный интерфейс TLI.....	426
Программный интерфейс высокого уровня.....	
Удаленный вызов процедур.....	440
Передача параметров.....	442
Связывание (binding).....	443
Обработка особых ситуаций (exception).....	444
Семантика вызова.....	444
Представление данных.....	445
Сеть.....	445
Как это работает?.....	446
log.x.....	447
log.h.....	448
log.c.....	448
client.c.....	449
ПОДДЕРЖКА СЕТИ В BSD UNIX.....	452
Структуры данных.....	453
Маршрутизация.....	458
Реализация TCP/IP.....	464
Модуль IP.....	466
Модуль UDP.....	468
Модуль TCP.....	469
ПОДДЕРЖКА СЕТИ В UNIX SYSTEM V.....	470
Интерфейс TPI.....	472
Взаимодействие с прикладными процессами.....	481
Интерфейс DLPI.....	487
Доступ к среде передачи.....	490
Протокол LLC.....	492
Инкапсуляция IP.....	493
Внутренняя архитектура.....	493
Примитивы DLPI.....	497
ЗАКЛЮЧЕНИЕ.....	501
<b>ПРИЛОЖЕНИЕ А. ЭЛЕКТРОННЫЙ СПРАВОЧНИК MAN(1).....</b>	<b>503</b>
<b>ПРИЛОЖЕНИЕ Б. ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ ОБ ОПЕРАЦИОННОЙ СИСТЕМЕ UNIX.....</b>	<b>504</b>
КНИГИ.....	504
ИНФОРМАЦИЯ в INTERNET.....	505
<b>ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ.....</b>	<b>509</b>

## **Выражение признательности**

Работая над книгой, я много раз продумывал содержание этого приятного раздела, каждый раз добавляя в него новые и новые имена людей, без помощи которых эта книга вряд ли увидела бы свет.

В первую очередь это заслуга директора издательства "ВНВ—Санкт-Петербург" Вадима Сергеева и моего коллеги, сотрудника Вузтелекомцентра и автора замечательного справочника "Желтые страницы Internet. Русские ресурсы" Алексея Сигалова. Именно они убедили меня в том, что такая книга окажется полезной и вдохновили взяться за перо.

Я благодарен руководителям Вузтелекомцентра Владимиру Васильеву и Сергею Хоружникову за помощь и внимание к работе над книгой. Их поддержка и терпимое отношение к выполнению моих основных обязанностей директора по развитию Вузтелекомцентра позволили выполнить эту работу.

Без помощи Кирилла Щукина книге грозила опасность увидеть свет без иллюстраций, что вряд ли сделало бы ее более ясной. Его терпение и профессионализм позволили превратить туманные наброски в полноценные схемы, от которых книга значительно выиграла.

Я неоднократно обращался за советом к экспертам по UNIX и прежде всего к моему коллеге Константину Федорову. Его ценные замечания и рекомендации помогли мне довести книгу до ее настоящего вида.

Я также хотел бы выразить признательность специалистам фирмы OLLY, и в особенности ее техническому директору Виталию Кузьмичеву, чьи советы и консультации благотворно повлияли на содержание этой книги.

Я также хотел бы выразить глубокую признательность рецензентам этой книги — зав. кафедрой "Вычислительная техника" Санкт-Петербургского государственного электротехнического университета д. т. н. профессору Д. В. Пузанкову и зав. кафедрой "Информационные и управляемые системы" Санкт-Петербургского государственного Технического университета д. т. н. профессору И. Г. Черноруцкому за полезные замечания.

Я хотел бы также поблагодарить зав. редакции издательства "ВНВ—Санкт-Петербург" Елизавету Кароник, которая первой ознакомилась с рукописью и вынесла положительный вердикт, за кредит доверия и координацию работ по созданию книги. Я хочу выразить благодарность Татьяне Темкиной за ее великолепную работу по редактированию книги. Случалось, что отдельные страницы рукописи содержали меньше основного материала, чем редакторской правки, с которой я, как правило, всегда соглашался.

Я не могу не выразить признательность моим коллегам по работе Владимиру Парfenову, Юрию Гугелю, Юрию Кирчину, Нине Рубиной, дружеская поддержка которых была так кстати.

И, конечно, я хотел бы поблагодарить моих жену и дочь за их терпение и веру в успешное завершение этой работы. Я также должен извиниться перед ними за то, что этот труд отнял у меня значительную часть времени, по праву принадлежащего им.

*Автор*

*Посвящается моим близким*

## **О книге "Операционная система UNIX"**

### **Назначение книги**

Данная книга не является заменой справочников и различных руководств по операционной системе UNIX. Более того, сведения, представленные в книге, подчас трудно найти в документации, поставляемой с операционной системой. Эти издания насыщены практическими рекомендациями, скрупулезным описанием настроек тех или иных подсистем, форматов вызова команд и т. п. При этом за кадром часто остаются такие вопросы, как внутренняя архитектура отдельных компонентов системы, их взаимодействие и принципы работы. Без знания этой "анатомии" работа в операционной системе превращается в использование заученных команд, а неизбежные ошибки приводят к необъяснимым последствиям. С другой стороны, в данной книге вопросам администрирования UNIX, настройке конкретных подсистем и используемым командам уделено значительно меньше внимания. Цель данной книги заключается в изложении основ организации операционной системы UNIX. Следует иметь в виду, что именем UNIX обозначается значительное семейство операционных систем, каждая из которых имеет свое название и присущие только ей особенности. В этой книге сделана попытка выделить то общее, что составляет "генотип" UNIX, а именно: базовый пользовательский и программный интерфейсы, назначение основных компонентов, их архитектуру и взаимодействие, и на основе этого представить систему в целом. В то же время там, где это имеет значение, приводятся ссылки на конкретную версию UNIX. Для иллюстрации отдельных положений использовались следующие операционные системы: Solaris 2.5 фирмы Sun Microsystems, SCO ODT 5.0 фирмы Santa Cruz Operation, BSDi/386 фирмы Berkeley Software Design.

Рождению этой книги предшествовал более чем трехлетний опыт чтения лекций по системе UNIX студентам третьего курса Санкт-Петербургского института точной механики и оптики (технического университета), а также вводного курса для пользователей и администраторов UNIX в различных организациях. Большая часть материала этих курсов нашла свое отражение в книге.

Книга может оказаться полезной при подготовке ряда лекционных программ по операционной системе UNIX и основам организации операционных систем в целом. Материал главы 1 является хорошей основой для вводного курса по UNIX. В нем представлены основные понятия и организация операционной системы в целом. В этой же главе приведены основные сведения о пользовательском интерфейсе и языке программирования командного интерпретатора shell.

Материал главы 2 может быть использован в курсах по программированию. Подробное обсуждение основных системных вызовов и библиотечных функций дает достаточно полное представление о программном интерфейсе этой операционной системы. Приведенные примеры иллюстрируют обсуждаемые вопросы и могут найти свое отражение в лабораторном практикуме.

Главы 3—6 содержат более детальное обсуждение отдельных компонентов UNIX: файловой подсистемы, подсистемы управления процессами и памятью, подсистемы ввода/вывода. Эти сведения подойдут как для углубленного курса по UNIX, так и для курса по принципам организации операционных систем. Отдельные части главы 6 могут быть также включены в курс по компьютерным сетям.

Книга может использоваться и в качестве учебного пособия для студентов старших курсов по специальностям "Информатика и вычислительная техника", "Прикладная

математика и информатика" (при подготовке бакалавров) и по специальности "Вычислительные машины, комплексы системы и сети" (при подготовке инженеров) она может быть полезной при подготовке магистров и аспирантов, а также всем студентам, специализирующимся в области компьютерных технологий.

Книга также является хорошим подспорьем для системных программистов и администраторов UNIX. Надеюсь, что более пристальный взгляд на внутреннюю организацию системы поможет им эффективнее решать поставленные задачи и откроет новые горизонты для экспериментов.

Наконец, книга может оказаться интересной для широкого круга пользователей, желающих побольше узнать об этой операционной системе.

## На кого рассчитана эта книга?

Бессмысленно разбираться в операционной системе, не работая с ней. Прежде всего, знание операционной системы, ее организации и структуры необходимо администратору, т.е. человеку, отвечающему за ее сопровождение и настройку. Задачи администратора многочисленны — от регистрации пользователей до конфигурации сети, от создания резервных копий системы до настройки производительности. Без понимания принципиального устройства операционной системы решение всех этих задач превращается в заучивание команд и пунктов меню, а нештатные ситуации вызывают панику.

Знание операционной системы нужно разработчику программного обеспечения. От того, насколько эффективно используются ресурсы операционной системы, зависит быстродействие вашей программы. Не понимая принципов работы, легко запутаться в тонкостях системных вызовов и библиотечных функций. Если же вы работаете с ядром системы — например, разрабатываете драйвер устройства, — без знания системы вы не продвинетесь ни на шаг.

Наконец, если вы просто пользователь, то знание операционной системы ограничивается теми задачами, которые вам необходимо решать в процессе работы. Скорее всего, это несколько команд, а если вы работаете с графической оболочкой, то и этого вам не понадобится. Но так ли приятно работать с черным ящиком?

## Принятые обозначения

Системные вызовы, библиотечные функции, команды shell выделены в тексте курсивом, например *open(2)*, *cat(l)* или *printf(3S)*. В скобках указывается раздел электронного справочника *man(1)* (описание справочника приведено в приложении А).

Структуры данных, переменные и внутренние функции подсистем ядра, исходные тексты программ и примеры работы в командной строке напечатаны шрифтом фиксированной ширины. Например, *d\_open()*, *sleep()* или пример программы:

```
int main()
{
    exit();
}
```

В примерах работы в командной строке ввод пользователя выделен полужирным шрифтом фиксированной ширины, например:

```
$ passwd
Enter old password:
```

Имена файлов выделены полужирным начертанием, например */etc/passwd* или *<sys/user.h>*.

Клавиши клавиатуры показаны курсивом и заключены в угловые скобки, например *<Del>* или *<Ctrl>+<C>* (в последнем случае показана комбинация клавиш).

# Введение

Скоро исполнится 30 лет с момента создания операционной системы UNIX. Изначально созданная для компьютера PDP-7 с 4 килобайтами оперативной памяти, сегодня UNIX работает на множестве аппаратных платформ, начиная с обыкновенного PC и заканчивая мощными много-процессорными системами и суперкомпьютерами.

Система UNIX была создана небольшой группой разработчиков, тысячи людей вложили в нее свой талант, десятки тысяч обогатили приложениями, и сегодня сотни тысяч людей используют эту операционную систему в своей деятельности.

За время своего существования система UNIX претерпела значительные изменения, стала мощней, сложней и удобней. Однако основные идеи сохранились, удивляя нас своим изяществом и простотой. Именно они определяют "генотип" операционной системы, позволяя увидеть за красивыми названиями различных версий лаконичное слово UNIX. Именно изящество и простота этих идей являются основой жизненной силы UNIX, ее способности всегда идти в ногу со временем.

## История создания

В 1965 году Bell Telephone Laboratories (подразделение AT&T) совместно с General Electric Company и Массачусетским институтом технологии (MIT) начали разрабатывать новую операционную систему, названную MULTICS (MULTiplexed Information and Computing Service). Перед участниками проекта стояла цель создания многозадачной операционной системы разделения времени, способной обеспечить одновременную работу нескольких сотен пользователей. От Bell Labs в проекте приняли участие два сотрудника — Кен Томпсон (Ken Thompson) и Дэннис Ритчи (Dennis Ritchie). Хотя система MULTICS так и не была завершена (в 1969 году Bell Labs вышла из проекта), она стала предтечей операционной системы, впоследствии получившей название UNIX.

Однако Томпсон, Ритчи и ряд других сотрудников продолжили работу над созданием удобной среды программирования. Используя идеи и разработ-

ки, появившиеся в результате работы над MULTICS, они создали в 1969 году<sup>1</sup> небольшую операционную систему, включавшую файловую систему, подсистему управления процессами и небольшой набор утилит. Система была написана на ассемблере и применялась на компьютере PDP-7. Эта операционная система получила название UNIX, созвучное MULTICS и придуманное другим членом группы разработчиков, Брайаном Керниганом (Brian Kernighan).

Хотя ранняя версия UNIX много обещала, она не смогла бы реализовать весь свой потенциал без применения в каком-либо реальном проекте. И такой проект нашелся. Когда в 1971 году патентному отделу Bell Labs понадобилась система обработки текста, в качестве операционной системы была выбрана UNIX. К тому времени система UNIX была перенесена на более мощный PDP-11, да и сама немного подросла: 16К занимала собственно система, 8К отводились прикладным программам, максимальный размер файла был установлен равным 64К при 512К дискового пространства.

Вскоре после создания первых ассемблерных версий Томпсон начал работать над компилятором для языка FORTRAN, а в результате разработал язык B. Это был интерпретатор со всеми свойственными интерпретатору ограничениями, и Ритчи переработал его в другой язык, названный C, позволявший генерировать машинный код. В 1973 году ядро операционной системы было переписано на языке высокого уровня C, — неслыханный до этого шаг, оказавший громадное влияние на популярность UNIX. Это означало, что теперь система UNIX может быть перенесена на другие аппаратные платформы за считанные месяцы, кроме того, значительная модернизация системы и внесение изменений не представляли особых трудностей. Число работающих систем в Bell Labs превысило 25, и для сопровождения UNIX была сформирована группа UNIX System Group (USG).

## Исследовательские версии UNIX

В соответствии с федеральным законодательством AT&T не имела права коммерческого распространения UNIX и использовала ее для собственных нужд, но начиная с 1974 года операционная система стала передаваться университетам для образовательных целей.

Операционная система модернизировалась, каждая новая версия снабжалась соответствующей редакцией Руководства Программиста, откуда и сами версии системы получили название редакций (Edition). Всего было выпущено 10 версий-редакций, первая из которых вышла в 1971, а последняя — в 1989 году. Первые семь редакций были разработаны в Bell Labs

<sup>1</sup> Официальной датой рождения UNIX можно считать 1 января 1970 года. Именно с этого момента любая система UNIX отсчитывает свое системное время.

Группой компьютерных исследований (Computer Research Group, CRG) и предназначались для компьютеров PDP-11, позже — для VAX. Другая группа, UNIX System Group, отвечала за сопровождение системы. Третья группа (Programmer's WorkBench, PWB) занималась разработкой среды программирования, ей мы обязаны появлением системы SCCS, именованных каналов и других важных идей. Вскоре после выпуска Седьмой редакции разработкой системы стала заниматься USG.

Наиболее важные версии:

Первая редакция	1971	Первая версия UNIX, написанная на ассемблере для PDP-11. Включала компилятор B и много известных команд и утилит, в том числе <i>cat(1)</i> , <i>chdir(1)</i> , <i>chmod(1)</i> , <i>cp(1)</i> , <i>ed(1)</i> , <i>find(1)</i> , <i>mail(1)</i> , <i>mkdir(1)</i> , <i>mkfs(1M)</i> , <i>mount(1M)</i> , <i>mv(1)</i> , <i>rm(1)</i> , <i>rmdir(1)</i> , <i>wc(1)</i> , <i>who(1)</i> . В основном использовалась как инструментальное средство обработки текстов для патентного отдела.
Третья редакция	1973	В системе появилась команда <i>cc(1)</i> , запускавшая компилятор C. Число установленных систем достигло 16.
Четвертая редакция	1973	Первая система, в которой ядро написано на языке высокого уровня C.
Шестая редакция	1975	Первая версия системы, доступная за пределами Bell Labs. Система полностью переписана на языке C. С этого времени начинается появление новых версий, разработанных за пределами Bell Labs, и рост популярности UNIX. В частности, эта версия системы была установлена Томпсоном в Калифорнийском университете в Беркли, и на ее основе вскоре была выпущена первая версия BSD (Berkeley Software Distribution) UNIX.
Седьмая редакция	1979	Эта версия включала командный интерпретатор Bourne Shell и компилятор C от Кернигана и Ритчи. Ядро было переписано для упрощения переносимости системы на другие платформы. Лицензия на эту версию была куплена фирмой Microsoft, которая разработала на ее базе операционную систему XENIX.

Популярность UNIX росла, и к 1977 году число работающих систем уже превысило 500. В 1977 году компания Interactive Systems Corporation стала первым VAR (Value Added Reseller) системы UNIX, расширив ее для использования в системах автоматизации. Этот же год стал годом первого портирования UNIX с незначительными изменениями на компьютер, отличный от PDP.

## Генеалогия UNIX

Хотя в книге речь пойдет о системах с общим названием UNIX, стоит оговориться, что обсуждать мы будем различные операционные системы. Не существует некоторой "стандартной" системы UNIX, вместо этого вы столкнетесь с множеством операционных систем, имеющих собственные названия и особенности. Но за этими особенностями и названиями все же нетрудно заметить архитектуру, пользовательский интерфейс и среду программирования UNIX. Объясняется это достаточно просто — все эти операционные системы являются близкими или дальними родственниками. Поэтому знакомство с ними мы начнем с рассказа о генеалогии UNIX.

### System V UNIX

Начиная с 1975 года фирма AT&T начала предоставлять лицензии на использование операционной системы как научно-образовательным учреждениям, так и коммерческим организациям. Поскольку основная часть системы поставлялась в исходных текстах, написанных на языке С, опытным программистам не требовалось детальной документации, чтобы разобраться в архитектуре UNIX. С ростом популярности микропроцессоров другие компании переносили UNIX на различные платформы, но простота и ясность операционной системы искушали многих на ее расширение и модификацию, в результате чего появилось много различных вариантов базовой системы.

Не желая терять инициативу, AT&T в 1982 объединила несколько существующих версий UNIX и создала версию под названием System III. В отличие от редакций, предназначавшихся, в первую очередь, для внутреннего использования и не получивших дальнейшего развития, System III была создана для распространения за пределами Bell Labs и AT&T и положила начало мощной ветви UNIX, которая и сегодня жива и развивается.

В 1983 году Bell Labs выпустила новую версию системы — System V. В 1984 году группа USG была трансформирована в лабораторию (UNIX System Development Laboratory, USDL), которая вскоре выпустила новую модификацию системы — System V Release 2 (SVR2). В этой версии были реализованы такие механизмы управления памятью, как замещение страниц и копирование при записи (copy on write), и представлена система межпроцессного взаимодействия (InterProcess Communication, IPC) с разделяемой памятью, очередью сообщений и семафорами.

В 1987 году появилась следующая версия — System V Release 3 (SVR3). За ее разработку отвечало новое подразделение AT&T — Информационные системы AT&T (AT&T Information Systems, ATTIS). Эта версия отличалась большим набором дополнительных возможностей, включавших:

- Подсистему ввода/вывода, основанную на архитектуре STREAMS.
- Переключатель файловой системы (File System Switch), обеспечивавший одновременную поддержку различных файловых систем.

- Разделяемые библиотеки.
- Программный интерфейс сетевых приложений Transport Layer Interface (ТЫ).

### **System V Release 4 (SVR4)**

В 1989 году была выпущена новая основная версия — System V Release 4. По существу она объединила возможности нескольких известных версий UNIX: SunOS фирмы Sun Microsystems, BSD UNIX компании Berkeley Software Distribution и предыдущих версий System V.

Новые черты системы включали:

- Командные интерпретаторы Korn и C (BSD)
- Символические ссылки
- Систему терминального ввода/вывода, основанную на STREAMS (System V)
- Отображаемые в память файлы (SunOS)
- Сетевую файловую систему NFS и систему вызова удаленной процедуры RPC (SunOS)
- Быструю файловую систему FFS (BSD)
- Сетевой программный интерфейс сокетов (BSD)
- Поддержку диспетчеризации реального времени

Многие компоненты системы были поддержаны стандартами ANSI, POSIX, X/Open и SVID.

### **UNIX компании Berkeley Software Distribution**

Четвертая редакция UNIX была установлена в Калифорнийском университете в Беркли в 1974 году. С этого момента начинает свою историю ветвь UNIX, известная под названием BSD UNIX. Первая версия этой системы основывалась на Шестой редакции и была выпущена в 1978 году. В 1979 году на базе Седьмой редакции была разработана новая версия UNIX — 3BSD. Она явилась первой версией BSD, перенесенной на ЭВМ VAX. В этой системе, в частности, были реализованы виртуальная память (virtual memory) и страничное замещение по требованию (demand paging).

Важным для развития системы явился 1980 год, когда фирма Bolt, Beranek and Newman (BBN) подписала контракт с Отделом перспективных исследовательских проектов (DARPA) Министерства обороны США на разработку поддержки семейства протоколов TCP/IP в BSD UNIX. Эта работа была закончена в конце 1981 года, а ее результаты интегрированы в 4.2BSD UNIX.

Версия 4.2BSD была выпущена в середине 1983 года и включала поддержку работы в сетях, в частности, в сетях Ethernet. Это способствовало широкому распространению локальных сетей, основанных на этой технологии. Система 4.2BSD также позволяла подключиться к сети ARPANET, быстрый рост которой наблюдается с начала 80-х. Разумеется, такая операционная система не могла не пользоваться большой популярностью. К тому же, в отличие от положения в AT&T, где сетевые разработки обычно не выходили за пределы компании, результаты, полученные в Беркли, были широко доступны. Поэтому 4.2BSD стала наиболее популярной системой в исследовательских кругах.

Однако большое количество нововведений привело к тому, что система получилась сырой, содержала ряд ошибок и имела определенные проблемы с быстродействием. В 1986 году была выпущена следующая версия — 4.3BSD, более надежная и с лучшей производительностью. В период с 1986 по 1990 год в систему было внесено много дополнений, включая сетевую файловую систему NFS, виртуальную файловую систему VFS, отладчик ядра и мощную поддержку сети.

Последними версиями, выпущенными в Беркли, стали системы 4.4BSD и BSD Lite, появившиеся в 1993 году.

## OSF/1

В 1988 году AT&T и Sun Microsystems заключили соглашение о сотрудничестве в области разработки будущих версий System V. В ответ на это ряд компаний, производящих компьютеры или имеющих отношение к вычислительной технике, включая IBM, DEC, Hewlett-Packard, создали организацию под названием Open Software Foundation (OSF), целью которой являлась разработка независимой от AT&T версии операционной системы. Результатом деятельности этой организации стала операционная система OSF/1. Хотя ряд коммерческих операционных систем связывают себя с этой ветвью, нельзя сказать, что OSF/1 явилась новым словом в мире UNIX. Скорее, это был политический шаг, призванный снизить доминирующую роль ряда фирм, занимавшихся разработкой UNIX System V.

## Версии UNIX, использующие микроядро

Идея микроядра заключается в сведении к минимуму функций, выполняемых ядром операционной системы, и, соответственно, предоставляемых базовых услуг. При этом основные компоненты операционной системы являются модулями, работающими на базе микроядра. С одной стороны, такой подход делает микроядро более универсальным, позволяя конструировать специализированные операционные системы, а с другой, — упрощает настройку и конфигурирование.

Наиболее известны следующие версии микроядра:

- Микроядро Mach, разработанное в университете Карнеги-Меллона. Сегодня Mach используется в системе OSF/1 фирмы DEC для серверов с процессорами Alpha, а также в операционной системе Workplace фирмы IBM.
- Микроядро Chorus. На базе этого микроядра созданы системы Chorus/MiX V.3 и Chorus/MiX V.4, являющиеся "серверизацией" SVR3 и SVR4. При этом ядро UNIX разделено на множество серверов, выполняющихся под управлением микроядра, причем эти серверы могут находиться как на одном компьютере, так и быть распределены в сети.

## **Свободно распространяемая система UNIX**

Достаточно дешевый PC и свободно распространяемая система UNIX делают эту систему сегодня доступной практически каждому.

Очень популярная версия UNIX для PC, называемая Minix, была разработана Энди Тэненбаумом (Andy Tanenbaum) как приложение к его книге по архитектуре UNIX. Книга Тэненбаума содержит полные листинги исходных текстов системы. Дополнительный набор дисков позволяет установить Minix даже на PC с процессором 8086 (если найдется такой компьютер).

В последнее время все большую популярность приобретает свободно распространяемая версия UNIX под названием Linux, разработанная исследователем университета Хельсинки Линусом Торвальдсом (Linus Torvalds). Разработанная "с нуля" для процессора Intel i386, сегодня она перенесена на ряд других аппаратных платформ, включая серверы Alpha фирмы DEC.

## **Основные стандарты**

UNIX явилась первой действительно переносимой системой, и в этом одна из причин ее успеха.

Как в ранние, бесплатно распространяемые, исследовательские версии, так и в сегодняшние коммерческие и свободно распространяемые версии UNIX постоянно вносятся изменения. С одной стороны, это расширяет возможности системы, делает ее мощнее и надежнее, с другой — ведет к значительным различиям между существующими версиями, отсутствию канонического UNIX.

Чем больше появлялось версий UNIX (и особенно коммерческих), тем очевиднее становилась необходимость стандартизации системы. Наличие стандартов облегчает переносимость приложений и защищает как пользователей, так и производителей. В результате возникло несколько органи-

заций, связанных со стандартизацией, и был разработан ряд стандартов, оказывающих влияние на развитие UNIX.

## IEEE и POSIX

В 1980 году была создана инициативная группа под названием /usr/group с целью стандартизации программного интерфейса UNIX, т. е. формального определения услуг, предоставляемых операционной системой приложениям. Решение этой задачи упростило бы переносимость приложений между различными версиями UNIX. Такой стандарт был создан в 1984 году и использовался комитетом ANSI, отвечающим за стандартизацию языка С, при описании библиотек. Однако с ростом числа версий операционной системы эффективность стандарта уменьшилась, и через год, в 1985 году, был создан Portable Operating System Interface for Computing Environment, сокращенно POSIX (переносимый интерфейс операционной системы для вычислительной среды).

В 1988 году группой был разработан стандарт POSIX 1003.1-1988, который определил программный интерфейс приложений (Application Programming Interface, API). Этот стандарт нашел широкое применение во многих операционных системах, в том числе и с архитектурой, отличной от UNIX. Спустя два года стандарт был принят как стандарт ШЕЕ 1003.1-1990. Заметим, что поскольку этот стандарт определяет интерфейс, а не конкретную реализацию, он не делает различия между системными вызовами и библиотечными функциями, называя все элементы программного интерфейса просто функциями.

Другими наиболее значительными стандартами POSIX, относящимися к UNIX, являются:

POSIX 1003.2-1992	Включает определение командного интерпретатора UNIX и набора утилит
POSIX 1003.1b-1993	Содержит дополнения, относящиеся к поддержке приложений реального времени
POSIX 1003.1c-1995	Включает определения "нитей" (threads) POSIX, известных также как pthreads

## X/Open

В 1984 году ряд европейских компьютерных компаний сформировал некоммерческую организацию, получившую название X/Open. Название полностью отражает цель этой организации — разработку общего набора интерфейсов операционной системы, согласованного между различными производителями, и создание действительно открытых систем, для которых стоимость переносимости приложений как между различными вер-

сиями одной операционной системы, так и между системами различных производителей была бы минимальной.

Основной задачей организации X/Open являлось согласование и утверждение стандартов для создания общего программного интерфейса и программной среды для приложений. В 1992 году появился документ, известный под названием X/Open Portability Guide версии 3 или XPG3, который включал POSIX 1003.1-1988 и стандарт на графическую систему X Window System, разработанную в Массачусеттском институте технологии.

В дальнейшем интерфейсы XPG3 были расширены, включив базовые API систем BSD и System V (SVID), в том числе и архитектуру STREAMS. В результате была выпущена спецификация, ранее известная как Spec 11/70, а в 1994 году получившая название XPG4.2.

В 1996 году объединение усилий X/Open и OSF привело к созданию консорциума The Open Group, продолжившего разработки в области открытых систем. В качестве примера можно привести такие направления, как дальнейшая разработка пользовательского интерфейса, Common Desktop Environment (CDE), и его сопряжение со спецификацией графической оболочки Motif. Другим примером является разработка стандартных интерфейсов для распределенной вычислительной среды Distributed Computing Environment (DCE), работа над которой была начата OSF.

## **SVID**

Вскоре после выхода в свет в 1984 году версии SVR2, группа USG выпустила документ под названием System V Interface Definition, SVID, в котором описывались внешние интерфейсы UNIX версий System V. По существу, этот труд (в двух томах) определял соответствие операционной системы версии System V.

В дополнение к SVID был выпущен т. н. System V Verification Suite, SVVS, — набор тестовых программ, позволяющих производителям получить ответ, достойна ли их система права носить имя System V.

С появлением SVR4 было выпущено новое издание SVID (уже в четырех томах) и, соответственно, новый SWS.

## **ANSI**

В конце 1989 года Американским национальным институтом стандартов (American National Standards Institute, ANSI) был утвержден стандарт X3.159-1989 языка программирования С. Целью появления этого стандарта являлось улучшение переносимости программ, написанных на языке С, в различные операционные системы (не только UNIX). Стандарт определяет не только синтаксис и семантику языка, но и содержимое стандартной библиотеки.

## Некоторые известные версии UNIX

Сегодня существуют десятки различных операционных систем, которые можно называть UNIX. В основном, это коммерческие версии, в которых создатели пытались как можно эффективнее решить вопросы реализации той или иной подсистемы. Во многих случаях, производитель операционной системы является и производителем аппаратной платформы, для которой эта система предназначена. В качестве примеров можно привести операционные системы SunOS и Solaris фирмы Sun Microsystems, HP-UX фирмы Hewlett-Packard, AIX фирмы IBM, IRIX фирмы Silicon Graphics. Вполне естественно, что производитель хочет сделать операционную систему привлекательнее, чем у конкурентов, и не только за счет лучшей производительности, но и за счет расширений и дополнительных возможностей, отсутствующих у других. С другой стороны, производитель желает, чтобы его операционная система оставалась открытой: сегодня закрытые корпоративные решения отпугивают потребителя. Понятно, что в такой ситуации единства и борьбы противоположностей вряд ли найдется система, которую можно назвать "чистой системой UNIX". Да и такое понятие сегодня вряд ли существует. По мнению некоторых разработчиков последней "чистой системой UNIX" являлась Седьмая редакция, сегодня же можно говорить только о наличии в операционной системе черт той или иной ветви — System V, BSD или OSF/1. Можно, например, сказать, что с точки зрения администрирования и набора утилит Digital UNIX представляет смесь System V и BSD UNIX, но с точки зрения интерфейсов и организации системы — это BSD.

Поэтому определение принадлежности конкретной операционной системы к той или иной генеалогической ветви носит весьма условный характер. С этой оговоркой в табл. 1 приведены несколько индикаторов (с точки зрения пользователя и администратора) принадлежности UNIX одной из двух основных ветвей.

**Таблица 1.** К какой генеалогической ветви принадлежит ваша система?

Индикатор	Типично для SVRx	Типично для xBSD
Имя ядра	/unix	/vmunix
Терминальная инициализация	/etc/inittab	/etc/ttys
Файлы инициализации системы	каталоги /etc/rc*.d	файлы /etc/rc.*
Конфигурация монтируемых файловых систем	/etc/mnttab	/etc/mtab
Обычный командный интерпретатор	<i>sh(1)</i> , <i>ksh(1)</i>	<i>csh(1)</i>

Таблица 1 (продолжение)

Индикатор	Типично для SVRx	Типично для <b>xBSD</b>
"Родная" файловая система	S5 (размер блока: 512–2048 байт), имена файлов <=14 символов	UFS (размер блока: 4К–8К), имена файлов < 255 символов
Система печати	<i>lp(1)</i> , <i>lpstat(1)</i> , <i>cancel(1)</i>	<i>lpr(1)</i> , <i>lpq(1)</i> , <i>lpmt(1M)</i> ( <i>lpd</i> daemon)
Управление терминалами	<i>terminfo(4)</i>	<i>termcap(4)</i>
Отображение активности процессов	<i>ps -ef</i>	<i>ps -aux</i>

Ниже приведены краткие характеристики наиболее популярных версий UNIX.

### **AIX**

Версия UNIX фирмы IBM на базе SVR2 со многими чертами SVR4, BSD и OSF/1. Собственная система администрации (SMIT).

### **HP-UX**

Версия UNIX фирмы Hewlett-Packard. В 1996 году компания выпустила новые версии — HP-UX 10.10 и HP-UX 10.20, включающие поддержку симметричных многопроцессорных систем (SMP), файловых систем большого размера (до 128 Гбайт) и расширение виртуального адресного пространства прикладных процессов до 3,75 Гбайт. В середине 1997 года планируется выпустить полностью 64-разрядную версию операционной системы.

### **IRIX**

Версия UNIX фирмы Silicon Graphics, предназначенная для аппаратной платформы этого производителя (MIPS). Ранние версии системы включали много черт BSD UNIX, однако современную систему IRIX (6.x) скорее можно отнести к ветви System V Release 4. Полностью 64-разрядная операционная система.

### **Digital UNIX**

Версия системы OSF/1 фирмы Digital Equipment Corporation (DEC). В прошлом система называлась DEC OSF/1 и по сути являлась BSD UNIX. В то же время в ней есть много черт ветви System V. Полностью 64-разрядная операционная система, разработанная в первую очередь для аппаратной платформы Alpha, содержит все возможности, присущие современным UNIX, — DCE, CDE, современную файловую систему. Поддерживает большинство сетевых интерфейсов, включая Fast Ethernet и ATM.

### **SCO UNIX**

В 1988 году компаний Santa Cruz Operation (SCO), Microsoft и Interactive Systems завершили совместную разработку версии System V Release 3.2 для

платформы Intel 386. В том же году SCO получила от AT&T лицензию на торговую марку и операционная система стала называться SCO UNIX System V/386. В 1995 году компания SCO выпустила версию системы под названием SCO OpenServer Release 5 (кодовое название Everest) — UNIX версии SVR3.2 со многими чертами SVR4. Новая версия системы поддерживает более 900 аппаратных платформ, включая мультипроцессорные вычислительные системы, и более 2000 периферийных устройств.

### ***Solaris***

Версия UNIX SVR4 фирмы Sun Microsystems. Версия 2.5.1 содержит компоненты ядра, использующие 64-разрядную аппаратную архитектуру. Поддерживает распространенные аппаратные платформы, в том числе SPARC, UltraSPARC, Intel 486, Pentium, Pentium Pro и PowerPC. В 1998 году планируется выпустить полностью 64-разрядную версию операционной системы.

## **Причины популярности UNIX**

Почти три десятилетия существования UNIX — очень большой срок для операционной системы. Смело можно сказать, что она полностью выдержала проверку временем. На каждом этапе своего развития операционная система UNIX решала определенные задачи, и сегодня, несмотря на появление более простых и удобных, с точки зрения администрирования, систем, UNIX прочно занимает место среди лидеров. Самое удивительное, что во многих случаях речь при этом идет не о конкретной версии, например Solaris или SCO, а именно о системе UNIX как таковой.

Перечислим основные черты UNIX, позволяющие понять причины долгожительства этой системы:

1. Код системы написан на языке высокого уровня С, что сделало ее простой для понимания, изменений и переноса на другие платформы. По оценкам одного из создателей UNIX, Дэнниса Ритчи, система на языке С имела на 20—40% больший размер, а производительность ее была на 20% ниже аналогичной системы, написанной на ассемблере. Однако ясность и переносимость, а в результате — и открытость системы сыграли решающую роль в ее популярности. Можно смело сказать, что UNIX является одной из наиболее открытых систем. Несмотря на то, что большинство UNIX поставляется сегодня не в исходных текстах, а в виде бинарных файлов, система остается легко расширяемой и настраиваемой.
2. UNIX — многозадачная многопользовательская система с широким спектром услуг. Один мощный сервер может обслуживать запросы большого количества пользователей. При этом необходимо администрирование только одной системы. Ваша система может выполнять раз-

личные функции — работать как вычислительный сервер, обслуживающий сотни пользователей, как сервер базы данных, как сетевой сервер, поддерживающий важнейшие сервисы сети (telnet, ftp, электронную почту, службу имен DNS и т. д.), или даже как сетевой маршрутизатор.

3. Наличие стандартов. Несмотря на многообразие версий UNIX, основой всего семейства являются принципиально одинаковая архитектура и ряд стандартных интерфейсов. Опытный администратор без большого труда сможет обслужить другую версию системы, для пользователей переход на другую версию и вовсе может оказаться незаметным.
4. Простой, но мощный модульный пользовательский интерфейс. Имея в своем распоряжении набор утилит, каждой из которых решает узкую специализированную задачу, вы можете конструировать из них сложные комплексы.
5. Использование единой, легко обслуживаемой иерархической файловой системы. Файловая система — это не только доступ к данным, хранящимся на диске. Через унифицированный интерфейс файловой системы осуществляется доступ к терминалам, принтерам, магнитным лентам, сети и даже к памяти.
6. Очень большое количество приложений, в том числе свободно распространяемых, начиная от простейших текстовых редакторов и заканчивающих мощными системами управления базами данных.

## Общий взгляд на архитектуру UNIX

Самый общий взгляд позволяет увидеть двухуровневую модель системы так, как она представлена на рис. 1.

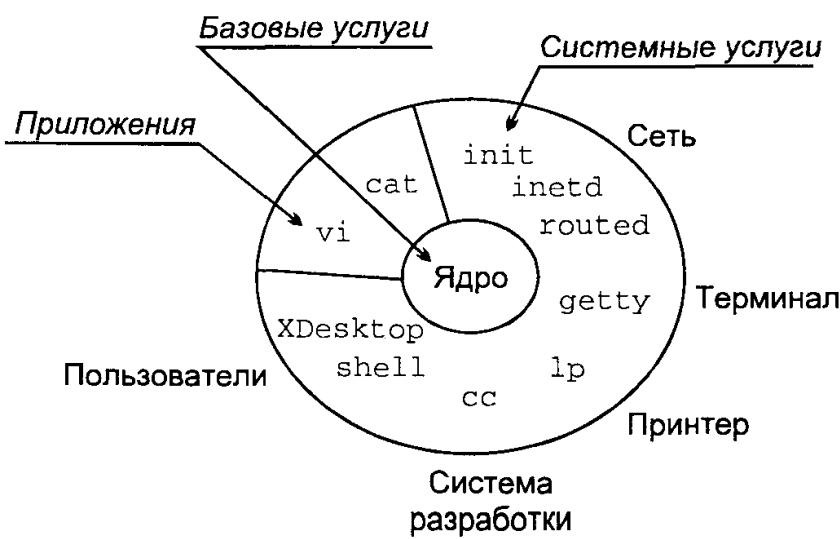


Рис. 1. Модель системы UNIX

В центре находится ядро системы (kernel). Ядро непосредственно взаимодействует с аппаратной частью компьютера, изолируя прикладные программы от особенностей ее архитектуры. Ядро имеет набор услуг, предоставляемых прикладным программам. К услугам ядра относятся операции ввода/вывода (открытия, чтения, записи и управления файлами), создания и управления процессами, их синхронизации и межпроцессного взаимодействия. Все приложения запрашивают услуги ядра посредством *системных вызовов*.

Второй уровень составляют приложения или задачи, как системные, определяющие функциональность системы, так и прикладные, обеспечивающие пользовательский интерфейс UNIX. Однако несмотря на внешнюю разнородность приложений, схемы их взаимодействия с ядром одинаковы.

Рассмотрим более внимательно отдельные компоненты ядра системы.

## Ядро системы

Ядро обеспечивает базовую функциональность операционной системы: создает процессы и управляет ими, распределяет память и обеспечивает доступ к файлам и периферийным устройствам.

Взаимодействие прикладных задач с ядром происходит посредством стандартного интерфейса системных вызовов. *Интерфейс системных вызовов* представляет собой набор услуг ядра и определяет формат запросов на услуги. Процесс запрашивает услугу посредством системного вызова определенной процедуры ядра, внешне похожего на обычный вызов библиотечной функции. Ядро от имени процесса выполняет запрос и возвращает процессу необходимые данные.

В приведенном примере программа открывает файл, считывает из него данные и закрывает этот файл. При этом операции открытия (open), чтения (read) и закрытия (close) файла выполняются ядром по запросу задачи, а функции *open(2)*, *read(2)* и *close(2)* являются системными вызовами.

```
main ()  
{  
    int fd;  
    char buf[80];  
    /*Откроем файл — получим ссылку (файловый дескриптор) fd*/  
    fd = open("file1", 0_RDONLY);  
    /*Считаем в буфер buf 80 символов*/  
    read(fd, buf, sizeof(buf));  
    /*Закроем файл*/  
    close(fd);  
}
```

Структура ядра представлена на рис. 2.

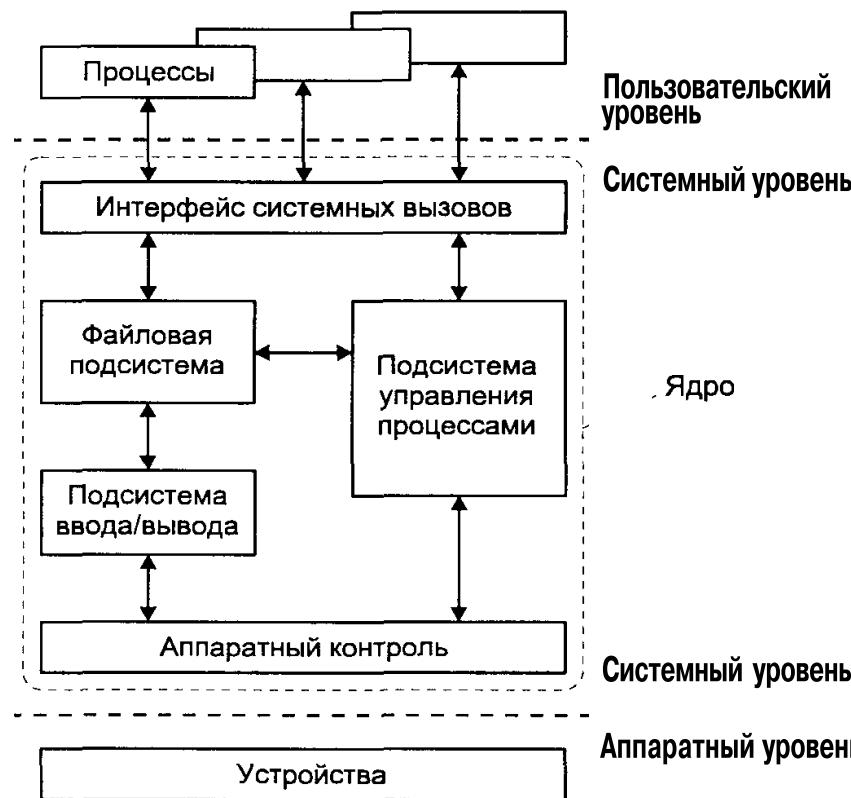


Рис. 2. Внутренняя структура ядра UNIX

Ядро состоит из трех основных подсистем:

1. Файловая подсистема
2. Подсистема управления процессами и памятью
3. Подсистема ввода/вывода

### Файловая подсистема

Файловая подсистема обеспечивает унифицированный интерфейс доступа к данным, расположенным на дисковых накопителях, и к периферийным устройствам. Одни и те же функции *open(2)*, *read(2)*, *write(2)* могут использоваться как при чтении или записи данных на диск, так и при выводе текста на принтер или терминал.

Файловая подсистема контролирует права доступа к файлу, выполняет операции размещения и удаления файла, а также выполняет запись/чтение данных файла. Поскольку большинство прикладных функций выполняется через интерфейс файловой системы (в том числе и доступ к периферийным устройствам), права доступа к файлам определяют привилегии пользователя в системе.

Файловая подсистема обеспечивает перенаправление запросов, адресованных периферийным устройствам, соответствующим модулям подсистемы ввода/вывода.

## Подсистема управления процессами

Запущенная на выполнение программа порождает в системе один или более *процессов* (или *задач*). Подсистема управления процессами контролирует:

- Создание и удаление процессов
- Распределение системных ресурсов (памяти, вычислительных ресурсов) между процессами
- Синхронизацию процессов
- Межпроцессное взаимодействие

Очевидно, что в общем случае число активных процессов превышает число процессоров компьютера, но в каждый конкретный момент времени на каждом процессоре может выполняться только один процесс. Операционная система управляет доступом процессов к вычислительным ресурсам, создавая ощущение одновременного выполнения нескольких задач.

Специальная задача ядра, называемая *распорядителем* или *планировщиком* процессов (scheduler), разрешает конфликты между процессами в конкуренции за системные ресурсы (процессор, память, устройства ввода/вывода). Планировщик запускает процесс на выполнение, следя за тем, чтобы процесс монопольно не захватил разделяемые системные ресурсы. Процесс освобождает процессор, ожидая длительной операции ввода/вывода, или по прошествии кванта времени. В этом случае планировщик выбирает следующий процесс с наивысшим приоритетом и запускает его на выполнение.

*Модуль управления памятью* обеспечивает размещение оперативной памяти для прикладных задач. Оперативная память является дорогостоящим ресурсом, и, как правило, ее редко бывает "слишком много". В случае, если для всех процессов недостаточно памяти, ядро перемещает части процесса или нескольких процессов во вторичную память (как правило, в специальную область жесткого диска), освобождая ресурсы для выполняющегося процесса. Все современные системы реализуют так называемую *виртуальную память*: процесс выполняется в собственном логическом адресном пространстве, которое может значительно превышать доступную физическую память. Управление виртуальной памятью процесса также входит в задачи модуля управления памятью.

*Модуль межпроцессного взаимодействия* отвечает за уведомление процессов о событиях с помощью сигналов и обеспечивает возможность передачи данных между различными процессами.

## Подсистема ввода/вывода

Подсистема ввода/вывода выполняет запросы файловой подсистемы и подсистемы управления процессами для доступа к периферийным устройствам (дискам, магнитным лентам, терминалам и т. д.). Она обеспечивает необходимую буферизацию данных и взаимодействует с *драйверами устройств* — специальными модулями ядра, непосредственно обслуживающими внешние устройства.

## **Работа в операционной системе UNIX**

Сегодня UNIX используется на самых разнообразных аппаратных платформах — от персональных рабочих станций до мощных серверов с тысячами пользователей. И прежде всего потому, что UNIX — это многозадачная многопользовательская система, обладающая широкими возможностями.

С точки зрения пользователя в операционной системе UNIX существуют два типа объектов: *файлы* и *процессы*. Все данные хранятся в виде файлов, доступ к периферийным устройствам осуществляется посредством чтения/записи в специальные файлы. Когда вы запускаете программу, ядро загружает соответствующий исполняемый файл, создает образ процесса и передает ему управление. Более того, во время выполнения процесс может считывать или записывать данные в файл. С другой стороны, вся функциональность операционной системы определяется выполнением соответствующих процессов. Работа системы печати или обеспечения удаленного доступа зависит от того, выполняются ли те или иные процессы в системе<sup>1</sup>.

В этой главе мы познакомимся с пользовательской средой операционной системы UNIX; попробуем взглянуть на UNIX глазами обычного пользователя и администратора системы; не вдаваясь во внутреннюю архитектуру, обсудим, что такое файлы и файловая система, рассмотрим ее организацию и характеристики; с этих же позиций рассмотрим процесс в UNIX, его роль, атрибуты и жизненный цикл.

Мы также постараемся ответить на вопрос, что представляет собой пользователь UNIX как с точки зрения самой системы, так и с точки зрения администрирования; изучим сеанс работы в операционной системе и подробно остановимся на командном интерпретаторе shell — базовой рабочей среде пользователя; познакомимся с наиболее часто используемыми утилитами, неразрывно связанными с UNIX. В заключение постараемся сформулировать основные задачи администрирования этой операционной системы.

<sup>1</sup> Конечно, возможность печати документа или работы в Internet зависят также от наличия принтера или сетевого адаптера, правильности их настройки, работы соответствующих пользовательских и системных приложений, умении пользоваться этими приложениями и многое другое. В следующих главах мы затронем эти аспекты.

## Файлы и файловая система

Файлы в UNIX играют ключевую роль, что не всегда справедливо для других операционных систем. Трудно отрицать значение файлов для пользователей, поскольку все их данные хранятся в виде файлов. Однако помимо этого, файлы в UNIX определяют привилегии пользователей, поскольку права пользователя в большинстве случаев контролируются с помощью прав доступа к файлам. Файлы обеспечивают доступ к периферийным устройствам компьютера, включая диски, накопители на магнитной ленте, CD-ROM, принтеры, терминалы, сетевые адаптеры и даже память. Для приложений UNIX доступ в дисковому файлу "неотличим" от доступа, скажем, к принтеру. Наконец, все программы, которые выполняются в системе, включая прикладные задачи пользователей, системные процессы и даже ядро UNIX, являются исполняемыми файлами.

Как и во многих современных операционных системах, в UNIX файлы организованы в виде древовидной структуры (дерева), называемой *файловой системой* (file system). Каждый файл имеет имя, определяющее его расположение в дереве файловой системы. Корнем этого дерева является *корневой каталог* (root directory), имеющий имя "/". Имена всех остальных файлов содержат *путь* — список каталогов (ветвей), которые необходимо пройти, чтобы достичь файла. В UNIX все доступное пользователям файловое пространство объединено в единое дерево каталогов, корнем которого является каталог "/". Таким образом, полное имя любого файла начинается с "/" и не содержит идентификатора устройства (дискового накопителя, CD-ROM или удаленного компьютера в сети), на котором он фактически хранится.

Однако это не означает, что в системе присутствует только одна файловая система. В большинстве случаев единое дерево, такое каким его видит пользователь системы, составлено из нескольких отдельных файловых систем, которые могут иметь различную внутреннюю структуру, а файлы, принадлежащие этим файловым системам, могут быть расположены на различных устройствах. Вопросы, связанные с объединением нескольких файловых систем в единое дерево, будут обсуждаться при рассмотрении внутреннего устройства файловой системы UNIX в главе 4.

Заметим, что имя файла является атрибутом файловой системы, а не набора некоторых данных на диске, который не имеет имени как такового. Каждый файл имеет связанные с ним *метаданные* (хранищиеся в *индексных дескрипторах* — inode), содержащие все характеристики файла и позволяющие операционной системе выполнять операции, заказанные прикладной задачей: открыть файл, прочитать или записать данные, создать или удалить файл. В частности, метаданные содержат указатели на дисковые блоки хранения данных файла. Имя файла в файловой системе является указателем на его метаданные, в то время как метаданные не содержат указателя на имя файла.

## Типы файлов

В UNIX существуют 6 типов файлов, различающихся по функциональному назначению и действиям операционной системы при выполнении тех или иных операций над файлами:

- Обычный файл (regular file)
- Каталог (directory)
- Специальный файл устройства (special device file)
- FIFO или именованный канал (named pipe)
- Связь (link)
- Сокет

**Обычный файл** представляет собой наиболее общий тип файлов, содержащий данные в некотором формате. Для операционной системы такие файлы представляют собой просто последовательность байтов. Вся интерпретация содержимого файла производится прикладной программой, обрабатывающей файл. К этим файлам относятся текстовые файлы, бинарные данные, исполняемые программы и т. п.

**Каталог.** С помощью каталогов формируется логическое дерево файловой системы. *Каталог* — это файл, содержащий имена находящихся в нем файлов, а также указатели на дополнительную информацию — метаданные, позволяющие операционной системе производить операции над этими файлами. Каталоги определяют положение файла в дереве файловой системы, поскольку сам файл не содержит информации о своем местонахождении. Любая задача, имеющая право на чтение каталога, может прочесть его содержимое, но только ядро имеет право на запись в каталог.

На рис. 1.1 в качестве примера приведена структура каталога. По существу каталог представляет собой таблицу, каждая запись которой соответствует некоторому файлу. Первое поле каждой записи содержит указатель на метаданные (номер mode), а второе определяет имя файла.

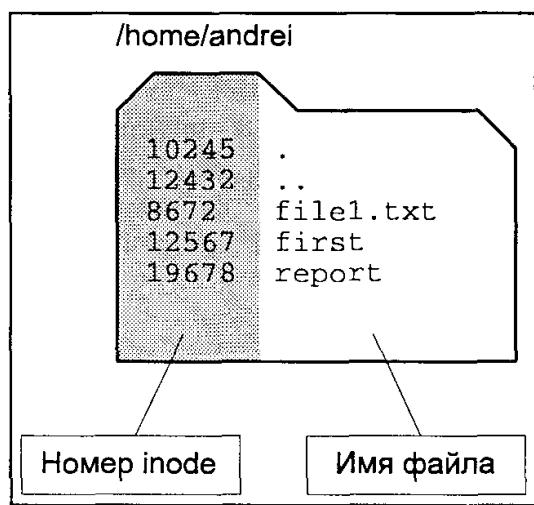


Рис. 1.1. Структура каталога

**Специальный файл устройства** обеспечивает доступ к физическому устройству. В UNIX различают *символьные* (character) и *блочные* (block) файлы устройств. Доступ к устройствам осуществляется путем открытия, чтения и записи в специальный файл устройства.

Символьные файлы устройств используются для небуферизированного обмена данными с устройством, в противоположность этому блочные файлы позволяют производить обмен данными в виде пакетов фиксированной длины — *блоков*. Доступ к некоторым устройствам может осуществляться как через символьные, так и через блочные специальные файлы.

Как производится работа с периферийными устройствами, описано в главе 5.

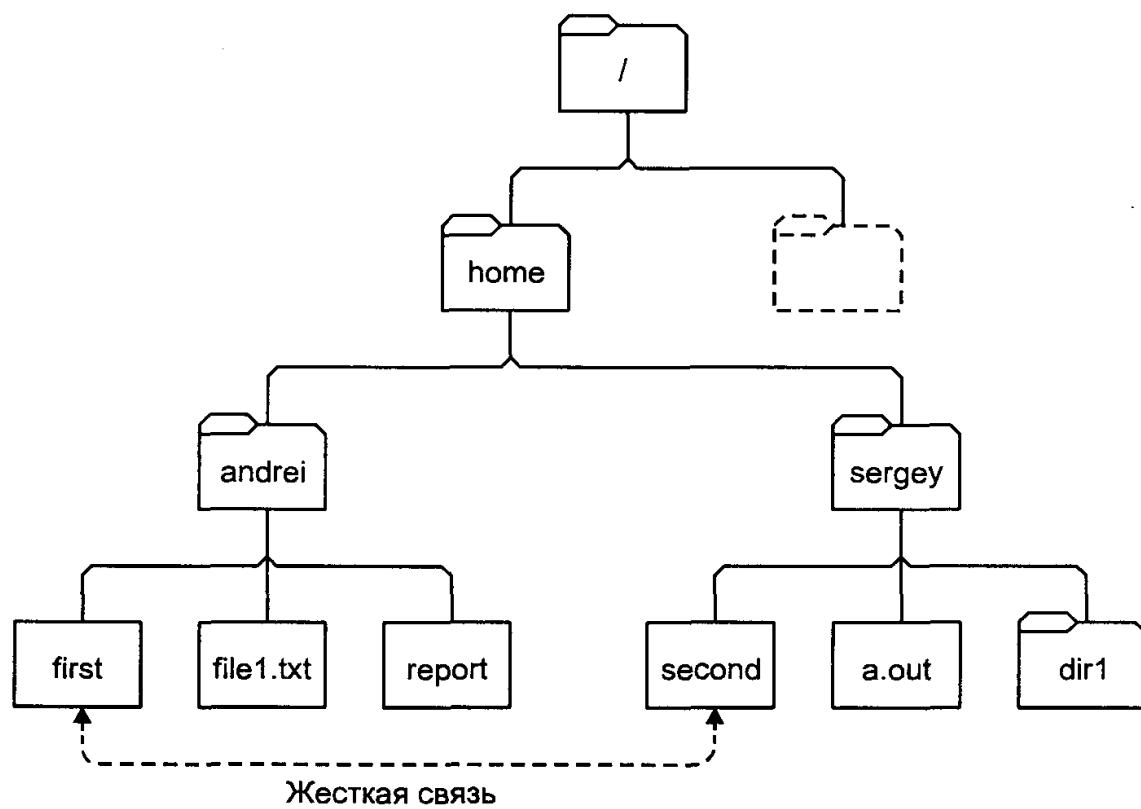
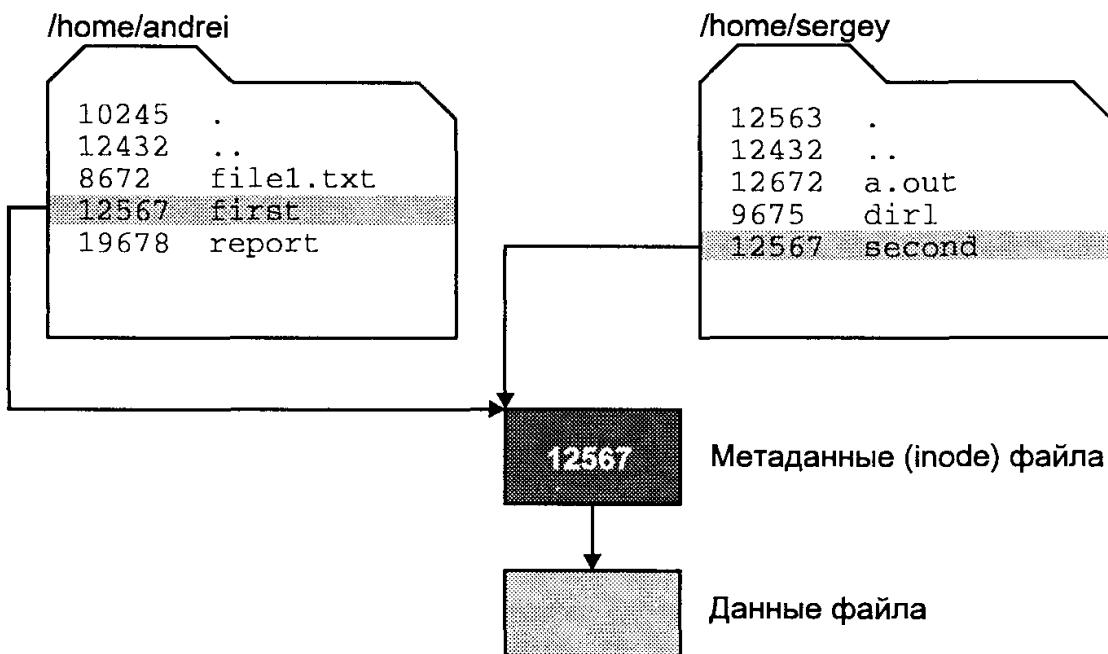
**FIFO или именованный канал** — это файл, используемый для связи между процессами. FIFO впервые появились в System V UNIX, но большинство современных систем поддерживают этот механизм. Более подробно мы рассмотрим этот тип файлов при обсуждении системы межпроцессного взаимодействия в главе 3.

**Связь.** Как уже говорилось, каталог содержит имена файлов и указатели на их метаданные. В то же время сами метаданные не содержат ни имени файла, ни указателя на это имя. Такая архитектура позволяет одному файлу иметь несколько имен в файловой системе. Имена жестко связаны с метаданными и, соответственно, с данными файла, в то время как сам файл существует независимо от того, как его называют в файловой системе<sup>2</sup>. Такая связь имени файла с его данными называется *жесткой связью* (hard link). Например, с помощью команды *ln(1)* мы можем создать еще одно имя (**second**) файла, на который указывает имя **first** (рис. 1.2).

```
$ pwd
/home/andrei
$ ln first /home/sergey/second
```

Жесткие связи абсолютно равноправны. В списках файлов каталогов, которые можно получить с помощью команды *ls(1)*, файлы **first** и **second** будут отличаться только именем. Все остальные атрибуты файла будут абсолютно одинаковыми. С точки зрения пользователя — это два разных файла. Изменения, внесенные в любой из этих файлов, затронут и другой, поскольку оба они ссылаются на одни и те же данные файла. Вы можете переместить один из файлов в другой каталог — все равно эти имена будут связаны жесткой связью с данными файла. Легко проверить, что удаление одного из файлов (**first** или **second**) не приведет к удалению самого файла, т. е. его метаданных и данных (если это не специальный файл устройства).

<sup>2</sup> Данное утверждение верно лишь отчасти. Действительно, файлу "безразлично", какие имена он имеет в каталогах, но "небезразлично" число этих имен. Если ни одно из имен файловой системы не ссылается на файл — он должен быть удален (т. е. физически удалены его данные на диске).



**Рис. 1.2.** Структура файловой системы после выполнения команды `ln(1)`. Жесткая связь имен с данными файла

По определению жесткие связи указывают на один и тот же индексный дескриптор inode. Поэтому проверить, имеют ли два имени файла жесткую связь, можно, вызвав команду `ls(l)` с ключом `-i`:

```
$ ls -i /home/andrei/first /home/sergey/second
12567      first
12567      second
```

Информацию о наличии у файла нескольких имен, связанных с ним жесткими связями, можно получить, просмотрев подробный листинг файлов с помощью команды `ls - /`:

```
$ ls -1 /home/sergey
```

```
-rw-r--r--  2 andrei  staff  7245 Jan 17 8:05 second
```

Во второй колонке листинга указано число жестких связей данного файла.

Сразу оговоримся, что жесткая связь является естественной формой связи имени файла с его метаданными и не принадлежит к особому типу файла. Особым типом файла является *символическая связь*, позволяющая косвенно адресовать файл. В отличие от жесткой связи, символическая связь адресует файл, который, в свою очередь, ссылается на другой файл. В результате, последний файл адресуется символической связью косвенно (рис. 1.3). Данные файла, являющегося символической связью, содержат только имя целевого файла.

Проиллюстрируем эти рассуждения на примере. Команда `ln(1)` с ключом `-s` позволяет создать символическую связь:

```
$ pwd
/home/andrei
$ ln -s first /home/sergey/symfirst
$ cd /home/sergey
$ ls -1
```

```
lrwxrwxrwx 1 andrei  staff  15 Jan 17 8:05  symfirst->../andrei/first
```

Как видно из вывода команды `ls(1)`, файл `symfirst` (символическая связь) существенно отличается от файла `second` (жесткая связь). Во-первых, фактическое содержимое файла `symfirst` отнюдь не то же, что и у файла `first` или `second`, об этом говорит размер файла — 15 байт. На самом деле в этом файле хранится не что иное как имя файла, на которую символическая связь ссылается — `../andrei/first` — ровно 15 байт. Во-вторых, файл `symfirst` не содержит никаких ограничений на доступ (2—10 символы в первой колонке).

Символическая связь является особым типом файла (об этом свидетельствует символ 'l' в первой позиции вывода `ls(1)`), и операционная система работает с таким файлом не так, как с обычным. Например, при выводе на экран содержимого файла `symfirst` появятся данные файла `/home/andrei/first`.

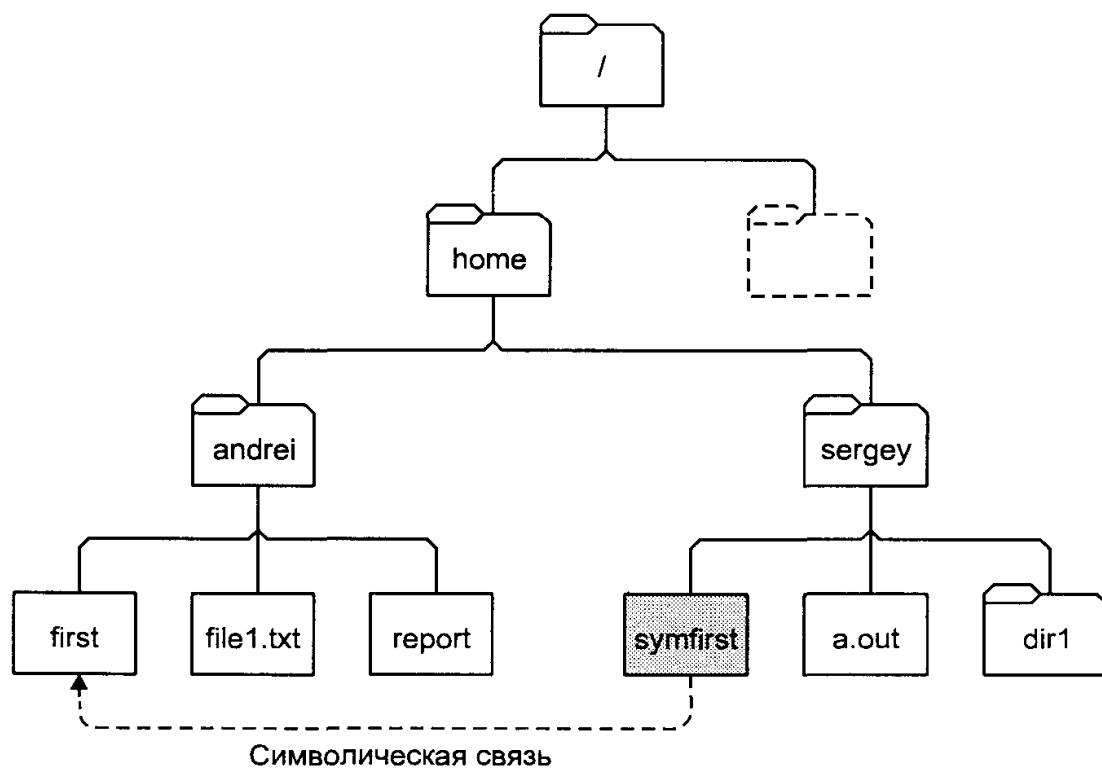
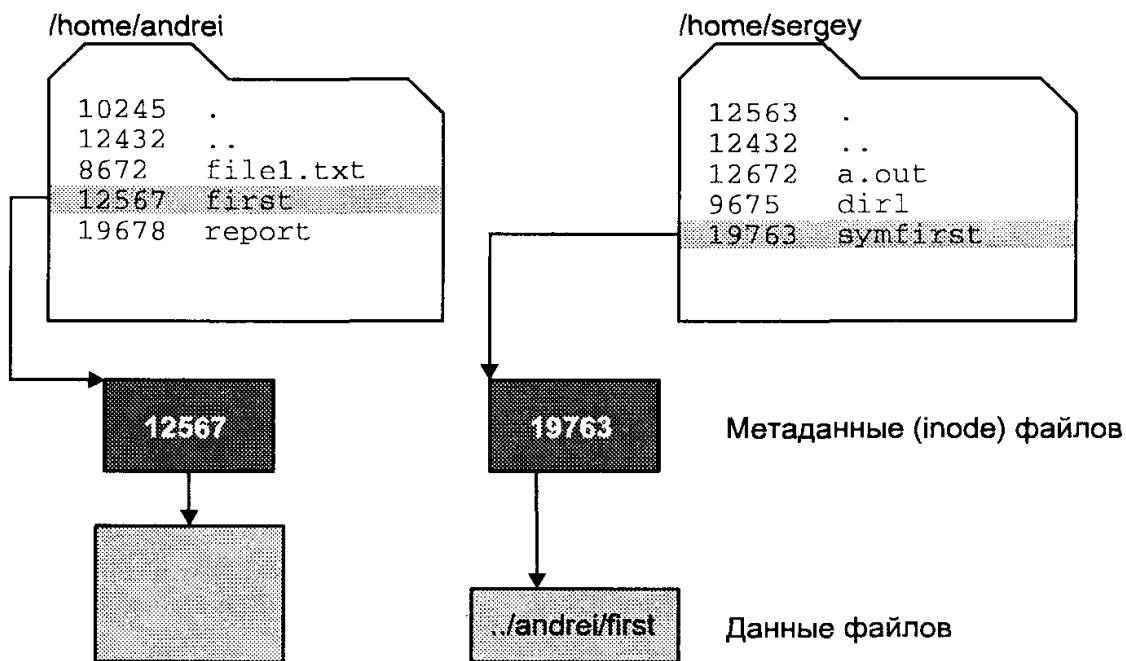


Рис. 1.3. Символическая связь

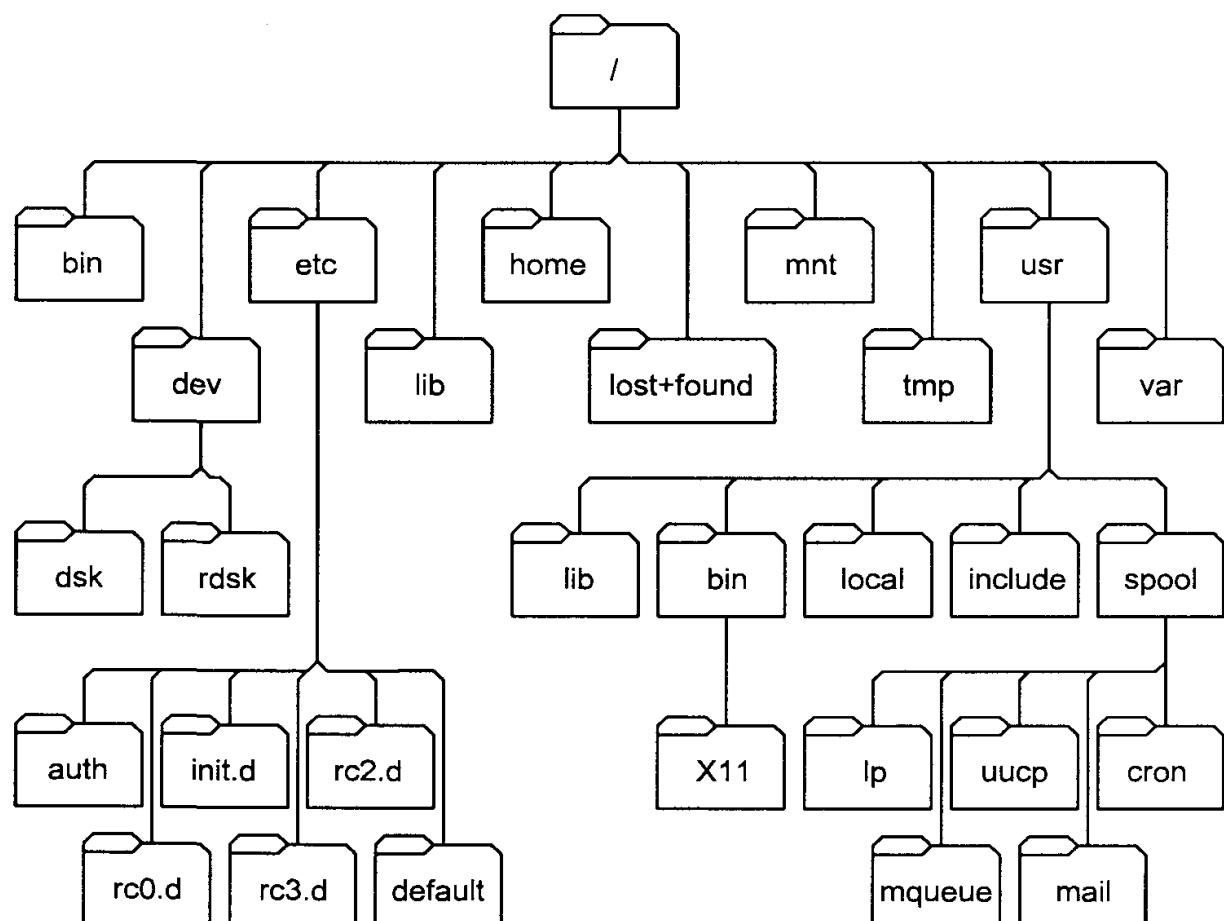
## Сокеты

Сокеты предназначены для взаимодействия между процессами. Интерфейс сокетов часто используется для доступа к сети TCP/IP. В системах, ветви

BSD UNIX на базе сокетов реализована система межпроцессного взаимодействия, с помощью которой работают многие системные сервисы, например, система печати. Мы подробнее познакомимся с сокетами в разделе "Межпроцессное взаимодействие в BSD UNIX" главы 3.

## Структура файловой системы UNIX

Использование общепринятых имен основных файлов и структуры каталогов существенно облегчает работу в операционной системе, ее администрирование и переносимость. Эта структура используется в работе системы, например при ее инициализации и конфигурировании, при работе почтовой системы и системы печати. Нарушение этой структуры может привести к неработоспособности системы или отдельных ее компонентов.



**Рис. 1.4.** Типичная файловая система UNIX

Приведем краткое описание основных каталогов.

### **Корневой каталог**

Корневой каталог "/" является основой любой файловой системы UNIX. Все остальные файлы и каталоги располагаются в рамках структуры, по-

рожденной корневым каталогом, независимо от их физического местонахождения.

### **/bin**

В каталоге **/bin** находятся наиболее часто употребляемые команды и утилиты системы, как правило, общего пользования.

### **/dev**

Каталог **/dev** содержит специальные файлы устройств, являющиеся интерфейсом доступа к периферийным устройствам.

Каталог **/dev** может содержать несколько подкаталогов, группирующих специальные файлы устройств одного типа. Например, каталог **/dev/dsk** содержит специальные файлы устройств для доступа к гибким и жестким дискам системы.

### **/etc**

В этом каталоге находятся системные конфигурационные файлы и многие утилиты администрирования. Среди наиболее важных файлов — скрипты инициализации системы. Эти скрипты хранятся в каталогах **/etc/rc0.d**, **/etc/rc1.d**, **/etc/rc2.d** и т. д., соответствующих уровням выполнения системы (run level), и управляются скриптами **/etc/rc0**, **/etc/rc1**, **/etc/rc2** и т. д. Во многих версиях BSD UNIX указанные каталоги отсутствуют, и загрузка системы управляется скриптами **/etc/rc.boot**, **/etc/re** и **/etc/rc.local**. В UNIX ветви System V здесь находится подкаталог **default**, где хранятся параметры по умолчанию многих команд (например, **/etc/default/su** содержит параметры для команды *su(1M)*). В UNIX System V большинство исполняемых файлов перемещены в каталог **/sbin** или **/usr/sbin**.

### **/lib**

В каталоге **/lib** находятся библиотечные файлы языка С и других языков программирования. Стандартные названия библиотечных файлов имеют вид **libx.a** (или **libx.so**), где x — это один или более символов, определяющих содержимое библиотеки. Например, стандартная библиотека С называется **libc.a**, библиотека системы X Window System имеет имя **libXll.a**. Часть библиотечных файлов также находится в каталоге **/usr/lib**.

### **/lost+found**

Каталог "потерянных" файлов. Ошибки целостности файловой системы, возникающие при неправильном останове UNIX или аппаратных сбоях, могут привести к появлению т. н. "безымянных" файлов — структура и содержимое файла являются правильными, однако для него отсутствует имя в каком-либо из каталогов. Программы проверки и восстановления файловой системы помешают такие файлы в каталог **/lost+found** под систем-

ными числовыми именами. Мы коснемся вопроса имен файлов далее в этой главе и, более подробно, в главе 4.

#### **/mnt**

Стандартный каталог для временного связывания (монтирования) физических файловых систем к корневой для получения единого дерева логической файловой системы. Обычно содержимое каталога **/mnt** пусто, поскольку при монтировании он перекрывается связанной файловой системой. Более подробно процесс монтирования и относящиеся к нему структуры данных ядра мы рассмотрим в главе 4.

#### **/и или /home**

Общеупотребительный каталог для размещения домашних каталогов пользователей. Например, имя домашнего каталога пользователя **andrei** будет, скорее всего, называться **/home/andrei** или **/u/andrei**. В более ранних версиях UNIX домашние каталоги пользователей размещались в каталоге **/usr**.

#### **/usr**

В этом каталоге находятся подкаталоги различных сервисных подсистем — системы печати, электронной почты и т. д. (**/usr/spool**), исполняемые файлы утилит UNIX (**/usr/bin**), дополнительные программы, используемые на данном компьютере (**/usr/local**), файлы заголовков (**/usr/include**), электронные справочники (**/usr/man**) и т. д.

#### **/var**

В UNIX System V этот каталог является заменителем каталога **/usr/spool**, используемого для хранения временных файлов различных сервисных подсистем — системы печати, электронной почты и т. д.

#### **/tmp**

Каталог хранения временных файлов, необходимых для работы различных подсистем UNIX. Обычно этот каталог открыт на запись для всех пользователей системы.

## **Владельцы файлов**

Файлы в UNIX имеют двух владельцев: пользователя (user owner) и группу<sup>3</sup> (group owner). Важной особенностью является то, что владелец-пользователь может не являться членом группы, владеющей файлом. Это

<sup>3</sup> Группой называется определенный список пользователей системы. Пользователь системы может быть членом нескольких групп, одна из которых является *первичной* (primary), остальные — *дополнительными* (supplementary).

дает большую гибкость в организации доступа к файлам. Совместное пользование файлами можно организовать практически для любого состава пользователей, создав соответствующую группу и установив для нее права на требуемые файлы. При этом для того чтобы некий пользователь получил доступ к этим файлам, достаточно включить его в группу-владельца, и наоборот — исключение из группы автоматически изменяет для пользователя права доступа к файлам.

Для определения владельцев файла достаточно посмотреть подробный листинг команды *ls -l*. Третья и четвертая колонки содержат имена владельца-пользователя и владельца-группы, соответственно:

1	2	3	4	5	6	7	8
-rw-r--r--	1	andy	group	235520	Dec 22	19:13	pride.tar
-rw-rw-r--	1	andy	student	3450	Nov 12	19:13	exams.quest

Владельцем-пользователем вновь созданного файла является пользователь, который создал файл. Порядок назначения владельца-группы зависит от конкретной версии UNIX. Например, в SCO UNIX владельцем-группой является первичная группа пользователя, создавшего файл, а в Digital UNIX владелец-группа наследуется от владельца группы — каталога, в котором создается файл<sup>4</sup>.

Для изменения владельца файла используется команда *chown(1)*. В качестве параметров команда принимает имя владельца-пользователя и список файлов, для которых требуется изменить данный атрибут. Например, следующая команда установит пользователя sergey владельцем файлов **client.c** и **server.c**:

```
$ chown sergey client.c server.c
```

Изменение владельца-группы производится командой *chgrp(1)*. Как и *chown(1)*, в качестве параметров команда принимает имя владельца-группы и список файлов, для которых требуется изменить данный атрибут. Например, для установки группы staff в качестве владельца всех файлов текущего каталога, необходимо задать следующую команду:

```
$ chgrp staff *
```

На самом деле файл создает не пользователь, а процесс, запущенный пользователем. Процесс имеет атрибуты, связанные с пользователем и группой, которые и назначаются файлу при его создании. Более точное описание передачи "владения" имеет вид:

1. Идентификатор владельца-пользователя файла (UID) устанавливается равным EUID процесса, создающего файл (т. е. вызвавшего функцию *open(2)* или *creat(2)*).
2. Идентификатор владельца-группы файла (group ID) устанавливается равным
  - a) EGID процесса (для версии System V);
  - б) GID каталога, в котором файл создается (для версии BSD).

Большинство систем, использующих наследование System V, позволяют также устанавливать наследование группового владельца в стиле BSD. Это достигается установкой флага SGID на каталог. Более подробно об этом см. раздел "Дополнительные атрибуты" далее в этой главе.

Владение файлом определяет тот набор операций, который пользователь может совершить с файлом. Часть из них, такие как изменение прав доступа или владельца файла (табл. 1.1), может осуществлять только владелец (или суперпользователь), другие операции, такие как чтение, запись и запуск на выполнение (для исполняемых файлов) дополнительно контролируются правами доступа.

**Таблица 1.1.** Операции изменения владельцев файла

Операция	Команда	Имеет право выполнять	
		в системе BSD 4.x	в системе SVR4
Изменение владельца-пользователя	<i>chown(1)</i>	суперпользователь	владелец файла
Изменение владельца-группы	<i>chgrp(1)</i>	суперпользователь	владелец файла только для группы, к которой сам принадлежит (в соответствии с POSIX)

### Права доступа к файлу

В операционной системе UNIX существуют три базовых класса доступа к файлу, в каждом из которых установлены соответствующие права доступа:

- |                  |   |
|------------------|---|
| User access (u)  | Для владельца-пользователя файла                      |
| Group access (g) | Для членов группы, являющейся владельцем файла        |
| Other access (o) | Для остальных пользователей (кроме суперпользователя) |

UNIX поддерживает три типа прав доступа для каждого класса: на чтение (read, обозначается символом r), на запись (write, обозначается символом w) и на выполнение (execute, обозначается символом x).

С помощью команды *ls -l* можно получить список прав доступа к файлу:

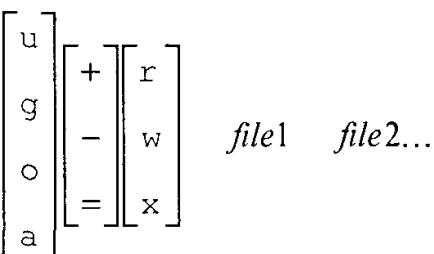
-rw-r--r--	1	andy	group	36482	Dec 22	19:13	report.txt.1
drwxr-xr--	2	andy	group	64	Aug 15	11:03	temp
-rwxr-xr--	1	andy	group	4889	Dec 22	15:13	a.out
-rw-r--r--	1	andy	group	7622	Feb 11	09:13	cont.c

Права доступа листинга отображаются в первой колонке (за исключением первого символа, обозначающего тип файла). Наличие права обозначается соответствующим символом, а отсутствие — символом '-'. Рассмотрим, например, права доступа к файлу **a.out**:

Тип файла	Права владельца-пользователя	Права владельца-группы	Права остальных пользователей
-----------	------------------------------	------------------------	-------------------------------

Обычный файл Чтение, запись, выполнение Чтение и выполнение Только чтение

Права доступа могут быть изменены только владельцем файла или суперпользователем (superuser) — администратором системы. Для этого используется команда *chmod(1)*. Ниже приведен общий формат этой команды.

chmod 

В качестве аргументов команда принимает указание классов доступа ('u' — владелец-пользователь, 'g' — владелец-группа, 'o' — остальные пользователи, 'a' — все классы пользователей), права доступа ('r' — чтение, 'w' — запись и 'x' — выполнение) и операцию, которую необходимо произвести ('+' — добавить, '−' — удалить и '=' — присвоить) для списка файлов *file1*, *file2* и т. д. Например, команда

\$ chmod g-wx **ownfile**

лишит членов группы-владельца файла **ownfile** права на запись и выполнение этого файла.

В одной команде можно задавать различные права для нескольких классов доступа, разделив их запятыми.

Приведем еще несколько примеров:

\$ chmod a+w text	Предоставить право на запись для всех пользователей
\$ chmod go=r text	Установить право на чтение для всех пользователей, за исключением владельца
\$ chmod g+x-w runme	Добавить для группы право на выполнение файла <i>runme</i> и снять право на запись
\$ chmod u+w,og+r-w text1 text2	Добавить право записи для владельца, право на чтение для группы и остальных пользователей, отключить право на запись для всех пользователей, исключая владельца

Последний пример демонстрирует достаточно сложную установку прав доступа. Вы можете установить сразу все девять прав доступа, используя числовую форму команды *chmod(1)*:

```
$ chmod 754 *
```

Число определяется следующим образом: нужно представить права доступа в двоичном виде (0 — отсутствие соответствующего права, 1 — его наличие) и каждую триаду, соответствующую классу доступа, в свою очередь преобразовать в десятичное число.

Владелец	Группа	Остальные
r w x	r — x	r — -
1 1 1	1 0 1	1 0 0
7	5	4

Таким образом, приведенный пример эквивалентен следующей символьной форме *chmod(1)*:

```
$ chmod u=rwx, g=rx, o=r *
```

Значение прав доступа различно для разных типов файлов. Для файлов операции, которые можно производить, следуют из самих названий прав доступа. Например, чтобы просмотреть содержимое файла командой *cat(l)*, пользователь должен иметь право на чтение (r). Редактирование файла, т. е. его изменение, предусматривает наличие права на запись (w). Наконец, для того чтобы запустить некоторую программу на выполнение, вы должны иметь соответствующее право (x). Исполняемый файл может быть как скомпилированной программой, так и скриптом командного интерпретатора *shell*. В последнем случае вам также понадобится право на чтение, поскольку при выполнении скрипта командный интерпретатор должен иметь возможность считывать команды из файла. Все сказанное, за исключением, пожалуй, права на выполнение, имеющего смысл лишь для обычных файлов и каталогов, справедливо и для других типов файлов: специальных файлов устройств, именованных каналов, и сокетов. Например, чтобы иметь возможность распечатать документ, вы должны иметь право на запись в специальный файл устройства, связанный с принтером<sup>5</sup>. Для каталогов эти права имеют другой смысл, а для символических связей они вообще не используются, поскольку контролируются целевым файлом.

Права доступа для каталогов не столь очевидны. Это в первую очередь связано с тем, что система трактует операции чтения и записи для каталогов отлично от остальных файлов. Право чтения каталога позволяет вам

<sup>5</sup> На самом деле специальный файл, связанный с устройством печати, не имеет общедоступных прав на запись. Доступ к принтеру контролируется системой печати, из которой и происходит доступ к этому файлу.

получить имена (и только имена) файлов, находящихся в данном каталоге. Чтобы получить дополнительную информацию о файлах каталога (например, подробный листинг команды `ls -l`), системе придется "заглянуть" в метаданные файлов, что требует права на выполнения для каталога. Право на выполнения также потребуется для каталога, в который вы захотите перейти (т. е. сделать его текущим) с помощью команды `cd ()`. Это же право нужно иметь для доступа ко всем каталогам на пути к указанному. Например, если вы установите право на выполнения для всех пользователей в одном из своих подкаталогов, он все равно останется недоступным, пока ваш домашний каталог не будет иметь такого же права.

Права `g` и `x` действуют независимо, право `x` для каталога не требует наличия права `g`, и наоборот. Комбинацией этих двух прав можно добиться интересных эффектов, например, создания "темных" каталогов, файлы которых доступны только в случае, если пользователь заранее знает их имена, поскольку получение списка файлов таких каталогов запрещено. Данный прием, кстати, используется при создании общедоступных архивов в сети (FTP-серверов), когда некоторые разделы архива могут использоваться только "посвященными", знающими о наличии того или иного файла в каталоге. Приведем пример создания "темного" каталога.

```
$ pwd                               Где мы находимся?  
/home/andrei  
$ mkdir darkroom                     Создадим каталог  
$ ls -l                             Получим его атрибуты  
  
-rwxr--r--  2 andy  group 65 Dec 22 19:13 darkroom  
$ chmod a-r+x darkroom               Превратим его в "темный" каталог  
$ ls -l                             Получим его атрибуты  
  
--wx--x--x  2 andy  group 65 Dec 22 19:13 darkroom  
$ cp file1 darkroom                 Поместим в каталог darkroom некоторый файл  
$ cd darkroom                       Перейдем в этот каталог  
$ ls -l darkroom                    Попытаемся получить листинг каталога  
##permission denied                Увы...  
$ cat file1                         Тем не менее, заранее зная имя файла (file1), можно  
ok                                    работать с ним (например, прочитать, если есть соответствующее право доступа)
```

Особого внимания требует право на запись для каталога. Создание и удаление файлов в каталоге требуют изменения его содержимого, и, следовательно, права на запись в этот каталог. Самое важное, что при этом не учитываются права доступа для самого файла. То есть для того, чтобы удалить некоторый файл из каталога, не обязательно иметь какие-либо права доступа к этому файлу, важно лишь иметь право на запись для каталога, в котором находится этот файл. Имейте в виду, что право на запись в каталог дает

большие полномочия, и предоставляемые это право с осторожностью. Правда, существует способ несколько обезопасить себя в случае, когда необходимо предоставить право на запись другим пользователям, — установка флага Sticky bit на каталог. Но об этом мы поговорим чуть позже.

В табл. 1.2 приведены примеры некоторых действий над файлами и минимальные права доступа, необходимые для выполнения этих операций.

**Таблица 1.2.** Примеры прав доступа

Команда	Смысл действия	Минимальные права доступа	
		для обычного файла	для каталога, содержащего файл
cd /u/andrei	Перейти в каталог /u/andrei	—	x
ls /u/andrei/* .c	Вывести все файлы с суффиксом .c этого каталога	—	r
ls -s /u/andrei/* .c	Вывести дополнительную информацию об этих файлах (размер)		rx
cat report.txt	Вывести на экран содержимое файла report.txt	r	x
cat >> report.txt	Добавить данные в файл report.txt	w	x
runme.sh	Выполнить программу runme	x	x
runme	Выполнить скрипт командного интерпретатора runme.sh	rx	x
rm runme	Удалить файл runme в текущем каталоге	—	xw

Итак, для выполнения операции над файлом имеют значение класс доступа, к которому вы принадлежите, и права доступа, установленные для этого класса. Поскольку для каждого класса устанавливаются отдельные права доступа, всего определено 9 прав доступа, по 3 на каждый класс.

Операционная система производит проверку прав доступа при создании, открытии (для чтения или записи), запуске на выполнение или удалении файла. При этом выполняются следующие проверки:

1. Если операция запрашивается суперпользователем, доступ разрешается. Никакие дополнительные проверки не производятся. Это позволяет

ет администратору иметь неограниченный доступ ко всей файловой системе.

2. Если операция запрашивается владельцем файла, то:
  - а) если требуемое право доступа определено (например, при операции чтения файла установлено право на чтение для владельца-пользователя данного файла), доступ разрешается,
  - б) в противном случае доступ запрещается.
3. Если операция запрашивается пользователем, являющимся членом группы, которая является владельцем файла, то:
  - а) если требуемое право доступа определено, доступ разрешается,
  - б) в противном случае доступ запрещается.
4. Если требуемое право доступа для прочих пользователей (other) установлено, доступ разрешается, в противном случае доступ запрещается.

Система проводит проверки в указанной последовательности. Например, если пользователь является владельцем файла, то доступ определяется исключительно из прав владельца-пользователя, права владельца-группы не проверяются, даже если пользователь является членом владельца-группы. Чтобы проиллюстрировать это, рассмотрим следующее:

```
—rw-r-- 2 andy group 65 Dec 22 19:13 file1
```

Даже если пользователь `andy` является членом группы `group`, он не сможет ни прочитать, ни изменить содержимое файла `file1`. В то же время все остальные члены этой группы имеют такую возможность. В данном случае, владелец файла обладает наименьшими правами доступа к нему. Разумеется, рассмотренная ситуация носит гипотетический характер, поскольку пользователь `andy` в любой момент может изменить права доступа к данному файлу как для себя (владельца), так и для группы, и всех остальных пользователей в системе.

## Дополнительные атрибуты файла

Мы рассмотрели основные атрибуты, управляющие доступом к файлу. Существует еще несколько атрибутов, изменяющих стандартное выполнение различных операций. Как и в случае прав доступа, эти атрибуты по-разному интерпретируются для каталогов и других типов файлов.

Дополнительные атрибуты также устанавливаются утилитой `chmod(l)`, но вместо кодов '`r`', '`w`' или '`Y`' используются коды из табл. 1.3. Например, для установки атрибута SGID для файла `file1` необходимо выполнить команду

```
$ chmod g+s file1
```

В табл. 1.3 приведены дополнительные атрибуты для файлов, и показано, как они интерпретируются операционной системой.

Таблица 1.3. Дополнительные атрибуты для обычных файлов

Код	Название	Значение
t	Sticky bit	Сохранить образ выполняемого файла в памяти после завершения выполнения
S	Set UID, SUID	Установить UID процесса при выполнении
S	Set GID, SGID	Установить GID процесса при выполнении
1	Блокирование	Установить обязательное блокирование файла

Установка атрибута Sticky bit (действительное название — save text mode) редко используется в современных версиях UNIX для файлов. В ранних версиях этот атрибут применялся с целью уменьшить время загрузки наиболее часто запускаемых программ (например, редактора или командного интерпретатора). После завершения выполнения задачи ее образ (т. е. код и данные) оставались в памяти, поэтому последующие запуски этой программы занимали значительно меньше времени.

Атрибуты (или флаги) SUID и SGID позволяют изменить права пользователя при запуске на выполнение файла, имеющего эти атрибуты. При этом привилегии будут изменены (обычно расширены) лишь на время выполнения и только в отношении этой программы<sup>6</sup>.

Обычно запускаемая программа получает права доступа к системным ресурсам на основе прав доступа пользователя, запустившего программу. Установка флагов SUID и SGID изменяет это правило, назначая права доступа исходя из прав доступа владельца файла. Таким образом, запущенный исполняемый файл, которым владеет суперпользователь, получает неограниченные права доступа к системным ресурсам, независимо от того, кто его запустил. При этом установка SUID приведет к наследованию прав владельца-пользователя файла, а установка SGID — владельца-группы.

В качестве примера использования этого свойства рассмотрим утилиту *passwd(1)*, позволяющую пользователю изменить свой пароль. Очевидно, что изменение пароля должно привести к изменению содержимого определенных системных файлов (файла пароля */etc/passwd* или */etc/shadow*, или базы данных пользователей, если используется дополнительная защищена системы). Понятно, что предоставление права на запись в эти файлы всем пользователям системы является отнюдь не лучшим решением. Установка SUID для программы *passwd(1)* (точнее, на файл **/usr/bin/passwd** — исполняемый файл утилиты *passwd(1)*) позволяет изящно разрешить это противоречие. Поскольку владельцем файла **/usr/bin/passwd** является су-

<sup>6</sup> Следует оговориться, что если программа в процессе выполнения запускает другие задачи, то они будут наследовать ее права доступа. Поэтому устанавливать флаги SUID и SGID следует с большой осторожностью и только для программ, которые не имеют возможности запуска произвольных задач.

перпользователь (его имя в системе — root), то кто бы ни запустил утилиту *passwd(1)* на выполнение, во время работы данной программы он временно получает права суперпользователя, т. е. может производить запись в системные файлы, защищенные от остальных пользователей.

```
$ ls -lFa /usr/bin/passwd
-r-sr-sr-x 3 root    sys 15688 Oct 25 1995 /usr/bin/passwd*
```

Понятно, что требования по безопасности для такой программы должны быть повышенны. Утилита *passwd(1)* должна производить изменение пароля только пользователя, запустившего ее, и не позволять никакие другие операции (например, вызов других программ).

Блокирование файлов позволяет устраниить возможность конфликта, когда две или более задачи одновременно работают с одним и тем же файлом. К этому вопросу мы вернемся в главе 4.

Однако вернемся к обсуждению дополнительных атрибутов для каталогов (табл. 1.4).

**Таблица 1.4.** Дополнительные атрибуты для каталогов

Код	Название	Значение
t	Sticky bit	Позволяет пользователю удалять только файлы, которыми он владеет или имеет права на запись
s	Set GID, SGID	Позволяет изменить правило установки владельца-группы создаваемых файлов, аналогично реализованному в BSD UNIX

При обсуждении прав доступа отмечалось, что предоставление права на запись в каталог дает достаточно большие полномочия. Имея такое право, пользователь может удалить из каталога любой файл, даже тот, владельцем которого он не является и в отношении которого не имеет никаких прав. Установка атрибута Sticky bit для каталога позволяет установить дополнительную защиту файлов, находящихся в каталоге. Из такого каталога пользователь может удалить только файлы, которыми он владеет, или на которые он имеет явное право доступа на запись, даже при наличии права на запись в каталог. Примером может служить каталог */tmp*, который является открытым на запись для всех пользователей, но в котором может оказаться нежелательной возможность удаления пользователем чужих временных файлов.

Атрибут SGID также имеет иное значение для каталогов. При установке этого атрибута для каталога вновь созданные файлы этого каталога будут наследовать владельца-группу по владельцу-группе каталога. Таким образом для UNIX версии System V удается имитировать поведение систем версии BSD, для которых такое правило наследования действует по умолчанию.

Посмотреть наличие дополнительных атрибутов можно с помощью подробного списка файлов:

```
$ ls -l
```

```
drwxrwxrwt 5 sys sys 367 Dec 19 20:29 /tmp
-r-sr-sr-x 3 root sys 15688 Oct 25 1995 /usr/bin/passwd*
```

**Таблица 1.5.** Операции изменения атрибутов файла

Операция	Команда/системный вызов	Кому разрешено
Изменение прав доступа	<i>chmod(1)</i>	владелец
Изменение дополнительного атрибута Sticky bit	<i>chmod(1)</i>	суперпользователь
Изменение дополнительного атрибута SGID	<i>chmod(1)</i>	владелец, причем его GID также должен совпадать с идентификатором группы файла

## Процессы

Процессы в операционной системе UNIX играют ключевую роль. От оптимальной настройки подсистемы управления процессами и числа одновременно выполняющихся процессов зависит загрузка ресурсов процессора, что в свою очередь имеет непосредственное влияние на производительность системы в целом. Ядро операционной системы предоставляет задачам базовый набор услуг, определяемых интерфейсом системных вызовов. К ним относятся основные операции по работе с файлами, управление процессами и памятью, поддержка межпроцессного взаимодействия. Дополнительные функциональные возможности системы, т. е. услуги, которые она предоставляет пользователям, определяются активными процессами. От того, какие процессы выполняются в вашей системе, зависит, является ли она сервером базы данных или сервером сетевого доступа, средством проектирования или вычислительным сервером. Даже так называемые уровни выполнения системы (run levels), которые мы рассмотрим позже, представляют собой удобный способ определения группы выполняющихся процессов и, соответственно, функциональности системы.

## Программы и процессы

Обычно *программой* называют совокупность файлов, будь то набор исходных текстов, объектных файлов или собственно выполняемый файл. Для того чтобы программа могла быть запущена на выполнение, операционная

система сначала должна создать *окружение* или *среду выполнения* задачи, куда относятся ресурсы памяти, возможность доступа к устройствам ввода/вывода и различным системным ресурсам, включая услуги ядра.

Это окружение (среда выполнения задачи) получило название *процесса*. Мы можем представить процесс как совокупность данных ядра системы, необходимых для описания образа программы в памяти и управления ее выполнением. Мы можем также представить процесс как программу в стадии ее выполнения, поскольку все выполняющиеся программы представлены в UNIX в виде процессов. Процесс состоит из инструкций, выполняемых процессором, данных и информации о выполняемой задаче, такой как размещенная память, открытые файлы и статус процесса.

В то же время не следует отождествлять процесс с программой хотя бы потому, что программа может породить более одного процесса. Простейшие программы, например, команда *who(1)* или *cat(1)*, при выполнении представлены только одним процессом. Сложные задачи, например системные серверы (печати, FTP, Telnet), порождают в системе несколько одновременно выполняющихся процессов.

Операционная система UNIX является многозадачной. Это значит, что одновременно может выполняться несколько процессов, причем часть процессов могут являться образцами одной программы.

Выполнение процесса заключается в точном следовании набору инструкций, который никогда не передает управление набору инструкций другого процесса. Процесс считывает и записывает информацию в раздел данных и в стек, но ему недоступны данные и стеки других процессов.

В то же время процессы имеют возможность обмениваться друг с другом данными с помощью предоставляемой UNIX системой межпроцессного взаимодействия. В UNIX существует набор средств взаимодействия между процессами, таких как сигналы (signals), каналы (pipes), разделяемая память (shared memory), семафоры (semaphores), сообщения (messages) и файлы, но в остальном процессы изолированы друг от друга.

## Типы процессов

### Системные процессы

Системные процессы являются частью ядра и всегда расположены в оперативной памяти. Системные процессы не имеют соответствующих им программ в виде исполняемых файлов и запускаются особым образом при инициализации ядра системы. Выполняемые инструкции и данные этих процессов находятся в ядре системы, таким образом они могут вызывать функции и обращаться к данным, недоступным для остальных процессов. Системными процессами являются: *shed* (диспетчер свопинга), *vhand* (диспетчер страничного замещения), *bdfflush* (диспетчер буферного кэша) и

kmadaemon (диспетчер памяти ядра). К системным процессам следует отнести init, являющийся прародителем всех остальных процессов в UNIX. Хотя init не является частью ядра, и его запуск происходит из исполняемого файла (`/etc/init`), его работа жизненно важна для функционирования всей системы в целом.

## Демоны

Демоны — это неинтерактивные процессы, которые запускаются обычным образом — путем загрузки в память соответствующих им программ (исполняемых файлов), и выполняются в фоновом режиме. Обычно демоны запускаются при инициализации системы (но после инициализации ядра, подробнее см. главу 3) и обеспечивают работу различных подсистем UNIX: системы терминального доступа, системы печати, системы сетевого доступа и сетевых услуг и т. п. Демоны не связаны ни с одним пользовательским сеансом работы и не могут непосредственно управляться пользователем. Большую часть времени демоны ожидают пока тот или иной процесс запросит определенную услугу, например, доступ к файловому архиву или печать документа.

## Прикладные процессы

К прикладным процессам относятся все остальные процессы, выполняющиеся в системе. Как правило, это процессы, порожденные в рамках пользовательского сеанса работы. С такими процессами вы будете сталкиваться чаще всего. Например, запуск команды `ls(l)` породит соответствующий процесс этого типа. Важнейшим пользовательским процессом является основной командный интерпретатор (`login shell`), который обеспечивает вашу работу в UNIX. Он запускается сразу же после вашей регистрации в системе, а завершение работы `login shell` приводит к отключению от системы.

Пользовательские процессы могут выполняться как в интерактивном, так и в фоновом режиме, но в любом случае время их жизни (и выполнения) ограничено сеансом работы пользователя. При выходе из системы все пользовательские процессы будут уничтожены.

Интерактивные процессы монопольно владеют терминалом, и пока такой процесс не завершит свое выполнение, пользователь не сможет работать с другими приложениями<sup>7</sup>.

Вы сможете работать с другими приложениями, если в функции интерактивного процесса входит запуск на выполнение других программ. Примером такой задачи является командный интерпретатор `shell`, который считывает пользовательский ввод и запускает соответствующие задачи. Более типичным в данном контексте является процесс, порожденный командой `ps(l)`. Пока `ps(l)` не завершит работу, вы не сможете вводить команды `shell`.

## Атрибуты процесса

Процесс в UNIX имеет несколько атрибутов, позволяющих операционной системе эффективно управлять его работой, важнейшие из которых рассмотрены ниже.

### Идентификатор процесса Process ID (PID)

Каждый процесс имеет уникальный идентификатор PID, позволяющий ядру системы различать процессы. Когда создается новый процесс, ядро присваивает ему следующий свободный (т. е. не ассоциированный ни с каким процессом) идентификатор. Присвоение идентификаторов происходит по возрастающей, т. е. идентификатор нового процесса больше, чем идентификатор процесса, созданного перед ним. Если идентификатор достиг максимального значения, следующий процесс получит минимальный свободный PID и цикл повторяется. Когда процесс завершает свою работу, ядро освобождает занятый им идентификатор.

### Идентификатор родительского процесса Parent Process ID (PPID)

Идентификатор процесса, породившего данный процесс.

### Приоритет процесса (Nice Number)

Относительный приоритет процесса, учитываемый планировщиком при определении очередности запуска. Фактическое же распределение процессорных ресурсов определяется *приоритетом выполнения*, зависящим от нескольких факторов, в частности от заданного относительного приоритета. Относительный приоритет не изменяется системой на всем протяжении жизни процесса (хотя может быть изменен пользователем или администратором) в отличие от приоритета выполнения, динамически обновляемого ядром.

### Терминальная линия (TTY)

Терминал или псевдотерминал, ассоциированный с процессом, если такой существует. Процессы-демоны не имеют ассоциированного терминала.

### Реальный (RID) и эффективный (EUID) идентификаторы пользователя

Реальным идентификатором пользователя данного процесса является идентификатор пользователя, запустившего процесс. Эффективный идентификатор служит для определения прав доступа процесса к системным ресурсам (в первую очередь к ресурсам файловой системы). Обычно реальный и эффективный идентификаторы эквивалентны, т. е. процесс имеет в системе те же права, что и пользователь, запустивший его. Однако существует возможность задать процессу более широкие права, чем права

пользователя путем установки флага SUID, когда эффективному идентификатору присваивается значение идентификатора владельца исполняемого файла (например, администратора).

### Реальный (RGID) и эффективный (EGID) идентификаторы группы

Реальный идентификатор группы равен идентификатору первичной или текущей группы пользователя, запустившего процесс. Эффективный идентификатор служит для определения прав доступа к системным ресурсам по классу доступа группы. Так же как и для эффективного идентификатора пользователя, возможна его установка равным идентификатору группы владельца исполняемого файла (флаг SGID).

Команда *ps(l)* (process status) позволяет вывести список процессов, выполняющихся в системе, и их атрибуты:

```
$ ps -ef |head -20
UID  PID  PPID  C  STIME   TTY    TIME   CMD
root  0    0     0  Dec 17   9      0:00   sched
root  1    0     0  Dec 17   9      0::01  /etc/init -
root  2    0     0  Dec 17   7      0::00  pageout
root  3    0     0  Dec 17   9      7::00  fsflush
root 164   1    164  Dec 17   9      0:01  /usr/lib/sendmail -bd -q1h
fed   627   311   0  Dec 17  pts/3   0::27  emiclock
fed   314   304   0  Dec 17  pts/4   0::00  /usr/local/bin/bash
fed   3521  512   0           0::01  <defunct>
```

Более подробное описание полей вывода команды *ps(l)* приведено далее в разделе "Основные утилиты UNIX".

### Жизненный путь процесса

Процесс в UNIX создается системным вызовом *fork(2)*. Процесс, сделавший вызов *fork(2)* называется *родительским*, а вновь созданный процесс — *дочерним*. Новый процесс является точной копией породившего его процесса. Как это ни удивительно, но новый процесс имеет те же инструкции и данные, что и его родитель. Более того, выполнение родительского и дочернего процесса начнется с одной и той же инструкции, следующей за вызовом *fork(2)*. Единственно, чем они различаются — это идентификатором процесса PID. Каждый процесс имеет одного родителя, но может иметь несколько дочерних процессов.

Для запуска задачи, т. е. для загрузки новой программы, процесс должен выполнить системный вызов *exec(2)*. При этом новый процесс не порождается, а исполняемый код процесса полностью замещается кодом запускаемой программы. Тем не менее окружение новой программы во многом сохраняется, в частности сохраняются значения переменных окружения, назначения стандартных потоков ввода/вывода, вывода сообщений об ошибках, а также приоритет процесса.

В UNIX запуск на выполнение новой программы часто связан с порождением нового процесса, таким образом сначала процесс выполняет вызов *fork(2)*, порождая дочерний процесс, который затем выполняет *exec(2)*, полностью замещаясь новой программой.

Рассмотрим эту схему на примере.

Допустим, пользователь, работая в командном режиме (в командном интерпретаторе shell) запускает команду *ls(1)*. Текущий процесс (shell) делает вызов *fork(2)*, порождая вторую копию shell. В свою очередь, порожденный shell вызывает *exec(2)*, указывая в качестве параметра имя исполняемого файла, образ которого необходимо загрузить в память вместо кода shell. Код *ls(1)* замещает код порожденного shell, и утилита *ls(1)* начинает выполняться. По завершении работы *ls(1)* созданный процесс "умирает". Пользователь вновь возвращается в командный режим. Описанный процесс представлен на рис. 1.5. Мы также проиллюстрируем работу командного интерпретатора в примере, приведенном в главе 2.

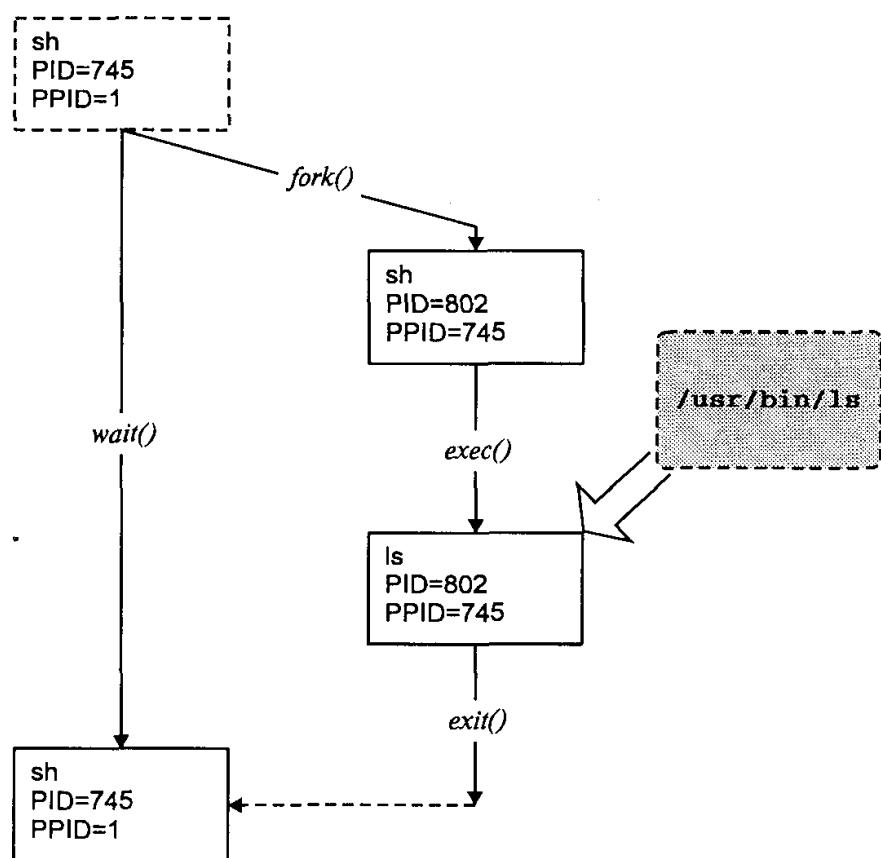


Рис. 1.5. Создание процесса и запуск программы

Если сделать "отпечаток" выполняемых процессов, например командой *ps(l)*, между указанными стадиями, результат был бы следующим:

Пользователь работает в командном режиме:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
user1	745	1	10	10:11:34	ttyp4	0:01	sh

Пользователь запустил команду *ls(l)*, и shell произвел вызов *fork(2)*:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
user1	745	1	10	10:11:34	tttyp4	0:01	sh
user1	802	745	14	11:00:00	tttyp4	0:00	sh

Порожденный shell произвел вызов *exec(2)*:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
user1	745	1	10	10:11:34	tttyp4	0:01	sh
user1	802	745	12	11:00:00	tttyp4	0:00	ls

Процесс *ls(l)* завершил работу:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
user1	745	1	10	10:11:34	tttyp4	0:01	sh

Описанная процедура запуска новой программы называется *fork-and-exec*.

Однако бывают ситуации, когда достаточно одного вызова *fork(2)* без последующего *exec(2)*. В этом случае исполняемый код родительского процесса должен содержать логическое ветвление для родительского и дочернего процессов<sup>8</sup>.

Все процессы в UNIX создаются посредством вызова *fork(2)*. Запуск на выполнение новых задач осуществляется либо по схеме *fork-and-exec*, либо с помощью *exec(2)*. "Праородителем" всех процессов является процесс *init(1M)*, называемый также *распределителем процессов*. Если построить граф "родственных отношений" между процессами, то получится дерево, корнем которого является *init(1M)*. Показанные на рис. 1.6 процессы *sched* и *vhand* являются системными и формально не входят в иерархию (они будут рассматриваться в следующих главах).

## Сигналы

Сигналы являются способом передачи от одного процесса другому или от ядра операционной системы какому-либо процессу уведомления о возникновении определенного события. Сигналы можно рассматривать как простейшую форму межпроцессного взаимодействия. В то же время сигналы больше напоминают программные прерывания, — средство, с помощью которого нормальное выполнение процесса может быть прервано. Например, если процесс производит деление на 0, ядро посылает ему сигнал *SIGFPE*, а при нажатии клавиш прерывания, обычно *<Del>* или *<Ctrl>+<C>*, текущему процессу посыпается сигнал *SIGINT*.

<sup>8</sup> Такое ветвление можно организовать на основании значения, возвращаемого системным вызовом *fork(2)*. Для родительского процесса *fork* возвращает идентификатор созданного дочернего процесса, а дочерний процесс получает значение, равное 0. Подробнее эти вопросы будут рассмотрены в главе 2.

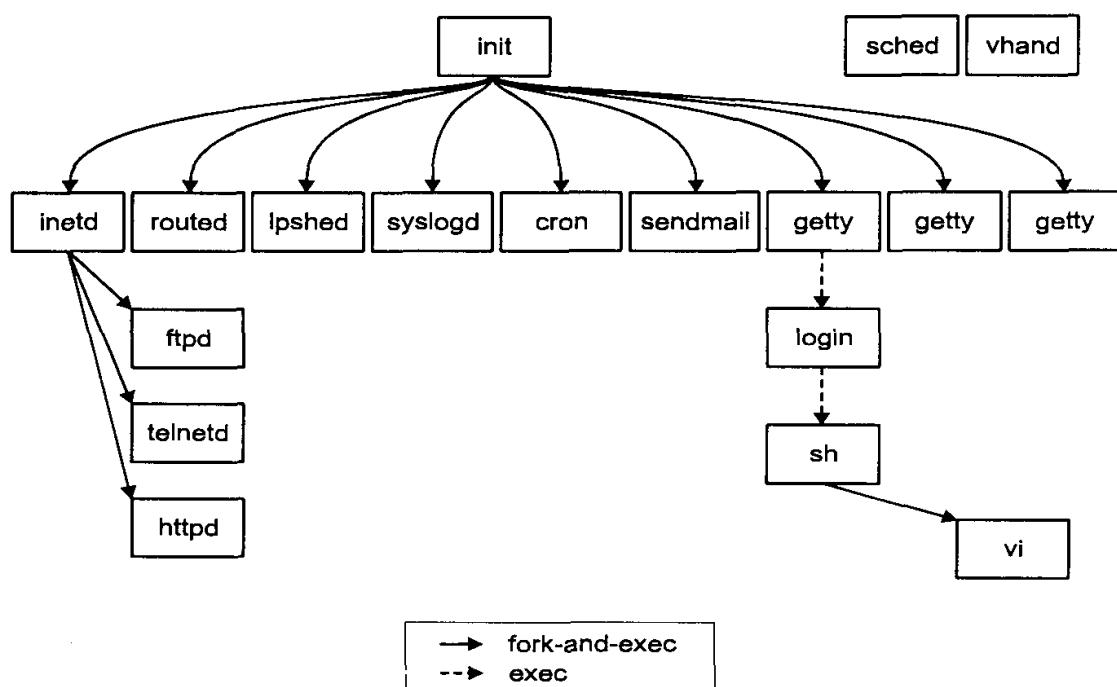


Рис. 1.6. Типично "дерево" процессов в UNIX

Для отправления сигнала служит команда *kill(1)*:

`kill sig_no pid`

где *sig\_no* — номер или символьическое название сигнала, а *pid* — идентификатор процесса, которому посыпается сигнал. Администратор системы может посыпать сигналы любым процессам, обычный же пользователь может посыпать сигналы только процессам, владельцем которых он является (реальный и эффективный идентификаторы процесса должны совпадать с идентификатором пользователя<sup>9</sup>). Например, чтобы послать процессу, который вы только что запустили в фоновом режиме, сигнал завершения выполнения SIGTERM, можно воспользоваться командой:

<code>\$ long_program &amp;</code>	Запустим программу в фоновом режиме
<code>\$ kill \$!</code>	По умолчанию команда <i>kill(1)</i> посыпает сигнал SIGTERM; переменная <i>\$!</i> содержит PID последнего процесса, запущенного в фоновом режиме

При получении сигнала процесс имеет три варианта действий для выбора:

1. Он может игнорировать сигнал. Не следует игнорировать сигналы, вызванные аппаратной частью, например, при делении на 0 или ссылке на недопустимые области памяти, так как дальнейшие результаты в отношении данного процесса непредсказуемы.

<sup>9</sup> Точнее, с реальным и эффективным идентификаторами процесса, посыпающего сигнал. Если вы посыпаете сигнал командой *kill(l)*, работая в shell, то речь идет о командном интерпретаторе.

2. Процесс может потребовать действия по умолчанию. Как ни печально, обычно это сводится к завершению выполнения процесса.
3. Наконец, процесс может перехватить сигнал и самостоятельно обработать его. Например, перехват сигнала SIGINT позволит процессу удалить созданные им временные файлы, короче, достойно подготовиться к "смерти". Следует иметь в виду, что сигналы SIGKILL и SIGSTOP нельзя ни перехватить, ни игнорировать.

По умолчанию команда *kill(1)* посыпает сигнал с номером 15 — SIGTERM<sup>10</sup>, действие по умолчанию для которого — завершение выполнения процесса, получившего сигнал.

Иногда процесс продолжает существовать и после отправления сигнала SIGTERM. В этом случае можно применить более жесткое средство — послать процессу сигнал SIGKILL с номером (9), — поскольку этот сигнал нельзя ни перехватить, ни игнорировать:

**\$ kill -9 pid**

Однако возможны ситуации, когда процесс не исчезает и в этом случае. Это может произойти для следующих процессов:

- Процессы-зомби. Фактически процесса как такового не существует, осталась лишь запись в системной таблице процессов, поэтому удалить его можно только перезапуском операционной системы. Зомби в небольших количествах не представляют опасности, однако если их много, это может привести к переполнению таблицы процессов.
- Процессы, ожидающие недоступные ресурсы NFS (Network File System), например, записывающие данные в файл файловой системы удаленного компьютера, отключившегося от сети. Эту ситуацию можно преодолеть, послав процессу сигнал SIGINT или SIGQUIT.
- Процессы, ожидающие завершения операции с устройством, например, перемотки магнитной ленты.

Сигналы могут не только использоваться для завершения выполнения процессов, но и иметь специфическое для приложения (обычно для системных демонов) значение (естественно, это не относится к сигналам SIGKILL и SIGSTOP). Например, отправление сигнала SIGHUP серверу имен DNS (*named(1M)*) вызовет считывание базы данных с диска. Для других приложений могут быть определены другие сигналы и соответствующие им значения.

Более подробно сигналы мы рассмотрим в главах 2 и 3.

Соответствие между символьными именами и номерами сигналов может отличаться различных версиях UNIX. Команда *kill -l* выводит номера сигналов и их имена.

## Устройства

Как уже отмечалось, UNIX "изолирует" приложения (а значит и пользователя) от аппаратной части вычислительной системы. Например, в имени файла отсутствует указатель диска, на котором этот файл расположен, а большая часть взаимодействия с периферийными устройствами неотличима от операций с обычными файлами.

UNIX предоставляет единый интерфейс различных устройств системы в виде специальных файлов устройств. Специальный файл устройства связывает прикладное приложение с драйвером устройства. Каждый специальный файл соответствует какому-либо физическому устройству (например, диску, накопителю на магнитной ленте, принтеру или терминалу) или т. н. псевдоустройству (например, сетевому интерфейсу, пустому устройству, сокету или памяти). Вся работа приложения с устройством происходит через специальный файл, а соответствующий ему драйвер обеспечивает выполнение операций ввода/вывода в соответствии с конкретным протоколом обмена данными с устройством.

Существует два типа специальных файлов устройств:

- Файлы блочных устройств
- Файлы символьных устройств

### Файлы блочных устройств

Файлы блочных устройств служат интерфейсом к устройствам, обмен данными с которыми происходит большими фрагментами, называемыми *блоками*. При этом ядро операционной системы обеспечивает необходимую буферизацию. Примером физических устройств, соответствующих этому типу файлов, являются жесткие диски. Приведем фрагмент подробного списка файлов каталога `/dev` системы Digital UNIX, отражающий файлы для доступа к первому и второму разделам первого диска SCSI:

```
brw-rw---- 1 root system 8, 1 Apr 18 11:03 /dev/rz0a
brw-rw---- 1 root system 8, 1 Apr 18 13:15 /dev/rz0b
```

### Файлы символьных устройств

Файлы символьных устройств используются для доступа к устройствам, драйверы которых обеспечивают собственную буферизацию и побайтную передачу данных. В качестве примера устройств с символьным интерфейсом можно привести терминалы, принтеры и накопители на магнитной ленте. Заметим, что одно и то же физическое устройство может иметь как блочный, так и символьный интерфейсы. Для блочных устройств такой интерфейс также называют *интерфейсом доступа низкого уровня* (raw

interface). Так, для побайтного доступа к разделам диска, приведенным в предыдущем примере, используются соответствующие файлы:

crw-----1	root	system	8, 1	Apr 18	11:04	/dev/rrz0a
crw-----1	root	system	8, 1	Apr 18	13:15	/dev/rrz0b
crw-r-----1	root	system	13,1	Apr 18	18:08	/dev/kmem
crw-rw-rw- 1	root	system	7, 0	Apr 18	15:20	/dev/ptyp0
crw-rw-rw- 1	root	system	8, 1	Apr 18	15:20	/dev/ptyp1
crw-rw-rw- 1	bin	terminal	3, 2	Apr 18	16:10	/dev/tty02
crw-rw-rw- 1	bin	terminal	3, 3	Apr 18	16:10	/dev/tty03

Последние три строки списка представляют интерфейс доступа к виртуальной памяти ядра и двум псевдотерминалам.

В поле размера файла (пятая колонка вывода команды *ls(1)*) у специальных файлов устройств выводятся два числа. Это так называемые *старшее* (major) и *младшее* (minor) числа. Часто драйвер обслуживает более одного устройства. При этом старшее число указывает ядру на конкретный драйвер (например, драйвер псевдо-терминалов), а младшее передается драйверу и указывает на конкретное устройство (например, конкретный псевдотерминал).

Интерфейс файловой системы для взаимодействия с устройством схематически представлен на рис. 1.7.

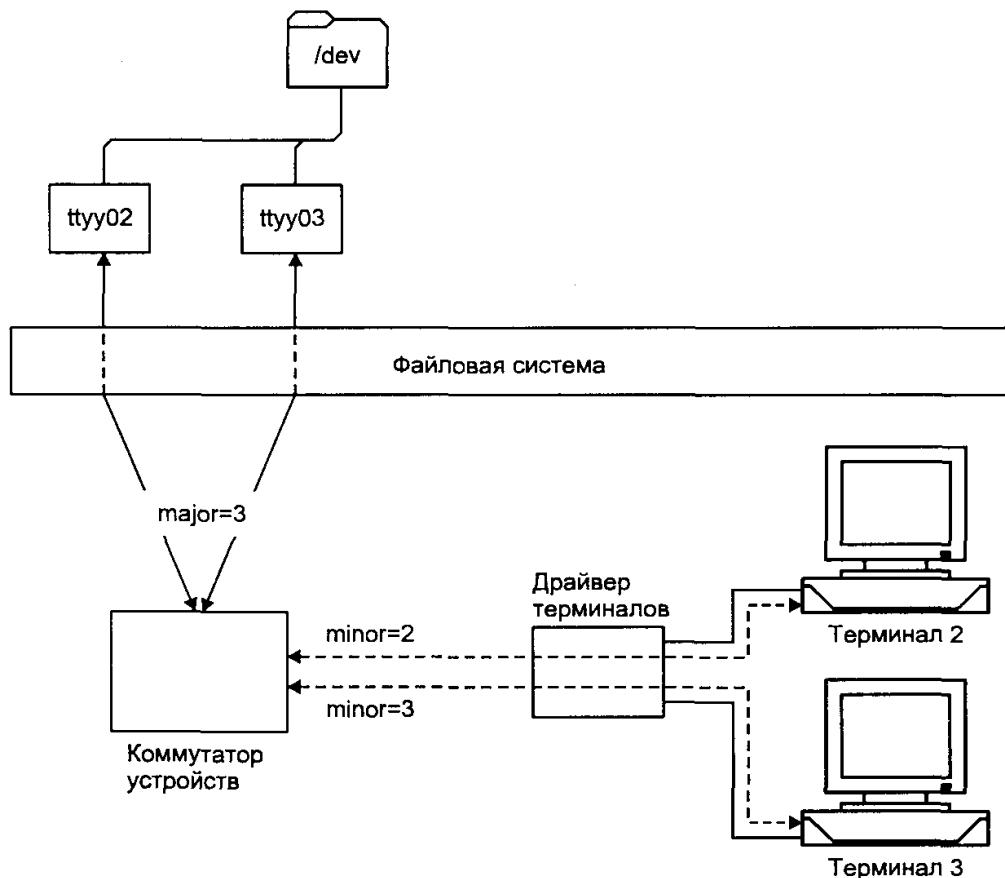


Рис. 1.7. Взаимодействие с устройством

## Мнемоника названий специальных файлов устройств в файловой системе UNIX

Названия специальных файлов устройств в большой степени зависят от конкретной версии UNIX. Тем не менее в этих названиях присутствует общая логика, позволяющая даже в незнакомой системе определить, какие файлы отвечают за конкретные устройства. Например, имена файлов доступа к дисковым устройствам обычно содержат указание на тип диска, номер контроллера, логический номер устройства, раздел диска и т. д. По названию также легко определить, какой вид доступа предоставляет данный интерфейс (блочный или символьный).

В качестве примера рассмотрим специальный файл устройства для доступа к разделу диска в операционной системе Solaris:

**/dev/dsk/c0t4d0s2**

Данный файл предоставляет блочный интерфейс, а соответствующий ему символьный (или необработанный) файл имеет имя:

**/dev/rdsk/c0t4d0s2**

Файлы доступа к дисковым устройствам располагаются в специальных подкаталогах — **/dev/dsk** (для блочных устройств) и **/dev/rdsk** (для символьных устройств). Такая структура хранения специальных файлов характерна для систем UNIX версии System V.

Имя файла, характерное для систем версии SVR4, можно представить в общем виде:

**cktlidmsn**

где *k* — номер контроллера, */* — номер устройства (для устройств SCSI это идентификатор устройства ID), *m* — номер раздела, а *n* — логический номер устройства (LUN) SCSI.

Таким образом файл устройства **/dev/rdsk/c0t4d0s2** обеспечивает доступ к первому разделу (нумерация разделов начинается с 0) диска с ID=4, LUN=2 первого контроллера.

Такой формат имен файлов в версии SVR4 применяется для всех дисковых устройств и накопителей на магнитной ленте. Иногда для этих стандартных имен в файловой системе имеются символические связи с более простыми названиями. Например, в Solaris имя **/dev/sd0a** может использоваться вместо **/dev/dsk/c0t3d0s**, также обеспечивая доступ к устройству:

```
lrwxrwxrwx 1 root root 12 Oct 31 17:48 /dev/sd0a ->dsk/c0t3d0s
```

В SCO UNIX имеются специальные файлы с более простыми именами **/dev/root**, **/dev/usr** и т. п., которые предоставляют доступ к разделам диска с такими же именами (root, usr).

Более простая мнемоника обнаруживается в именах специальных файлов других устройств. Так, например, параллельный порт в большинстве систем имеет имя `/dev/lpn`, где `p` — номер порта (0, 1 и т. д.). Терминальные линии, подключенные к последовательным портам компьютера обозначаются как `/dev/ttynn`, где `nn` является идентификатором линии. В табл. 1.6 приведены примеры других специальных файлов устройств.

**Таблица 1.6.** Имена некоторых специальных файлов устройств

Общий вид имени	Пример	Описание <b>устройства</b> , доступ к которому обеспечивается через файл
<code>/dev/rmtn</code>	<code>/dev/rmt0</code>	Накопитель на магнитной ленте
<code>/dev/nrmtn</code>	<code>/dev/nrmt0</code>	Накопитель на магнитной ленте в режиме без перемотки назад по окончании работы
<code>/dev/rstn</code>	<code>/dev/rst1</code>	SCSI-накопитель на магнитной ленте
<code>/dev/cdn</code>	<code>/dev/cd0</code>	CD-ROM
<code>/dev/cdrom</code>		
<code>/dev/ttypn</code>	<code>/dev/ttyp12</code>	Псевдотерминал (подчиненный)
<code>/dev/ptypn</code>	<code>/dev/ptyp5</code>	Псевдотерминал (мастер)
<code>/dev/console</code>		Системная консоль
<code>/dev/tty</code>		Синоним терминальной линии управляющего терминала для данного процесса
<code>/dev/mem</code>		Физическая оперативная память
<code>/dev/kmem</code>		Виртуальная память ядра
<code>/dev/null</code>		Нулевое устройство — весь вывод на него уничтожается, а при попытке ввода с этого устройства возвращается 0 байтов
<code>/dev/zero</code>		Нулевое устройство — весь вывод на него уничтожается, а ввод приводит к получению последовательности 0

## Пользователи системы

Прежде чем вы сможете начать работу в UNIX, вы должны стать *пользователем системы*, т. е. получить имя, пароль и ряд других атрибутов.

С точки зрения системы, пользователь — не обязательно человек. Пользователь является объектом, который обладает определенными правами, может запускать на выполнение программы и владеть файлами. В качестве пользователей могут, например, выступать удаленные компьютеры или группы пользователей с одинаковыми правами и функциями. Такие поль-

зователи называются *псевдопользователями*. Они обладают правами на определенные файлы системы и от их имени запускаются задачи, обеспечивающие ту или иную функциональность UNIX.

Как правило, большинство пользователей являются реальными людьми, которые регистрируются в системе, запускают те или иные программы, короче говоря, используют UNIX в своей работе.

В системе существует один пользователь, обладающий неограниченными правами. Это *суперпользователь* или *администратор системы*.

Каждый пользователь системы имеет уникальное имя (или *регистрационное имя* — `login name`). Однако система различает пользователей по ассоциированному с именем *идентификатору пользователя* или **UID** (User Identifier). Понятно, что идентификаторы пользователя также должны быть уникальными. Пользователь является членом одной или нескольких *групп* — списков пользователей, имеющих сходные задачи (например пользователей, работающих над одним проектом). Принадлежность к группе определяет дополнительные права, которыми обладают все пользователи группы. Каждая группа имеет уникальное имя (уникальное среди имен групп, имя группы и пользователя могут совпадать), но как и для пользователя, внутренним представлением группы является ее идентификатор **GID** (Group Identifier). В конечном счете **UID** и **GID** определяют, какими правами обладает пользователь в системе.

Вся информация о пользователях хранится в файле **/etc/passwd**. Это обычный текстовый файл, право на чтение которого имеют все пользователи системы, а право на запись имеет только администратор (суперпользователь). В этом файле хранятся пароли пользователей, правда в зашифрованном виде. Подобная открытость — недостаток с точки зрения безопасности, поэтому во многих системах зашифрованные пароли хранятся в отдельном закрытом для чтения и записи файле **/etc/shadow**.

Аналогично, информация о группах хранится в файле **/etc/group** и содержит списки пользователей, принадлежащих той или иной группе.

## Атрибуты пользователя

Как правило, все атрибуты пользователя хранятся в файле **/etc/passwd**. В конечном итоге, добавление пользователя в систему сводится к внесению в файл **/etc/passwd** соответствующей записи. Однако во многих системах информация о пользователе хранится и в других местах (например, в специальных базах данных), поэтому создание пользователя простым редактированием файла **/etc/passwd** может привести к неправильной регистрации пользователя, а иногда и к нарушениям работы системы. Вместо этого при возможности следует пользоваться специальными утилитами, поставляемыми с системой. Более подробно мы поговорим об этом при обсуждении задач администрирования UNIX в конце этой главы.

Сейчас же наша задача — разобраться, какую информацию хранит система о пользователе. Для этого рассмотрим фрагмент файла **/etc/passwd**:

```
root:x:0:1:0000-Admin(0000)::/bin/bash
daemon:x:1:1:0000-Admin(0000):/
bin:x:2:2:0000-Admin(0000):/usr/bin:
sys:x:3:3:0000-Admin(0000):/
adm:x:4:4:0000-Admin(0000):/var/adm:
lp:x:71:8:0000-lp(0000):/usr/spool/lp:
uucp:x:5:5:0000-uucp(0000):/usr/lib/uucp:
nobody:x:60001:60001:uid no body:/
andy:x:206:101:Andrei Robachevsky:/home/andy:/bin/bash
```

Каждая строка файла является записью конкретного пользователя и имеет следующий формат:

*name:passwd-encod: UID:GID:comments:home-dir:shell*

— всего семь полей (атрибутов), разделенных двоеточиями.

Рассмотрим подробнее каждый из атрибутов:

*name*

Регистрационное имя пользователя. Это имя пользователя вводит в ответ на приглашение системы *login*. Для небольших систем имя пользователя достаточно произвольно. В больших системах, в которых зарегистрированы сотни пользователей, требования уникальности заставляют применять определенные правила выбора имен.

*passwd-encod*

Пароль пользователя в закодированном виде. Алгоритмы кодирования известны, но они не позволяют декодировать пароль. При входе в систему пароль, который вы набираете, кодируется, и результат сравнивается с полем *passwd-encod*. В случае совпадения пользователю разрешается войти в систему.

Даже в закодированном виде доступность пароля представляет некоторую угрозу для безопасности системы. Поэтому часто пароль хранят в отдельном файле, а в поле *passwd-encod* ставится символ 'x' (в некоторых системах '!').

Пользователь, в данном поле которого стоит символ '\*', никогда не сможет попасть в систему. Дело в том, что алгоритм кодирования не позволяет символу '\*' появиться в закодированной строке. Таким образом, совпадение введенного и затем закодированного пароля и '\*' невозможно. Обычно такой пароль имеют псевдопользователи.

<i>UID</i>	Идентификатор пользователя является внутренним представлением пользователя в системе. Этот идентификатор наследуется задачами, которые запускает пользователь, и файлами, которые он создает. По этому идентификатору система проверяет пользовательские права (например, при запуске программы или чтении файла). Суперпользователь имеет $UID = 0$ , что дает ему неограниченные права в системе.
<i>GID</i>	Определяет <i>идентификатор первичной группы пользователя</i> . Этот идентификатор соответствует идентификатору в файле <b>/etc/group</b> , который содержит имя группы и полный список пользователей, являющихся ее членами. Принадлежность пользователя к группе определяет дополнительные права в системе. Группа определяет общие для всех членов права доступа и тем самым обеспечивает возможность совместной работы (например, совместного использования файлов).
<i>comments</i>	Обычно, это полное "реальное" имя пользователя. Это поле может содержать дополнительную информацию, например, телефон или адрес электронной почты. Некоторые программы (например, <i>finger(1)</i> и почтовые системы) используют это поле.
<i>home-dir</i>	Домашний каталог пользователя. При входе в систему пользователь оказывается в этом каталоге. Как правило, пользователь имеет ограниченные права в других частях файловой системы, но домашний каталог и его подкаталоги определяют область файловой системы, где он является полноправным хозяином.
<i>shell</i>	Имя программы, которую UNIX использует в качестве командного интерпретатора. При входе пользователя в систему UNIX автоматически запустит указанную программу. Обычно это один из стандартных командных интерпретаторов <b>/bin/sh</b> (Bourne shell), <b>/bin/csh</b> (C shell) или <b>/bin/ksh</b> (Korn shell), позволяющих пользователю вводить команды и запускать задачи. В принципе, в этом поле может быть указана любая программа, например, командный интерпретатор с ограниченными функциями (restricted shell), клиент системы управления базой данных или даже редактор. Важно то, что, завершив выполнение этой задачи, пользователь автоматически выйдет из системы. Некоторые системы имеют файл <b>/etc/shells</b> , содержащий список программ, которые могут быть использованы в качестве командного интерпретатора.

## Пароли

Наличие пароля позволяет защитить ваши данные, а возможно (если вы — суперпользователь) и всю систему в целом. Уточним: наличие хорошего пароля, потому что неверно выбранный пароль — серьезная брешь в безопасности системы. Поэтому мы более подробно остановимся на основных рекомендациях по выбору пароля.

Назначить или изменить пароль можно командой *passwd(1)*. Обычный пользователь может изменить свой пароль, администратор может назначить пароль любому пользователю.

Перед запуском программы *passwd(1)* стоит держать в голове общее правило выбора пароля: пароль должен хорошо запоминаться и быть трудным для подбора.

Не рекомендуется записывать пароль, его необходимо запомнить. Собственная фамилия, кличка любимой собаки, год и месяц рождения, безусловно, легки для запоминания, но такие пароли нетрудно подобрать. Многие системы предлагают пароль, сгенерированный самой системой. Предполагается, что он совершенно лишен какого-либо смысла, т. е. не содержит имен, названий и вообще каких-либо произносимых слов. Хотя система предлагает его в виде, удобном для запоминания, это не всегда помогает.

Если по правилам работы в вашей системе можно самостоятельно выбрать пароль, постарайтесь подобрать что-нибудь, что легче будет запомнить. Никогда не используйте примеры паролей, приводимые в книгах и руководствах (чтобы не было искушения, в этой книге примеры не приводятся).

Многие системы требуют, чтобы пароль удовлетворял следующим требованиям:

- длина пароля не должна быть меньше шести символов;
- пароль должен включать по крайней мере 2 алфавитных символа и одну цифру или специальный символ;
- пароль должен содержать хотя бы 3 символа, не встречавшихся в вашем предыдущем пароле.

Пароли играют значительную роль в обеспечении безопасности системы. Общие рекомендации, адресованные прежде всего администраторам, можно свести к следующим:

1. В системе не должно существовать незащищенных пользовательских входов. Это относится как к пользовательским входам без пароля, так и ко входам пользователей, покинувших систему. Если пользователь длительное время не работает в системе, удалите его запись или хотя бы защитите его вход символом '\*' в поле пароля.
2. Если ваша система допускает, установите минимальную длину пароля. В зависимости от требований безопасности в системе это число может варьироваться от 8 до 12.

**3. Всегда меняйте пароль в следующих случаях:**

- если кто-либо узнал ваш пароль.
- если пользователь больше не работает в вашей системе, все пароли, которые он знал, должны быть изменены.
- если меняется администратор системы, должны быть изменены все системные пароли.
- если у вас появилось подозрение, что файл паролей был считан по сети, будет разумным сменить все пароли в системе.

4. Пароль администратора должен периодически меняться, независимо от обстоятельств.

5. Это может показаться странным, но не стоит заставлять пользователей менять пароли чересчур часто. Скорее всего, в этом случае пользователь выберет не лучший пароль. Но менять пароли все же следует. Частота смены зависит от степени доступности вашей системы (изолированная станция, сервер с сетевым доступом, наличие сетевых экранов).

Не преуменьшайте роль паролей в системе.

**Стандартные пользователи и группы**

После установки UNIX обычно уже содержит несколько зарегистрированных пользователей. Перечислим основные из них (в разных версиях системы UID этих пользователей могут незначительно отличаться):

<b>Имя</b>	<b>Пользователь</b>
root	Суперпользователь, администратор системы, UID=0. Пользователь с этим именем имеет неограниченные полномочия в системе. Для него не проверяются права доступа, и таким образом он имеет все "рычаги" для управления системой. Для выполнения большинства функций администрирования требуется вход именно с этим именем. Следует отметить, что root — это только имя. На самом деле значение имеет UID. Любой пользователь с UID=0 имеет полномочия суперпользователя
adm	Псевдопользователь, владеющий файлами системы ведения журналов
bin	Обычно это владелец всех исполняемых файлов, являющихся командами UNIX
cron	Псевдопользователь, владеющий соответствующими файлами, от имени которого выполняются процессы подсистемы запуска программ по расписанию
lp или lpd	Псевдопользователь, от имени которого выполняются процессы системы печати, владеющий соответствующими файлами
news	Псевдопользователь, от имени которого выполняются процессы системы телеконференций

(продолжение)

<b>Имя</b>	<b>Пользователь</b>
nobody	Псевдопользователь, используемый в работе NFS
иисср	Псевдопользователь подсистемы UNIX-to-UNIX copy (иисср), позволяющей передавать почтовые сообщения и файлы между UNIX-хостами

Новая система также содержит ряд предустановленных групп. Поскольку группы, как правило, менее значимы, приведем лишь две категории:

<b>Имя</b>	<b>Группа</b>
root или wheel	Административная группа, GID=0
user или users или staff	Группа, в которую по умолчанию включаются все обычные пользователи UNIX

## **Пользовательская среда UNIX**

Сегодня характер работы в UNIX существенно отличается от того, каким он был, скажем, пятнадцать лет назад. Графический многооконный интерфейс, миллионы цветов, системы меню, техника drag-and-drop, — все это, казалось бы, стирает различия в работе с UNIX и, например, с Windows NT. Но взгляните внимательнее на экран монитора — и вы обязательно найдете хотя бы одно окно простого алфавитно-цифрового терминала.

Это — базовая пользовательская среда. Интерфейс командной строки может показаться безнадежно устаревшим, но в случае с UNIX это — самый непосредственный способ выполнения множества небольших задач администрирования. И программа, с которой вы рано или поздно столкнетесь, — командный интерпретатор shell. Поэтому здесь мы рассмотрим базовый пример работы в UNIX — использование командной строки интерпретатора shell.

## **Командный интерпретатор shell**

Все современные системы UNIX поставляются по крайней мере с тремя командными интерпретаторами: Bourne shell (**/bin/sh**), C shell (**/bin/csh**) и Korn shell (**/bin/ksh**). Существует еще несколько интерпретаторов, например Bourne-Again shell (**bash**), со сходными функциями.

Командный интерпретатор занимает важное место в операционной системе UNIX, прежде всего, благодаря следующим обстоятельствам:

1. Первая программа, с которой по существу начинается работа пользователя, — shell. В UNIX реализуется следующий сценарий работы в системе (рис. 1.8):
  - При включении терминала активизируется процесс *getty(1M)*, который является сервером терминального доступа и запускает программу *login(1)*<sup>11</sup>, которая, в свою очередь, запрашивает у пользователя имя и пароль.
  - Если пользователь зарегистрирован в системе и ввел правильный пароль, *login(1)* запускает программу, указанную в последнем поле записи пользователя в файле **/etc/passwd**. В принципе это может быть любая программа, но в нашем случае — это командный интерпретатор shell.
  - Shell выполняет соответствующий командный файл инициализации, и выдает на терминал пользователя приглашение. С этого момента пользователь может вводить команды.
  - Shell считывает ввод пользователя, производит синтаксический анализ введенной строки, подстановку шаблонов и выполняет действие, предписанное пользователем (это может быть запуск программы, выполнение внутренней функции интерпретатора) или сообщает об ошибке, если программа или функция не найдены.
  - По окончании работы пользователь завершает работу с интерпретатором, вводя команду *exit*, и выходит из системы.
2. Командный интерпретатор является удобным средством программирования. Синтаксис языка различных командных интерпретаторов несколько отличается, в качестве базового мы рассмотрим командный интерпретатор Bourne. С помощью shell вы можете создавать сложные программы, конструируя их, как из кирпичиков, из существующих утилит UNIX. Программы на языке shell часто называют *скриптами* или *сценариями* (script). Интерпретатор считывает строки из файла-скрипта и выполняет их, как если бы они были введены пользователем в командной строке.
3. Как уже упоминалось, при входе пользователя в систему запускается его инициализационный скрипт, выполняющий несколько функций: установку пути поиска программ, инициализацию терминала, определение расположения почтового ящика. Помимо этого может быть выполнен целый ряд полезных действий, — например, установка приглашения. Скорее всего вам придется "покопаться" в этом скрипте, по крайней мере, чтобы добавить необходимые пути поиска. Инициализационный скрипт находится в домашнем каталоге пользователя.

<sup>11</sup> В данном разделе мы не останавливаемся на подробностях запуска *login(1)*. Эти вопросы будут рассмотрены позднее в главе 3.

Для разных командных интерпретаторов используются различные скрипты инициализации:

### Командный интерпретатор

- Bourne shell (sh)
- C shell (csh)
- Korn shell (ksh)
- Bourne-Again shell (bash)

### Скрипт инициализации

- .profile
- .login и .cshrc
- .profile и .kshrc
- .profile и .bashrc

Скрипты .profile и .login выполняются при первом входе в систему. Скрипты .cshrc, .kshrc и .bashrc выполняются при каждом запуске интерпретатора.

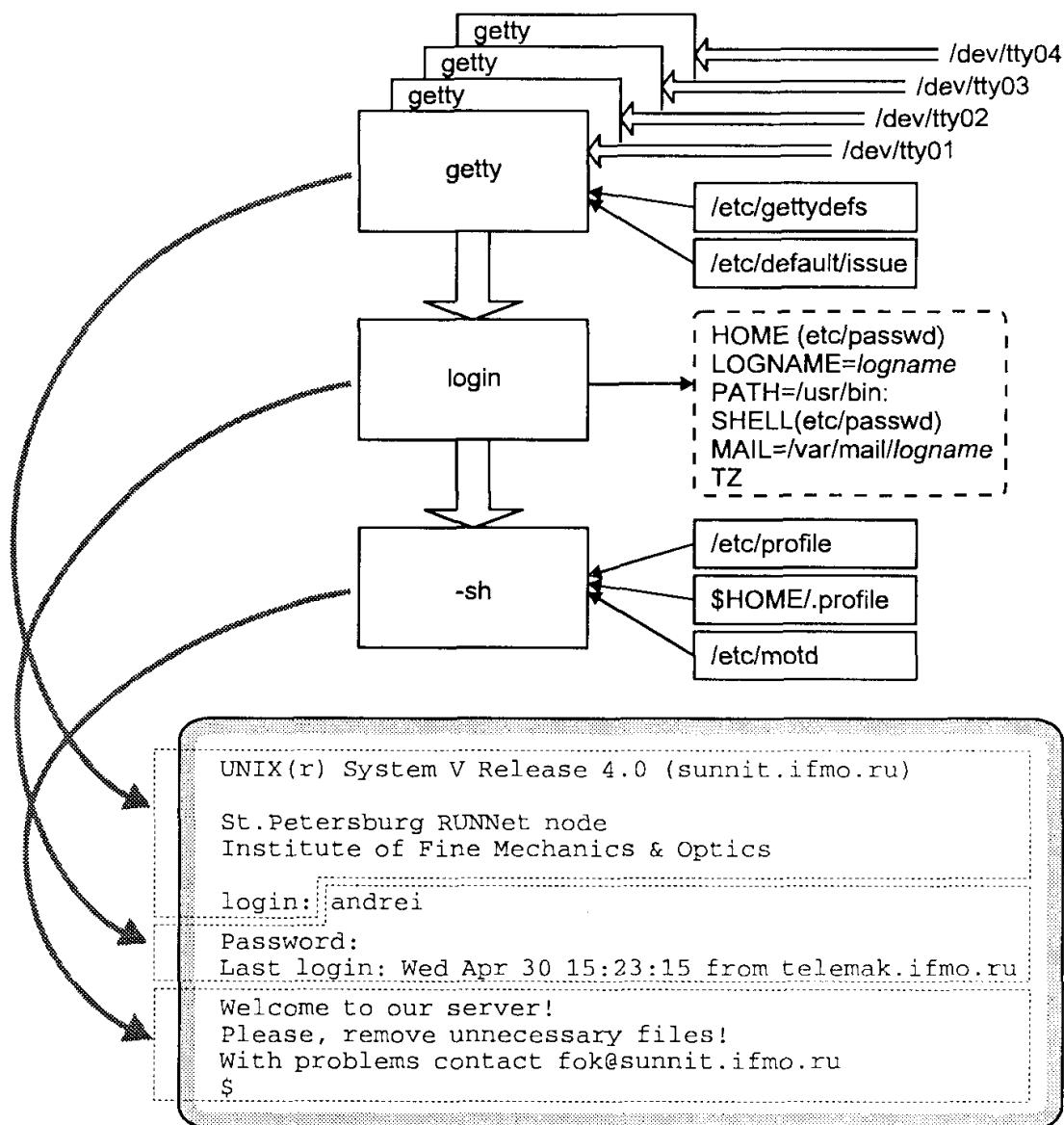


Рис. 1.8. Процессы, обеспечивающие вход пользователя в систему

4. Наконец, основная инициализация операционной системы происходит в результате выполнения скриптов shell. Если вам понадобится модифицировать процесс инициализации (например, добавить новый системный сервис), то придется заглянуть в эти скрипты.

## Синтаксис языка Bourne shell

Любой из стандартных командных интерпретаторов имеет развитый язык программирования, позволяющий создавать командные файлы, или скрипты, для выполнения достаточно сложных задач. Следует, однако, иметь в виду, что shell является интерпретатором, он последовательно считывает команды из скрипта и выполняет их, как если бы они последовательно вводились пользователем с терминала. При таком характере работы трудно ожидать большой производительности от скриптов, однако их эффективность определяется простотой и наглядностью. Если же производительность программы играет главную роль, то самым эффективным средством по-прежнему остается язык программирования С.

В этом разделе приведены сведения о языке Bourne shell, достаточные, чтобы разобраться в системных скриптах и написать простейшие скрипты. Данное описание ни в коем случае не претендует на полное руководство по программированию на языке командного интерпретатора.

### Общий синтаксис скрипта

Как уже было замечено, скрипт представляет собой обычный текстовый файл, в котором записаны инструкции, понятные командному интерпретатору. Это могут быть команды, выражения shell или функции. Командный интерпретатор считывает эти инструкции из файла и последовательно выполняет их.

Безусловно, как и в случае любого другого языка программирования, применение комментариев существенно облегчает последующее использование и модификацию написанной программы. В Bourne shell комментарии начинаются с символа '#':

```
# Этот скрипт выполняет поиск "мусора" (забытых временных
# файлов, файлов core и т.п.) в каталогах пользователей
```

Комментарии могут занимать не всю строку, а следовать после команды:

```
find /home -name core -print # Выполним поиск файлов core
```

Поскольку в системе могут существовать скрипты для различных интерпретаторов, имя интерпретирующей команды обычно помещается в первой строке следующим образом:

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

В данном случае последующий текст скрипта будет интерпретироваться Bourne shell. Заметим, что при запуске скрипта из командной строки (для

этого он должен обладать правом на выполнение — x), будет запущен новый командный интерпретатор, ввод команд для которого будет выполняться из файла скрипта.

### Переменные

В командной строке или скрипте командного интерпретатора можно определить и использовать переменные. Значением переменной является строка, которая передается присвоением:

```
var=value,
```

где var — имя переменной, а *value* — ее значение.

Значение переменной можно получить, используя знак '\$'. Например, вывести значение переменной name на экран можно с помощью команды echo следующим образом:

```
$ echo $name
```

Так же можно присвоить другой переменной (name1) значение переменной name:

```
$ name1=$name
```

Значение переменной можно присвоить иначе. Поскольку значение представляет собой строку, shell предоставляет удобный способ генерации строк из потока вывода команды. Синтаксис присвоения при этом следующий:

```
var=`command`
```

Так, например, где var — имя переменной, а *command* — название команды, команда *pwd(1)* выводит строку со значением текущего каталога:

```
$ pwd
/usr/home/andrei/test
```

Можно присвоить переменной cdir значение текущего каталога, которое сохранится в ней:

```
$ cdir=`pwd`
$ echo $cdir
/usr/home/andrei/test
$ cd /usr/bin
$ pwd
/usr/bin
$ cd $cdir
$ pwd
/usr/home/andrei/test
```

При использовании переменной, например var, командный интерпретатор подставляет вместо \$var ее значение. Более сложные синтаксические конструкции получения значения переменной приведены в табл. 1.7.

**Таблица 1.7.** Способы получения значения переменной

<code>\$var</code>	Значение var; ничего, если переменная var не определена
<code> \${var}</code>	То же, но отделяет имя переменной var от последующих символов
<code> \${var:-string}</code>	Значение var, если определено; в противном случае — string. <u>Значение var при этом не изменяется</u>
<code> \${var:=string}</code>	То же, но если переменная var не определена, ей присваивается значение строки string
<code> \${var:?string}</code>	Если переменная var не определена, выводится строка string и интерпретатор прекращает работу. Если строка string пуста, то выводится сообщение var: parameter not set
<code> \${var:+string}</code>	Строка string, если переменная var определена, в противном случае — ничего

Приведем несколько примеров, используя команду echo:

```
$ var=user1
$ var1=user2
$ echo $var1
user2
$ echo ${var}1
user1
$ echo ${var1:+"do you want to redefine var?"}
do you want to redefine var?
```

Для нормальной работы в UNIX ряд переменных должен быть определен и зависит от тех приложений, с которыми вы работаете. Приведем несколько наиболее употребительных переменных:

Имя	Описание	Возможное значение
HOME	Каталог верхнего уровня пользователя	/usr/'logname' <sup>12</sup>
PATH	Поисковый путь	/bin:/etc:/usr/bin:.
MAIL	Имя почтового ящика	/usr/spool/mail/'logname'
TERM	Имя терминала	ansi
PS1	Первичное приглашение shell	#
PS2	Вторичное приглашение shell	>

Начальное окружение вашего сеанса устанавливается программой *login(1)*, исходя из записей в файле паролей, и имеет следующий вид:

#### Переменная окружения

`HOME=домашний_каталог`

`LOGNAME=зарегистрированное_имя`

#### Поле файла паролей

<sup>12</sup> В данном примере утилита *logname(1)* выводит регистрационное имя пользователя, таким образом для пользователя andrei переменная HOME примет следующее значение: /usr/andrei.

(продолжение)

<b>Переменная окружения</b>	<b>Поле файла паролей</b>
<u>PATH=/usr/bin:</u>	-
<u>SHELL=интерпретатор_сессии</u>	7
<u>MAIL=/var/mail/зарегистрированное_имя</u>	1
<u>TZ=временная_зона</u>	определенено системой

Переменная **HOME** в основном используется в команде *cd*, которая служит для перехода в каталог:

```
$ pwd
/u/usr
$ cd some/new/directory
$ pwd
/u/usr/some/new/directory
```

В результате текущим каталогом (команда *pwd(1)* выводит на терминал полное имя текущего каталога) становится **/u/usr/some/new/directory**. Вызов команды *cd* без параметра эквивалентен следующему вызову:

```
$ cd $HOME
```

который вернет вас в домашний каталог.

Переменная **PATH** служит для поиска командным интерпретатором запускаемых на выполнение программ, если их имя не содержит пути. Например, при запуске программы:

```
$ run
```

интерпретатор попытается найти файл *run* в каталогах пути поиска. В то же время при запуске программы *run* с указанием пути, переменная **PATH** использоваться не будет:

```
$ ./run
```

В последнем примере было задано относительное имя программы (относительно текущего каталога, обозначаемого точкой). Предполагается, что файл программы имеется в текущем каталоге, в противном случае *shell* выведет сообщение об ошибке.

Каталоги поиска в переменной **PATH** разделены символом **'.'**. Заметим, что текущий каталог поиска должен быть задан явно **('.)'**, *shell* не производит поиск в текущем каталоге по умолчанию.

Поиск запускаемых программ в текущем каталоге таит потенциальную опасность, поэтому для суперпользователя переменная **PATH** обычно инициализируется без **'.'**. Рассмотрим следующую ситуацию. Злоумышленник создает программу, наносящую вред системе (удаляющую файл паролей), помещает ее в каталог общего пользования, например в */tmp*, открытый на запись всем пользователям системы, с именем **ls**. Известно, что в

UNIX существует стандартная команда *ls(1)* (она обычно находится в каталоге **/bin**), выводящая на экран список файлов каталога. Допустим теперь, что администратор системы делает текущим каталог **/tmp** и хочет вывести список файлов данного каталога. Если текущий каталог ('.') расположен в пути поиска (переменной PATH) раньше каталога **/bin**, то выполнится программа, "подложенная" злоумышленником. Даже если текущий каталог указан последним в пути поиска, все равно существует вероятность, что вы захотите запустить команду, которая расположена в каталоге, не попавшем в переменную PATH, на самом деле вы можете запустить троянского коня.

Переменная MAIL определяет местоположение вашего почтового ящика, программы работы с электронной почтой используют эту переменную. Переменная MAIL инициализируется программой *login(1)*.

Переменная TERM содержит имя терминала и используется программами для доступа к базе данных терминалов. Обычно это программы, обеспечивающие полноэкранный режим работы, цвета и системы меню (редакторы, различные пользовательские оболочки). Поскольку наборы команд работы с различными терминалами отличаются друг от друга, используется специальная база данных, где хранятся конкретные команды для конкретного терминала.

Переменные PS1 и PS2 устанавливают первичное и вторичное приглашения командного интерпретатора. *Первичное приглашение* указывает на готовность интерпретатора к вводу команд. Значение этой переменной устанавливается при исполнении инициализационного скрипта (**.profile**) при входе пользователя в систему, и имеет вид "\$" для обычных пользователей и "#" для суперпользователя. Однако вид приглашения легко изменить, соответствующим образом задав значение переменной PS1. Например, если вы хотите, чтобы в приглашении присутствовало имя хоста, на котором вы работаете, задайте значение PS1 следующим образом:

```
PS1='`uname - n`">"
```

В этом случае, если имя вашей системы, например, *telemak*, при входе в систему командный интерпретатор выведет следующее приглашение:

```
telemak>
```

*Вторичное приглашение* появляется, если вы нажали клавишу *<Enter>*, синтаксически не закончив ввод команды. Например:

```
$ while :          нажатие клавиши <Enter>
> do             нажатие клавиши <Enter>
> echo Привет!    нажатие клавиши <Enter>
> done            нажатие клавиши <Enter>
```

После этого вы увидите слово "Привет!", выводимое на экран в бесконечном цикле. (Если вы все-таки воспроизвели этот пример, нажмите клавиши *<Ctrl>+<C>* или *<Del>*.)

Переменные, которые определены, являются внутренними переменными командного интерпретатора и не попадают в его окружение автоматически. Таким образом, они не могут быть использованы другими программами, запускаемыми из shell (окружение наследуется порожденными процессами). Для того чтобы поместить необходимые переменные в окружение shell и тем самым сделать их доступными для других приложений, эти переменные должны быть отмечены как экспортные. В этом случае при вызове какой-либо программы они автоматически попадут в ее окружение. Например, программа работы с электронной почтой получает имя файла — почтового ящика через переменную MAIL, программы, работающие с терминалом, например полноэкранный редактор, обращаются к базе данных терминалов, используя переменную TERM. Разработанная вами программа также может получать часть информации через переменные окружения. Для этого она должна использовать соответствующие функции (*getenv(3C)* и *putenv(3C)*), которые мы подробнее рассмотрим в следующей главе.

### Встроенные переменные

Помимо переменных, определяемых явно, shell имеет ряд внутренних переменных, значения которых устанавливаются самим интерпретатором. Поскольку это внутренние переменные, имя переменной вне контекста получения ее значения не имеет смысла (т. е. не существует переменной #, имеет смысл лишь ее значение \$#). Эти переменные приведены в табл. 1.8.

**Таблица 1.8.** Внутренние переменные shell

<u>\$1, \$2, ...</u>	<u>Позиционные параметры скрипта</u>
<u>\$#</u>	Число позиционных параметров скрипта
<u>\$?</u>	Код возврата последнего выполненного процесса
<u>\$\$</u>	<u>PID текущего shell</u>
<u>\$!</u>	<u>РЮ последнего процесса, запущенного в фоновом режиме</u>
<u>\$*</u>	Все параметры, переданные скрипту. Передаются как единое слово, будучи заключенным в кавычки: "\$\$" = "\$1 \$2 \$3 ..."
<u>\$@</u>	Все параметры, переданные скрипту. Передаются как отдельные слова, будучи заключенным в кавычки: "\$@" = "\$1" "\$2" "\$3" ..."

Эти переменные редко используются при работе в командной строке, основная область их применения — скрипты. Рассмотрим несколько примеров.

#### Текст скрипта test1.sh: Запуск скрипта

```
#!/bin/sh
echo скрипт $0
echo $1 $2 $3
shift
echo $1 $2 $3
```

```
$ ./test1.sh a1 a2 a3
скрипт ./test.sh
a1 a2 a3
a2 a3 a4
```

Переменные \$1, \$2, ... \$9 содержат значения позиционных параметров — аргументов запущенного скрипта. В \$1 находится первый аргумент (a1), в \$2 — a2 и т. д. до девятого аргумента. При необходимости передать большее число аргументов, требуется использовать команду *shift n*, производящую сдвиг значений аргументов на *n* позиций (по умолчанию — на одну позицию). Приведенный скрипт иллюстрирует этот прием. В переменной \$0 находится имя запущенного скрипта. Здесь наблюдается полная аналогия с массивом параметров argv[], передаваемом программе на языке С.

Значение \$# равно числу позиционных параметров. Его удобно использовать при проверке соответствия числа введенных пользователем параметров требуемому.

#### Текст скрипта test2.sh:

```
#!/bin/sh
if [ $# -lt 2 ]
then
  echo usage: $0 arg1 arg2
  exit 1
fi
```

#### Запуск скрипта

```
$ test2.sh
usage: test2.sh arg1 arg2
$ test2.sh h1 h2
```

В данном примере использовано условное выражение *if* и проверка, которые мы рассмотрим ниже.

Код возврата последней выполненной задачи (\$?) удобно использовать в условных выражениях. По правилам успешным завершением задачи считается код возврата, равный 0, ненулевой код возврата свидетельствует об ошибке. Код возврата скриптов генерируется с помощью команды *exit n*, где *n* — код возврата (см. предыдущий пример). В приведенном ниже примере определяется, зарегистрирован ли в системе пользователь с именем "sergey". Для этого программой *grep(l)* производится поиск слова *sergey* в файле паролей. В случае удачи *grep(l)* возвращает 0. Если слово не найдено, то *grep(l)* возвращает ненулевое значение, в данном случае это свидетельствует, что пользователь с именем *sergey* в системе не зарегистрирован.

#### Текст скрипта test3.sh:

```
#!/bin/sh
grep sergey /etc/passwd
if [ $? -ne 0 ]
then
  echo пользователь sergey в системе не зарегистрирован
fi
```

Каждый активный процесс в UNIX имеет уникальный идентификатор процесса, PID. Запуская скрипт, вы порождаете в системе процесс с уникальным PID. Значение PID сохраняется в переменной *\$\$*. Эту переменную удобно использовать в названиях временных файлов, поскольку их имена будут уникальными, например:

**Текст скрипта test4.sh:**

```
#!/bin/sh
 tempfile=/usr/tmp/tmp.$$
rm $tempfile
```

**Перенаправление ввода/вывода**

Каждая запущенная из командного интерпретатора программа получает три открытых потока ввода/вывода:

- стандартный ввод
- стандартный вывод
- стандартный вывод ошибок

По умолчанию все эти потоки ассоциированы с терминалом. То есть любая программа, не использующая потоки, кроме стандартных, будет ожидать ввода с клавиатуры терминала, весь вывод этой программы, включая сообщения об ошибках, будет происходить на экран терминала. Большое число утилит, с которыми вам предстоит работать, используют только стандартные потоки. Для таких программ shell позволяет независимо перенаправлять потоки ввода/вывода. Например, можно подавить вывод сообщений об ошибках, установить ввод или вывод из файла и даже передать вывод одной программы на ввод другой.

В табл. 1.9 приведен синтаксис перенаправления ввода/вывода, а на рис. 1.9 схематически показаны примеры перенаправления потоков.

**Таблица 1.9.** Перенаправление потоков ввода/вывода

>file	Перенаправление стандартного потока вывода в файл file
>>file	Добавление в файл file данных из стандартного потока вывода
<file	Получение стандартного потока ввода из файла file
p1 p2	Передача стандартного потока вывода программы p1 в поток ввода p2
n>file	Переключение потока вывода из файла с дескриптором n в файл file
n>>file	То же, но записи добавляются в файл file
n>&m	Слияние потоков с дескрипторами n и m
<<str	"Ввод здесь": используется стандартный поток ввода до подстроки str. При этом выполняются подстановки метасимволов командного интерпретатора
<<\str	То же, но подстановки не выполняются

Рассмотрим несколько примеров перенаправления потоков.

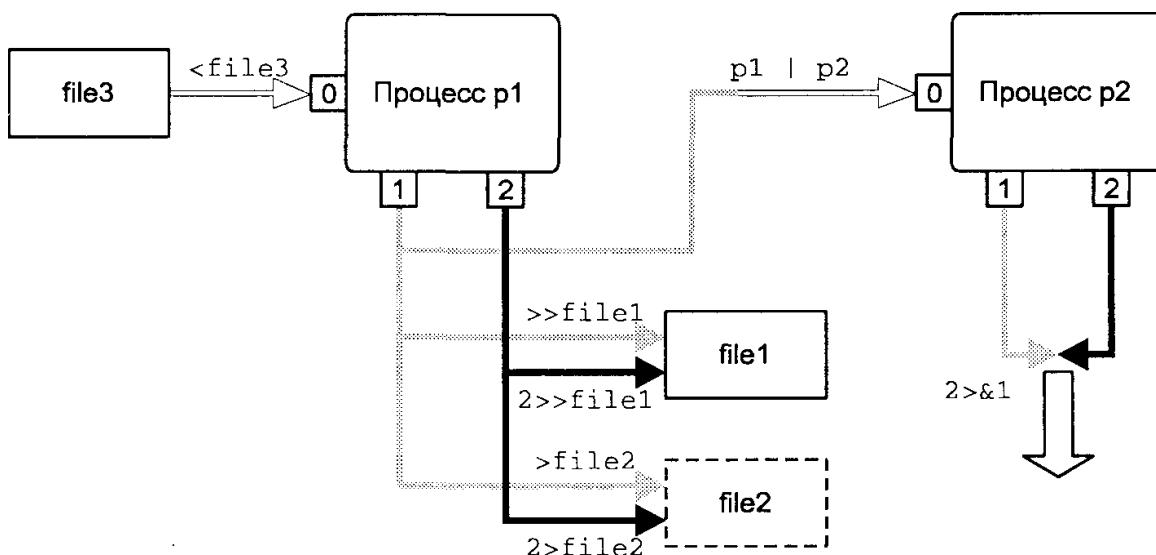
Запуск некой программы ведения журнала можно выполнить следующим образом:

```
$ logger >> file.log
```

При этом вывод программы logger будет записываться в конец файла **file.log**, сохраняя все предыдущие записи. Если файла **file.log** не существует, он будет создан. В отличие от этого, использование символа '**>**' указывает, что сначала следует очистить файл, а затем производить запись.

Стандартным потокам ввода, вывода и вывода ошибок присваиваются дескрипторы — числовые значения, являющиеся указателями на соответствующий поток. Они, соответственно, равны 0, 1 и 2. Перенаправлять потоки можно, используя эти числовые значения. Таким образом, предыдущему примеру эквивалентна следующая запись:

```
$ logger 1>>file . log
```



**Рис. 1.9.** Пример перенаправления стандартных потоков ввода/вывода

Чаще всего числовое значение дескриптора потока используется для потока ошибок. Например, чтобы подавить вывод ошибок, можно использовать следующую запись:

```
$ run 2>/dev/null
```

где **/dev/null** является псевдоустройством, удаляющим все введенные в него символы.

Командный интерпретатор предоставляет возможность слияния потоков. Например, при запуске команды

```
$ run_long_program >/dev/null 2>S1 &
```

сообщения об ошибках будут также выводиться в файл **/dev/null**. Символ '**&**' перед именем потока необходим, чтобы отличить его от файла с именем 1. Заметим, что изменение порядка двух перенаправлений потоков приведет к тому, что сообщения об ошибках будут по-прежнему выводиться на экран. Дело в том, что Shell анализирует командную строку слева направо, таким образом сначала будет осуществлено слияние потоков и

оба будут указывать на терминал пользователя, а затем стандартный поток вывода (1) будет перенаправлен в файл **/dev/null**.

Передача потока вывода одной программы в поток ввода другой осуществляется с помощью конвейера '**|**' (программного канала). Программные каналы часто используются для фильтрации вывода некоторой команды:

```
$ ps — ef | grep тургос
```

позволяет получить информацию о конкретном процессе тургос. Утилита *ps(1)* выводит на экран информацию обо всех процессах в системе, программа *grep(1)* фильтрует этот поток, оставляя лишь строки, в которых присутствует слово тургос<sup>13</sup>.

Можно усложнить задачу и попытаться получить идентификатор процесса тургос. Однако здесь нам не обойтись без других средств системы. В данном случае мы будем использовать интерпретатор *awk(1)*:

```
$ ps -ef | grep тургос | awk '{ print $2 }'
```

Идея заключается в фильтрации второго поля записи о процессе тургос, содержащего идентификатор процесса (см. описание утилиты *ps(1)*).

Иногда возникает необходимость разместить поток ввода вместе с командой. Для этого используется выражение "ввод здесь". Проиллюстрируем его на примере:

```
$ at Dec 31 <<!
cat happy.new.year | elm -s"С Новым Годом"
Congratulations@everybody.ru
'
```

По определению, команда *at(1)* устанавливает вызов команды, полученной ею со стандартного ввода (клавиатуры терминала), на определенное время (в данном случае — на 31 декабря каждого года). С помощью выражения "ввод здесь" мы явно задали вид этой команды, точнее комплекса команд: *cat(1)* передает текст поздравления программе *elm(1)*, отвечающей за отправление сообщения электронной почты.

## Команды, функции и программы

Все команды, которые вводятся в строке приглашения shell, относятся к одной из следующих категорий:

<sup>13</sup> Более правильно было бы записать:

```
$ ps -ef | grep тургос | grep -v grep
```

Дело в том, что в списке, созданном командой *ps*, будут две строки, содержащие слово тургос: собственно строка процесса тургос и строка процесса *grep(1)* с параметром тургос (*ps -ef* распечатывает имя программы, породившей процесс, вместе со всеми параметрами).

- встроенные функции
- функции shell, определенные пользователем
- внешние программы и утилиты

Непосредственное отношение к shell имеют только первые две категории, а программы и утилиты являются обычными исполняемыми файлами.

Запуск встроенной функции не требует порождения нового процесса, поскольку эта функция реализована в самой программе shell (например, **/bin/sh**). Соответственно, встроенные функции shell выполняются быстрее всего. Рассмотрим важнейшие встроенные функции shell.

:	Пустая команда. Код возврата всегда 0 (успех).
	Пустая команда удобна для создания бесконечных циклов, например:
while :	
do	
	done
	Текущий командный интерпретатор выполняет команды, указанные в файле <b>runme</b> . При этом не происходит порождения нового shell, как в случае запуска на выполнение <b>runme</b> . Например, использование в скрипте команды <code>. /usr/bin/include_script</code> выполнит команды файла <b>include_script</b> , как если бы они являлись частью текущего скрипта.
. runme	
break [л]	<b>Производит выход из цикла <i>for</i> или <i>while</i>. Если параметр <i>л</i> указан, происходит выход из <i>n</i> вложенных циклов</b>
ps -ef   awk '{ print \$1 " " \$2}'	
while read uid pid	
do	
if [ \$pid -eq \$PID]	
then	
echo pid=\$pid user=\$uid	
break	
fi	
done	
cd [dir]	Осуществляет переход в каталог <b>dir</b> . Если параметр не указан, происходит переход в домашний каталог (\$HOME)
echo [string]	Строка <b>string</b> выводится на стандартное устройство вывода (терминал)
exec runme	Выполняет программу <b>runme</b> , заменяя ею текущий командный интерпретатор. Например, если в login shell (командном интерпретаторе, запускаемом при регистрации пользователя в системе) мы вызовем exec ls, то после вывода имен файлов текущего каталога произойдет завершение работы в системе

j obs

<code>trap command sig1</code>	Определяет команду <i>command</i> , которая будет выполнена при получении сигналов, указанных в качестве аргументов <i>sig</i> . См. раздел "Сигналы" ранее в этой главе
<code>type name</code>	Показывает, как <i>name</i> будет интерпретироваться командным интерпретатором
<code>ulimit</code>	Выводит или устанавливает значение пределов, ограничивающих использование задачей системных ресурсов (времени процессора, памяти, дискового пространства). Ограничения будут рассматриваться в главе 2
<code>umask nnn</code>	Устанавливает маску прав доступа для вновь создаваемых файлов равной <i>nnn</i>
<code>unset var1 var2 .</code>	Удаляет переменные, указанные в качестве аргументов, из списка определенных переменных командного интерпретатора. Некоторые переменные, например PATH, PS1, PS2, не могут быть удалены
<code>wait pid</code>	Ожидает завершения выполнения процесса с идентификатором <i>pid</i> и возвращает его код возврата

Пользователь может определить функцию командного интерпретатора и использовать ее как встроенную функцию shell. С другой стороны, функции мало отличаются от скриптов, включая синтаксис и передачу аргументов. Однако являясь частью shell, функции работают быстрее.

Синтаксис функции имеет следующий вид:

```
function()
{
  command1
  command2
}
```

Как можно заметить, телом функции является обычный скрипт shell.

В качестве примера приведем функцию `mcd`, позволяющую отобразить в приглашении shell имя текущего каталога.

```
mcd ()
{
  cd $*
  PS1=`pwd`
}
```

### Подстановки, выполняемые командным интерпретатором

Прежде чем выполнить команду, указанную либо в командной строке, либо в скрипте, командный интерпретатор производит определенную последовательность действий:

1. Анализирует синтаксис команды. В случае, если обнаружена синтаксическая ошибка, выводится соответствующее сообщение. Естествен-

но, shell анализирует командную строку в соответствии с синтаксисом собственного языка, а не семантику вызова конкретной команды, например, наличие тех или иных аргументов.

2. Производит подстановки, а именно:

- Заменяет все указанные переменные их значениями. Например, если значение переменной `var` равно `/usr/bin`, то при вызове команды `find $var -name sh -print` переменная `$var` будет заменена ее значением. Другими словами, фактический запуск команды будет иметь вид:

```
find /usr/bin -name sh -print
```

- Формирует списки файлов, заменяя шаблоны. При этом производится подстановка следующих шаблонов:
  - \* — соответствует любому имени файла (или его части), кроме начинающихся с символа '.',
  - [abc] — соответствует любому символу из перечисленных (а или b или c),
  - ? — соответствует любому одиночному символу.

3. Делает соответствующие назначения потоков ввода/вывода. Если в строке присутствуют символы перенаправления (>, <, >>, <<, |), shell производит соответствующее перенаправление потоков. Программный интерфейс ввода/вывода мы рассмотрим в разделе "Работа с файлами" следующей главы.

4. Выполняет команду, передавая ей аргументы с выполненными подстановками. При этом:

- Если команда является функцией, определенной пользователем, вызывается функция.
- В противном случае, если команда является встроенной командой shell, запускается встроенная команда.
- В противном случае производится поиск программы в каталогах, указанных переменной `$PATH`, если имя команды задано без пути. Если имя команды задано явно, т. е. содержит элементы пути (относительный или абсолютный путь), производится запуск программы. В случае, если программа не найдена, выводится сообщение об ошибке.

Описанные подстановки, выполняемые интерпретатором, следует иметь в виду при запуске команд. Например, запуск команды `rm` приведет к удалению всех файлов данного каталога:

<code>\$ ls</code>	Вывести список файлов каталога
<code>a.out client client.c</code>	
<code>server server.c shmem.h</code>	
<code>\$ rm *</code>	Удалить файлы
<code>\$ ls</code>	
<code>\$</code>	Каталог пуст

Команда *rm(l)* без колебаний выполнит свою функцию, поскольку в качестве аргументов она получит обычный список файлов. Замену символа '\*' на список всех файлов каталога произведет shell, и *rm(l)* трудно догадаться, что вы собираетесь удалить все файлы. Реальный же вызов *rm(l)* будет иметь вид:

```
rm a.out client client.c server server.c shmem.h
```

Точно так же запускаемые программы ничего не знают о перенаправлении потоков ввода/вывода, произведенных командным интерпретатором. Напомним, что перенаправление ввода/вывода возможно лишь для стандартных потоков ввода, вывода и сообщений об ошибках. Впрочем, большинство утилит UNIX используют только стандартные потоки.

### Запуск команд

Как уже говорилось, запускаемые команды могут являться либо функциями, определенными пользователем, либо встроенными командами интерпретатора, либо исполняемыми файлами — прикладными программами и утилитами. В любом случае, синтаксис их вызова одинаков.

Если необходимо запустить сразу *несколько* команд, это можно сделать в одной строке, разделив команды символом ';'• Например:

```
$ pwd; date
/home/andy
Apr 18 1997 21:07
```

Заметим, что команды будут выполнены последовательно: сначала выполнится команда *pwd(l)*, которая выведет имя текущего каталога, а затем *date(l)*, которая покажет дату и время.

Можно запустить программу в *фоновом режиме*. В этом случае shell не будет ожидать завершения выполнения программы, а сразу выведет приглашение, и вы сможете продолжить работу в командном интерпретаторе. Для этого строку команды необходимо завершить символом '&':

```
$ find -name myfile.txt.1 -print >/tmp/myfile.list 2>/dev/null &
$
```

Пока утилита *find(l)* производит поиск файла с именем **myfile.txt.1**, сканируя файловую систему, вы сможете выполнить еще массу полезных дел, например, отправить почту или распечатать документ на принтере. Мы вернемся к этой схеме запуска программ далее в этой главе при обсуждении системы управления заданиями.

Наконец, командный интерпретатор предоставляет возможность *условного запуска* команд. Например, если необходимо выполнить команду только в случае успешного завершения предыдущей, следует воспользоваться следующей синтаксической конструкцией:

```
cmd1 && cmd2
```

В качестве примера рассмотрим поиск имени пользователя в файле паролей, и в случае успеха — поиск его имени в файле групп:

```
$ grep sergey /etc/passwd && grep sergey /etc/group
```

Успехом считается нулевой код возврата программы, неудачей — все другие значения.

Можно назначить выполнение команды только в случае неудачного завершения предыдущей. Для этого команды следует разделить двумя символами '|':

```
$ cmd1 \| echo Команда завершилась неудачно
```

Приведенный синтаксис является упрощенной формой условного выражения. Командный интерпретатор имеет гораздо более широкие возможности проверки тех или иных условий, которые мы рассмотрим в следующем разделе.

## Условные выражения

Язык Bourne shell позволяет осуществлять ветвление программы, предостав员я оператор *if*. Приведем синтаксис этого оператора:

```
if условие
then
  command1
  command2
fi
```

Команды *command1*, *command2* и т. д. будут выполнены, если истинно *условие*. Условие может генерироваться одной или несколькими командами. По существу, ложность или истинность условия определяется кодом возврата последней выполненной команды. Например:

```
if grep sergey /etc/passwd >/dev/null 2>&1
then
  echo пользователь sergey найден в файле паролей
fi
```

Если слово *sergey* будет найдено программой *grep(1)* в файле паролей (код возврата *grep(1)* равен 0), то будет выведено соответствующее сообщение.

Возможны более сложные формы оператора *if*.

```
set `who -r`          Установим позиционные параметры равными значениям
                      полей вывода программы who(1)
if [ "$9" = "S" ]      Девятое поле вывода — предыдущий уровень выполнения
                      системы; символ 'S' означает однопользовательский режим
then
  echo Система загружается
```

```

elif [ "$7" = "2" ] Седьмое поле – текущий уровень
echo Переход на уровень выполнения 2
else
echo Переход на уровень выполнения 3
fi

```

Данный фрагмент скрипта проверяет уровень выполнения, с которого система совершила переход, и текущий уровень выполнения системы. Соответствующие сообщения выводятся на консоль администратора. В этом фрагменте условие генерируется командой *test*, эквивалентной (и более наглядной) формой которой является "[ ]". Команда *test* является наиболее распространенным способом генерации условия для оператора *if*.

### Команда *test*

Команда *test* имеет следующий синтаксис:

*test выражение*

или

[ *выражение* ]

Команда вычисляет логическое выражение (табл. 1.10) и возвращает 0, если выражение истинно, и 1 в противном случае.

**Таблица 1.10.** Выражения, используемые в команде *test*

#### Выражения с файлами

<b>-s <i>file</i></b>	Размер файла <i>file</i> больше 0
<b>-r <i>file</i></b>	<u>Для файла <i>file</i> разрешен доступ на чтение</u>
<b>-w <i>file</i></b>	<u>Для файла <i>file</i> разрешен доступ на запись</u>
<b>-x <i>file</i></b>	<u>Для файла <i>file</i> разрешено выполнение</u>
<b>-f <i>file</i></b>	Файл <i>file</i> существует и является обычным файлом
<b>-d <i>file</i></b>	Файл <i>file</i> является каталогом
<b>-c <i>file</i></b>	Файл <i>file</i> является специальным файлом символьного устройства
<b>-b <i>file</i></b>	<u>Файл <i>file</i> является специальным файлом блочного устройства</u>
<b>-p <i>file</i></b>	Файл <i>file</i> является поименованным каналом
<b>-u <i>file</i></b>	Файл <i>file</i> имеет установленный флаг SUID
<b>-g <i>file</i></b>	Файл <i>file</i> имеет установленный флаг SGID
<b>-k <i>file</i></b>	Файл <i>file</i> имеет установленный флаг sticky bit

#### Выражения со строками

<b>-z <i>string</i></b>	Строка <i>string</i> имеет нулевую длину
<b>-n <i>string</i></b>	Длина строки <i>string</i> больше 0
<b><i>string1</i> = <i>string2</i></b>	<u>Две строки идентичны</u>
<b><i>string1</i> != <i>string2</i></b>	Две строки различны

Таблица 1.10 (продолжение)

## Сравнение целых чисел

<i>i1</i> – <code>eq</code> <i>i2</i>	<i>i1</i> равно <i>i2</i>
<i>i1</i> – <code>ne</code> <i>i2</i>	<i>i1</i> не равно <i>i2</i>
<i>i1</i> – <code>lt</code> <i>i2</i>	<i>i1</i> строго меньше <i>i2</i>
<i>i1</i> – <code>le</code> <i>i2</i>	<i>i1</i> меньше или равно <i>i2</i>
<i>i1</i> – <code>gt</code> <i>i2</i>	<i>i1</i> строго больше <i>i2</i>
<i>i1</i> – <code>ge</code> <i>i2</i>	<i>i1</i> больше или равно <i>i2</i>

Более сложные выражения могут быть образованы с помощью логических операторов:

<u><i>выражение</i></u>	Истинно, если выражение ложно (оператор NOT)
<u><i>выражение1 -a выражение2</i></u>	Истинно, если оба выражения истинны (оператор AND)
<u><i>выражение1 -o выражение2</i></u>	Истинно, если хотя бы одно из выражений истинно (оператор OR)

Приведем несколько примеров использования выражений.

Фрагмент скрипта, используемый при регистрации нового пользователя. Скрипт проверяет наличие в домашнем каталоге инициализационного скрипта `.profile` и в случае его отсутствия копирует шаблон:

```
if [ ! -f $HOME/.profile ]
then
    echo "файла .profile не существует – скопируем шаблон"
    cp /usr/lib/mkuser/sh/profile $HOME/.profile
fi
```

Фрагмент скрипта, проверяющего наличие новой почты в почтовом ящике пользователя

```
if [ -s $MAIL ]
then
    echo "Пришла почта"
fi
```

Фрагмент скрипта инициализации системы — запуска "суперсервера" Internet `inetd(1M)`. Если исполняемый файл `/etc/inetd` существует, он запускается на выполнение.

```
if [ -x /etc/inetd ]
then
    /etc/inetd
    echo "запущен сервер inetd"
fi
```

Фрагмент скрипта, анализирующий ввод пользователя, сохраненный в переменной `ANSW`. Если пользователь ввел 'N' или 'n', скрипт завершает свою работу.

```
if [ "$ANSW" = "N" -o "$ANSW" = "n" ]
then
    exit
fi
```

## Циклы

Язык программирования Bourne shell имеет несколько операторов цикла. Приведем их синтаксис:

1) while условие

```
do
    command1
    command2
```

```
done
```

2) until условие

```
do
    command1
    command2
```

```
done
```

3) for var in список

```
do
    command1
    command2
```

```
done
```

С помощью оператора *while* команды *command1*, *command2* и т. д. будут выполняться, пока условие не станет ложным. Как и в случае с оператором *if*, *условие* генерируется кодом возврата команды, например, *test*.

В случае оператора *until* команды *command1*, *command2* и т. д. будут выполняться, пока *условие* не станет истинным.

Оператор *for* обеспечивает выполнение цикла столько раз, сколько слов в *списке*. При этом переменная *var* последовательно принимает значения, равные словам из списка. Список может формироваться различными способами, например как вывод некоторой команды ('*имя\_команды\_формирующей\_список*') или с помощью шаблонов shell.

В другой форме *for*, когда список отсутствует, переменная *var* принимает значения позиционных параметров, переданных скрипту.

Чтобы наглядно представить себе приведенные операторы, обратимся к конкретным примерам.

Например, скрипт монтирования всех файловых систем */etc/mounall* для системы Solaris 2.5 включает в себя их проверку, исходя из данных, указанных в файле */etc/vfsck*. При этом используется оператор *while*.

```

#
cat /etc/vfsck |
while read special fsckdev mountp fstype fsckpass automnt mntopts
# Построчно считывает записи файла vfsck и присваивает переменным spe-
# cial, fsckdev и т. д. значения соответствующих конфигурационных полей.
do
    case $special in
        '#'* | '') # Игнорируем комментарии
        continue ;;
        '-') tt Игнорируем строки, не требующие действия
        continue ;;
    esac
# Последовательно проверяем файловые системы с помощью утилиты
# /usr/sbin/fsck
/usr/sbin/fsck -m -F $fstype $fsckdev >/dev/null 2>&1

done

```

Скрипт очистки давно не используемых файлов во временных каталогах (обычно он запускается при загрузке системы) использует оператор *for*.

```

for dir in /tmp /usr/tmp /home/tmp
do
    find $dir ! -type d -atime +7 -exec rm {} \;
done

```

При этом удаляются все файлы в указанных каталогах (**/tmp**, **/usr/tmp** и **/home/tmp**), последний доступ к которым осуществлялся более недели назад.

## Селекторы

Оператор *case* предоставляет удобную форму селектора:

```

case слово in
шаблон1)
    command

шаблон2)
    command

*)
    command

esac

```

Значение *слово* сравнивается с шаблонами, начиная с первого. Если совпадение найдено, то выполняются команды соответствующего раздела, который заканчивается двумя символами ';'• Шаблоны допускают наличие масок, которые были рассмотрены нами в разделе "Подстановки, выполняемые ко-

мандным интерпретатором". Раздел с шаблоном '\*' аналогичен разделу *default* в синтаксисе селектора *switch* языка C: если совпадения с другими шаблонами не произошло, то будут выполняться команды раздела '\*'.

В качестве примера использования селектора приведем скрипт запуска и останова системы печати в SCO UNIX.

```
state=$1
set `who -r`
case $state in
  'start')
    if [ $9 = "2" -o $9 = "3" ]
    then
      exit
    fi
    [ -f /usr/lib/lpsched ] && /usr/lib/lpsched
  'stop')
    [ -f /usr/lib/lpshut ] && /usr/lib/lpshut
  *)
    echo "usage $0 start|stop"
esac
```

В случае, когда скрипт вызван с параметром *start*, будет произведен запуск системы печати. Если параметр скрипта — *stop*, то система печати будет остановлена. Запуск скрипта с любым другим параметром приведет к выводу сообщения об ошибке.

## Ввод

Как мы уже видели, присвоение значений переменным может осуществляться явно или с помощью вывода некоторой программы. Команда *read* предоставляет удобный способ присвоить переменным значения, считанные из стандартного потока ввода. Это может быть строка, введенная пользователем или считанная из файла в случае перенаправления потока.

Команда *read* считывает строку из стандартного потока ввода и последовательно присваивает переменным, переданным в качестве параметров, значения слов строки. Если число слов в строке превышает число переменных, то в последней переменной будут сохранены все оставшиеся слова. Продемонстрируем это на простом примере:

### Текст скрипта *test5.sh*:

```
#!/bin/sh
echo "input: "
while read var1 var2 var3
do
  echo var1=$var1
  echo var2=$var2
  echo var3=$var3
  echo "input: "
done
```

### Запуск скрипта

```
$ test5.sh
input: пример работы команды read
var1=пример
var2=работы
var3=команды read
input: еще пример
var1=еще
var2=пример
var3=
input:^D
$
```

В приведенном примере *read* в цикле считывает пользовательский ввод. Цикл завершается, когда достигнут конец файла (что эквивалентно пользовательскому вводу *<Ctrl>+<D>*), поскольку при этом *read* возвращает неудачу (код возврата равен 1) и *while* завершает работу. В первом цикле число введенных слов превышает количество переменных, поэтому значение переменной *var3* состоит из двух слов. Во втором цикле значение *var3* пусто.

### Система управления заданиями

Командный интерпретатор может поддерживать управление заданиями. Для Bourne shell (**/bin/sh**), который мы рассматриваем, систему управления заданиями включает парный ему интерпретатор **/bin/jsh**. В остальном этот интерпретатор имеет те же возможности.

В системе управления заданиями каждая команда (простая или составная), которую пользователь запускает со своего терминала, называется *заданием*. Все задания могут выполняться либо в текущем режиме, либо в фоновом режиме, либо быть приостановлены. Задание в каждом из этих состояний обладает рядом характеристик:

Состояние задания	Характеристики
Выполняется в текущем режиме	Задание может считывать данные и выводить данные на терминал пользователя
Выполняется в фоновом режиме	Заданию запрещен ввод с терминала. Возможность вывода на терминал определяется дополнительными установками
Приостановлено	Задание не выполняется

Каждое задание при запуске получает уникальный идентификатор, называемый *номером задания*, который используется в командах системы управления. Синтаксис номера задания, применяемый в командах:

*% jobid*

где *jobid* может принимать следующие значения:

% или +	Текущее задание — самое последнее запущенное или вновь запущенное задание
	Предыдущее задание (по отношению к текущему)
? строка	Задание, для которого строка присутствует в командной строке запуска
n	Задание с номером n
<i>pref</i>	Задание, на которое можно уникально указать префиксом <i>pref</i> , например, команда <i>ls(1)</i> , запущенная в фоновом режиме, адресуется заданием <i>%ls</i>

Система управления заданиями позволяет использовать следующие дополнительные команды:

bg [%jobid]	Продолжает выполнение остановленного задания в фоновом режиме. Без параметра относится к текущему заданию.
fg [%jobid]	Продолжает выполнение остановленного задания в текущем режиме. Если задание <i>jobid</i> выполнялось в фоновом режиме, команда перемещает его в текущий режим.
jobs [-p   -1] [%jobid ...]	Выводит информацию об остановленных и фоновых заданиях с указанными номерами. Если последний аргумент опущен, выводится информация обо всех остановленных и фоновых заданиях. Приведенные ниже опции изменяют формат вывода:
	-1 Вывести идентификатор группы процессов и рабочий каталог.
	-p Вывести только идентификатор группы процессов.
kill [-signo] %jobid	Обеспечивает те же возможности, что и команда <i>kill(1)</i> , но по отношению к заданиям.
stop %jobid	Останавливает выполнения фонового задания.
wait %jobid	Ожидает завершения выполнения задания <i>jobid</i> и возвращает его код возврата.

Приведенный ниже пример иллюстрирует использование команд управления заданиями и не нуждается в комментариях:

```
$ inf.j &
[1] 9112
$ comm1 &
[2] 9113
$ jobs
[1] - Running      inf.j
[2] + Running      comm1
$ stop %1
$ jobs
[1] - Stopped (signal)  inf.j
[2] + Running      comm1
$ stop %%
$ jobs -1
[1] - 9112 Stopped (signal)      inf.j (wd: /home/andy/SH/JOB)
[2] + 9113 Stopped (signal)      comm1 (wd: /home/andy/SH/JOB)
$ bg %1
[1] inf.j &
$ jobs
[1] + Running      inf.j
[2] - Stopped (signal)  comm1
$ kill %1 %2
$ jobs
```

```
[1] + Done (208)  inf.j
[2] - Done (208)  comml
$
```

## Основные утилиты UNIX

В предыдущих разделах мы использовали некоторые утилиты UNIX. Ниже приводятся краткие характеристики утилит, выпавших из поля нашего зрения. Более подробно с различными утилитами можно познакомиться в электронном справочнике *man(1)*.

### Утилиты для работы с файлами

Поле [opt] содержит конкретные опции каждой утилиты.

**cd [dir]**

Изменяет текущий каталог. При задании без параметра — производит переход в домашний каталог пользователя.

**cmp [opt] file1 file2**  
**diff [opt] file1 file2**

Утилита *cmp(1)* сравнивает два файла, указанных в качестве аргументов. Если файлы одинаковы, никакого сообщения не выводится. В противном случае выводятся данные о первом несоответствии между этими файлами (в данном примере первое различие найдено в 13-м символе 4-й строки):

```
$ cat file1
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
$ cat file2
1
2 3
4 5 6
diff1
7 8 9 10
11 12 13 14 15 diff2
$ cmp file1 file2
file1 file2 differ: char 13, line 4
```

Утилита *diff(1)* также сравнивает два файла и выводит список изменений, которые необходимо внести в содержимое этих файлов для того, чтобы преобразовать первый файл во второй. По существу, вывод утилиты *diff(1)* представляет собой команды редактора *ed(1)*, необходимые для преобразования **file1** в **file2**:

```
$ diff file1 file2
3a4
> diff1
5c6,7
< 11 12 13 14 15
> 11 12 13 14 15 diff2
```

**cp** [opt] *file1 file2*  
**cp** [opt] *file1 ... dir*  
**mv** [opt] *file1 file2*  
**mv** [opt] *file1 ... dir*

Утилита *cp(1)* служит для копирования файлов. При этом создается не жесткая связь, а новый файл:

```
$ cp file1 file2
$ ls -li file1 file2
261425 -rw-r--r-- 1 andy user 49 Dec 24 12:58 file1
261427 -rw-r--r-- 1 andy user 49 Dec 24 13:13 file2
```

Утилита *mv(1)* изменяет имя файла.

Если последний параметр является каталогом, то число аргументов утилит *cp(1)* или *mv(1)* может превышать 2. В этом случае будет производиться копирование или перемещение указанных файлов в каталог.

**rm** [opt] *file1...*  
**rmdir** *dir1...*

Утилиты удаления файлов и каталогов. При этом удаляются только записи имен файлов в соответствующих каталогах, фактическое содержимое файла (методанные и дисковые данные) будет удалено, если число жестких связей для файла станет равным 0.

**ls** [opt] [*file1 file2*]

Без параметров утилита *ls(1)* выводит имена файлов текущего каталога. В качестве параметров можно задать имена каталогов, содержимое которых необходимо вывести, или имена файлов, информацию о которых нужно получить. Опции утилиты позволяют получить список различной информативности и формата.

**ln** [opt] *source target*

Утилита *ln(1)* создает жесткую связь имени *source* с файлом, адресуемым именем *target*. При использовании опции *-s* будет создана символьическая связь.

**mkdir** [-m mode] [-p] *dir1...*

Создать каталог.

**pwd**

Вывести имя текущего каталога.

**fgrep** [opt] <подстрока> *file1...*

Утилиты поиска фрагментов текста в файлах. Могут использоваться в качестве фильтров в программных каналах.

**grep** [opt] <рег\_выражение> *file1.*

Для поиска подстроки в файлах можно использовать самую простую из утилит *fgrep(1)* (fast grep).

**egrep** [opt] <рег\_выражение> *file1.*

Если подстрока поиска содержит пробелы или знаки табуляции, ее необходимо заключить в кавычки. Если подстрока уже содержит кавычки, их надо экранировать, поместив символ '\' непосредственно перед кавычками:

**5 fgrep "рассмотрим в разделе \"Создание процесса\" chap"**

Если вы хотите сделать поиск нечувствительным к заглавным/строчным символам, используйте ключ *-u*. Для поиска строк, не содержащих указанную подстроку, используется ключ *-v*.

Утилиты *grep(1)* и *egrep(1)* позволяют производить более сложный поиск, например, когда вы не уверены в написании искомого слова, или хотите найти слова, расположенные в определенных местах файла. В этом случае

в качестве подстроки поиска указывается *регулярное выражение* (*рег\_выражение*).

Например, чтобы произвести поиск слова "центр" в американском (center) и британском (centre) написании, можно задать следующую команду:

```
$ grep "cent[er]" file
```

или

```
$ grep "cent[er] [er]" file
```

[er] является регулярным выражением, соответствующим либо символу 'е', либо 'r'. Регулярное выражение должно быть заключено в кавычки для предотвращения интерпретации специальных символов командным интерпретатором shell.

Утилиты просмотра содержимого файла.

Команда **cat** *file* выводит содержимое файла *file* на экран терминала. Если у вас есть подозрение, что файл не текстовый, т. е. содержит "непечатные" символы, лучше запустить *cat(1)* с ключом *-v*. В этом случае вывод таких символов (которые, кстати, могут нарушить настройки вашего терминала) будет подавлен.

Если размер файла велик и его содержимое не помещается в терминальном окне, удобнее будет воспользоваться утилитами *rd(1)* и *more(1)*, позволяющими выводить файл порциями.

Посмотреть только начало (первые *n* строк) или конец (последние *n* строк) файла можно с помощью утилит *head(1)* и *tail(1)*, соответственно.

**sort**

Для сортировки строк файла используется утилита *sort(1)*. Например, для сортировки текста в алфавитном порядке необходимо ввести следующую команду:

```
$ sort -d file >sorted_file
```

Вы можете указать номер слова строки, по которому необходимо произвести сортировку (точнее, номер поля записи; по умолчанию записью является строка, а поля разделены пробелами). Например, для сортировки строк файла *file*

```
Андрей Май
Борис Январь
Владимир Март
```

по месяцам, можно использовать команду

```
$ sort -m +1 file
```

в результате получим:

```
Борис Январь
Владимир Март
Андрей Май
```

Опция **-m** определяет сортировку по месяцам (не по алфавиту), опция **+1** указывает, что сортировку необходимо проводить по второму полю каждой строки.

**cut**

Позволяет отфильтровать указанные поля строк файла. Разделитель полей указывается опцией **-d<sep>**. Например, чтобы получить реальные имена пользователей системы (пятое поле файла паролей), можно использовать следующую команду:

```
$ cat /etc/passwd | cut -f5 -d:
```

```
WWW Administrator
Yuri Korenev
Serge Smirnoff
W3 group
Konstantin Fedorov
Andrei Robachevsky
Sergey Petrov
```

**wc file**

Позволяет вывести число строк, слов и символов текста файла.

**find dir [opt]**

Выполняет поиск файла в файловой системе UNIX, начиная с каталога **dir**. Например, для вывода полного имени исполняемого файла командного интерпретатора Bourne shell введите команду:

```
$ find / -name sh -print 2>/dev/null
/usr/bin/sh
/usr/xpg4/bin/sh
/sbin/sh
```

С помощью опции **-name** указывается имя искомого файла, а с помощью опции **-print** — действие (вывести полное имя).

С помощью *find(1)* можно производить поиск файлов по другим критериям, например, размеру, последнему времени модификации и т. д. Например, чтобы найти файлы с именем **core** (образ процесса, создаваемый при неудачном его завершении и используемый в целях отладки), последнее обращение к которым было, скажем, более месяца назад (скорее всего такие файлы не нужны пользователям и только "засоряют" файловую систему), можно задать команду:

```
$ find / -name core -atime +30 -print
/u/local/lib/zircon/lib/core
/u/local/etc/httpd/data/zzmaps/core
/home/amd/WORK/novosti/core
/home/amd/WORK/access/core
/home/guests/snell/core
```

Если вы сторонник жесткого администрирования, то можно применить следующую команду:

**\$ find / -name core -atime +30 -exec rm {} \;**  
которая автоматически удалит все найденные файлы.

**chown user file ...**

Изменяет владельца-пользователя указанных файлов.

**chgrp group file ...**

Изменяет владельца-группу указанных файлов.

**chmod mode file ...**

Изменяет права доступа и дополнительные атрибуты файлов.

**file file1 ...**

Сканирует начало файла и пытается определить его тип. Если это текстовый файл (ASCII), *file(1)* пытается определить его синтаксис (текст, программа на С и т. д.). Если это бинарный файл, то классификация ведется по так называемому *magic number*, определения которого находятся в файле **/etc/magic**.

```
$ file *
nlc-2.2d.tar:          tar archive
report.doc:            ascii text
work:                  directory
runme.c:               c program text
runme:                 ELF 32-bit MSB executable
figure.gif:            data
```

## Утилиты для управления процессами

**nice - [ -]n command**  
**renice new\_nice pid**

Утилита *nice(1)* применяется для запуска программы на выполнение с относительным приоритетом (nice number), отличным от принятого по умолчанию. Например, ввод команды:

**\$ nice -10 big\_program**

приведет к запуску *big\_program* с большим значением nice. В UNIX чем больше значение nice number, тем меньший приоритет имеет процесс. Таким образом, при планировании выполнения процессов вероятность того, что ядро операционной системы выберет именно *big\_program* для запуска, уменьшится. Как следствие, *big\_program* станет выполнять дольше, но будет менее интенсивно потреблять процессорные ресурсы.

Только администратор системы может повысить приоритет процесса (уменьшить значение nice number):

**\$ nice - -10 job1**

Утилита *renice(1)* позволяет изменять приоритет процесса во время его выполнения. Например, команда

**\$ renice 5 1836**

устанавливает значение nice number процесса с идентификатором 1836 равным 5. Как и в случае команды *nice(1)*, увеличить приоритет процесса может только администратор системы.

ps	Утилита <i>ps(1)</i> выводит информацию о существующих процессах. При использовании различных опций она позволяет получить следующую информацию:
F	статус процесса (системный процесс, блокировки в памяти и т. д.)
S	состояние процесса (O — выполняется процессором, S — находится в состоянии сна, R — готов к выполнению, I — создается, Z — зомби)
UID	идентификатор (имя) пользователя — владельца процесса
PID	идентификатор процесса
PPID	идентификатор родительского процесса
PRI	текущий динамический приоритет процесса
NI	значение nice number процесса
TTY	управляющий терминал процесса ('?' — означает отсутствие управляющего терминала)
TIME	суммарное время выполнения процесса процессором
STIME	время создания процесса (может отличаться от времени запуска команды)
COMMAND	имя команды, соответствующей процессу
<b>kill [signo]pid1, pid2...</b>	Посыпает процессам с идентификаторами <i>pid1</i> , <i>pid2</i> и т. д. сигнал <i>signo</i> . Сигнал <i>signo</i> может быть указан как в числовой, так и в символьной форме. Команда <i>kill -l</i> выводит таблицу соответствия между символьными именами сигналов и их числовыми значениями:

### \$ kill -l

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGIOT	7) SIGEMT	8) SIGFPE
9) SIGKILL	10) SIGBUS	11) SIGSEGV	12) SIGSYS
13) SIGPIPE	14) SIGALRM	15) SIGTERM	16) SIGUSR1

Таким образом, следующие две команды эквивалентны:

```
$ kill -9 18793
$ kill -SIGKILL 18793
```

at [opt] время_запуска	Утилита <i>at(1)</i> считывает команды стандартного потока ввода и группирует их в задание <i>at</i> , которое будет выполнено в указанное пользователем время. Для выполнения задания будет запущен командный интерпретатор, в среде которого и будут исполнены команды.
	Например, следующая команда, позволит вам поздравить друга с днем рождения в назначенное время:

```
$ at May 30 <<!
cat birthday.txt | elm -s"С Днем Рождения!"
andy@cool.com
'
```

Вы можете добавить опцию **-т**, и после выполнения задания вам будет отправлено уведомление по электронной почте.

## Об администрировании UNIX

Достаточно открыть оглавление любого "Руководства системного администратора" для UNIX, чтобы оценить то многообразие задач и проблем, с которыми приходится сталкиваться при обслуживании системы:

- Настройка жизненно важных для пользователей подсистем, таких как файловая система, система печати и сетевая поддержка. Каждая из них, в свою очередь, может быть разделена на десятки подзадач.
- Регистрация пользователей. Каждый новый пользователь добавляет "забот" администратору системы, но какой же UNIX без пользователей!
- Постоянный мониторинг системы и борьба с авариями. Причем, как правило, неполадки возникают в самый неподходящий момент и там, где их совсем не ждешь. Здесь от администратора потребуется хорошее знание не только операционной системы, но и аппаратуры, на которой она работает.
- Настройка производительности системы.
- Обучение, наставление, "ссоры" и "примирения" с пользователями операционной системы, которую вы обслуживаете.

В этой книге вы не найдете практического руководства по администрированию системы. Вместо этого в следующих главах мы попытаемся взглянуть на UNIX изнутри, понять как устроена эта система и как она работает. Может быть после этого вы посмотрите на руководства другими глазами, а администрирование системы не сведется к простому заучиванию команд.

В качестве компенсации за отсутствие практического руководства предлагаю вашему вниманию перевод материала, найденный мною на одном из WWW-серверов Internet, в котором приведена забавная классификация системных администраторов.

Можно выделить четыре типа системных администраторов UNIX:

- Технический бандит.** Обычно в прошлом системный программист, вынужденный заниматься системным администрированием. Пишет

скрипты на смеси языков интерпретатора Bourne shell, sed, C, awk, perl и APL.

- **Администратор-фашист.** Обычно это законченный тунеядец (реже — бывшая ведьма-секретарша), вынужденный заниматься системным администрированием.
- **Маньяк.** Стареющий хакер, обнаруживший, что ни Массада, ни Куба не собираются достойно оплачивать его услуги по компьютерному шпионажу, вследствие чего подавшийся в системные администраторы.
- **Идиот.** Полный кретин или старый программист на Коболе, выбранный в системные администраторы комитетом, состоящим из таких же кретинов или старых программистов на Коболе.

Как определить, к какому типу принадлежит ваш системный администратор?

### **Ситуация 1. Нехватка дискового пространства**

**Технический бандит.** Пишет набор скриптов для мониторинга использования дискового пространства, для сопровождения базы данных статистики использования диска, для прогнозирования будущего использования с помощью регрессионного анализа, для выявления пользователей, которые превысили стандартное отклонение от среднего значения и, наконец, для отправления нарушителям почтовых сообщений. Помещает скрипты под управление *cron(1)*. В результате свободное дисковое пространство не увеличивается, поскольку "дисковые обжоры" обычно не читают почту.

**Администратор-фашист.** Помещает правила использования диска в сообщение дня *mtfd*. Активно пользуется квотированием дискового пространства. Не допускает никаких исключений, чем полностью останавливает деятельность разработчиков. Блокирует регистрацию пользователей, превысивших квоту.

**Маньяк:**

```
# cd /home
# rm -rf `du -s * | sort -rn | head -1 | awk '{print $2}'`
```

**Идиот:**

```
# cd /home
# cat `du -s * | sort -rn | head -1 | awk '{ printf "%s/*\n", $2}'` | compress
```

### **Ситуация 2. Избыточная загрузка процессора**

**Технический бандит.** Пишет набор скриптов для мониторинга использования вычислительных ресурсов, для сопровождения базы данных статистики их использования, для выявления процессов, превысивших стандартное значение и для изменения приоритета таких процессов. Помещает скрипты под управление *cron(1)*. В результате понижения приоритета офисной базы данных, предает ее забвению, ставя всю работу на грань срыва к немалой радости поклонников игры в *xtrek*.

**Администратор-фашист.** Помещает правила использования вычислительных ресурсов в сообщение дня motd. Активно пользуется квотированием процессорных ресурсов. Не допускает никаких исключений, чем полностью останавливает деятельность разработчиков к немалой радости поклонников игры в xtrek.

**Маньяк:** `# kill -9 `ps -augxww | sort -rn +8 -9 1 head -1 | awk '{print $2}'``

**Идиот:** `# compress -f `ps -augxww | sort -rn +8 -9 | head -1 | awk '{print $2}'``

### Ситуация 3. Регистрация новых пользователей

**Технический бандит.** Пишет скрипт на языке Perl, создающий домашний каталог пользователя, определяющий непонятное окружение и помещающий записи в файлы **/etc/passwd**, **/etc/shadow** и **/etc/group**. Устанавливает на скрипт бит SUID и обязывает секретаршу обеспечить регистрацию новых пользователей. Поскольку обычно секретарша так и не может разобраться в разнице между *<Enter>* и *<Return>*, ни один новый пользователь не зарегистрирован.

**Администратор-фашист.** Помещает правила регистрации пользователей в сообщение дня motd. Поскольку незарегистрированные пользователи не могут прочитать это сообщение, никто не выполняет бюрократических требований, и, как следствие, ни один новый пользователь не зарегистрирован.

**Маньяк.** "Если ты настолько глуп, что не можешь взломать машину и самостоятельно зарегистрироваться, тебе нечего делать в моей системе. В этом ящике и так слишком много придурков".

**Идиот:** `# cd /home; mkdir "Bob's home directory"`  
`# echo "Bob Simon:gandalf:0:0:::dev/tty:compress -f" > /etc/passwd`

### Ситуация 4. Авария загрузочного диска

**Технический бандит.** Чинит диск. Обычно ему удается восстановить файловую систему прямо из приглашения загрузки. Если это не помогает, запускает микроядро, которое запускает на соседнем компьютере скрипт, копирующий на аварийную машину загрузочный код, переформатирующий диск и инсталлирующий операционную систему. Оставляет скрипт работать до конца уик-энда, а сам отправляется в поход в горы.

**Администратор-фашист.** Начинает расследование аварии. Отказывается исправить аварию до тех пор, пока виновный не найден, и с него не взыскана стоимость сломанного оборудования.

**Маньяк.** Извлекает диск. С помощью кузнечного молота пытается подогнать отдельные пластины. Звонит производителю. Во время установки нового диска и операционной системы наносит оскорблений присланному инженеру.

**Идиот.** Не замечает ничего необычного.

### **Ситуация 5. Слабая производительность сети**

**Технический бандит.** Пишет скрипт для мониторинга сети, переписывает программное обеспечение, чем повышает производительность на 2%. Пожимает плечами, говорит: "Я сделал все, что мог", и отправляется в поход в горы.

**Администратор-фашист.** Помещает правила работы в сети в сообщение дня motd. Звонит в Беркли и в AT&T, приставая к ним, как установить сетевые квоты. Пытается уволить поклонников игры в xtrek.

**Маньяк.** Каждые два часа размыкает кабель Ethernet и ждет тайм-аута на сетевых соединениях.

**Идиот:** # compress -f /dev/en0

### **Ситуация 6. "Глупые" вопросы пользователей**

**Технический бандит.** Отвечает на вопросы в шестнадцатеричном или двоичном виде, иногда по-французски, пока пользователь не уходит.

**Администратор-фашист.** Блокирует вход пользователя в систему, пока тот не представит веские доказательства своей квалификации.

**Маньяк:** # cat >> ~luser/.cshrc  
alias vi 'rm !\*; unalias vi; grep -v Bozo ~/.cshrc > ~/.z;  
mv -f ~/.z ~/.cshrc'  
^D

**Идиот.** Отвечает на все вопросы в меру своего понимания. Приглашает пользователя в группу администрирования системы.

### **Ситуация 7. Установка новой версии операционной системы**

**Технический бандит.** Изучает исходные тексты новой версии и выбирает из них только то, что ему нравится.

**Администратор-фашист.** В первую очередь изучает законодательные акты против производителя, поставляющего программное обеспечение с ошибками.

**Маньяк:** # uptime  
1:33pm up 19 days, 22:49, 167 users, load average: 6.49,  
6.45, 6.31

```
# wall
```

Итак, настало время установки новой версии. Займет несколько часов, и если нам повезет — управимся к 5-00. Мы работаем для вас!

```
^D
```

**Идиот:** # dd if=/dev/rmt8 of=/vmunix

## Ситуация 8. Пользователям необходима электронная телефонная книга

**Технический бандит.** Пишет программу на RDBMS, perl и Smalltalk. Отчаявшиеся пользователи возвращаются к использованию записных книжек.

**Администратор-фашист.** Устанавливает Oracle. Отчаявшиеся пользователи возвращаются к использованию записных книжек.

**Маньяк.** Предлагает пользователям хранить данные в едином сплошном файле и применять *grep(l)* для поиска телефонных номеров.

**Идиот:** % dd ibs=80 if=/dev/rdisk001s7 | grep "Fred"

## Заключение

Эта глава знакомит с пользовательской средой UNIX, а также с основными подсистемами этой операционной системы — файловой подсистемой, подсистемой управления процессами и памятью, и с подсистемой ввода/вывода.

Большое внимание уделено командному интерпретатору shell, и его языку программирования. Это, как вы убедились, достаточно мощный инструмент, который, в частности, используется при администрировании системы и конфигурации процесса инициализации UNIX. В конце главы приведены наиболее распространенные утилиты, которые можно найти в любой версии UNIX.



## **Среда программирования UNIX**

Одной из целей, которые изначально ставились перед разработчиками UNIX, являлось создание удобной среды программирования. Во многом это справедливо и сегодня.

Разговор в данной главе пойдет о программировании в UNIX. Может показаться, что предлагаемый материал интересен лишь разработчикам программного обеспечения. Это не совсем так. Безусловно, разработка программ невозможна без знания интерфейса системных вызовов и без понимания внутренних структур и функций, предоставляемых операционной системой. Однако осмысленное администрирование системы также затруднительно без представления о том, как работает UNIX. Программный интерфейс UNIX позволяет наглядно показать внутренние механизмы этой операционной системы.

В начале главы дана общая характеристика программного интерфейса UNIX и связанной с ним среды разработки; затронуты такие важные темы, как обработка ошибок, различия между системными вызовами и функциями стандартных библиотек, форматы исполняемых файлов и размещение образа программы в памяти; также описано, как происходит запуск и завершение программы с точки зрения программиста.

Следующие два раздела посвящены подробному обсуждению программного интерфейса двух важнейших подсистем операционной системы UNIX: файловой подсистемы и подсистемы управления процессами и памятью. В них рассматриваются важнейшие системные вызовы работы с файлами, функции стандартной библиотеки ввода/вывода, системные вызовы создания процесса, запуска новой программы и управления процессами.

В заключение приводятся два типичных приложения: демон и командный интерпретатор, на примере которых проиллюстрированы темы, затронутые в данной главе.

### **Программный интерфейс UNIX**

#### **Системные вызовы и функции стандартных библиотек**

Все версии UNIX предоставляют строго определенный ограниченный набор входов в ядро операционной системы, через которые прикладные за-

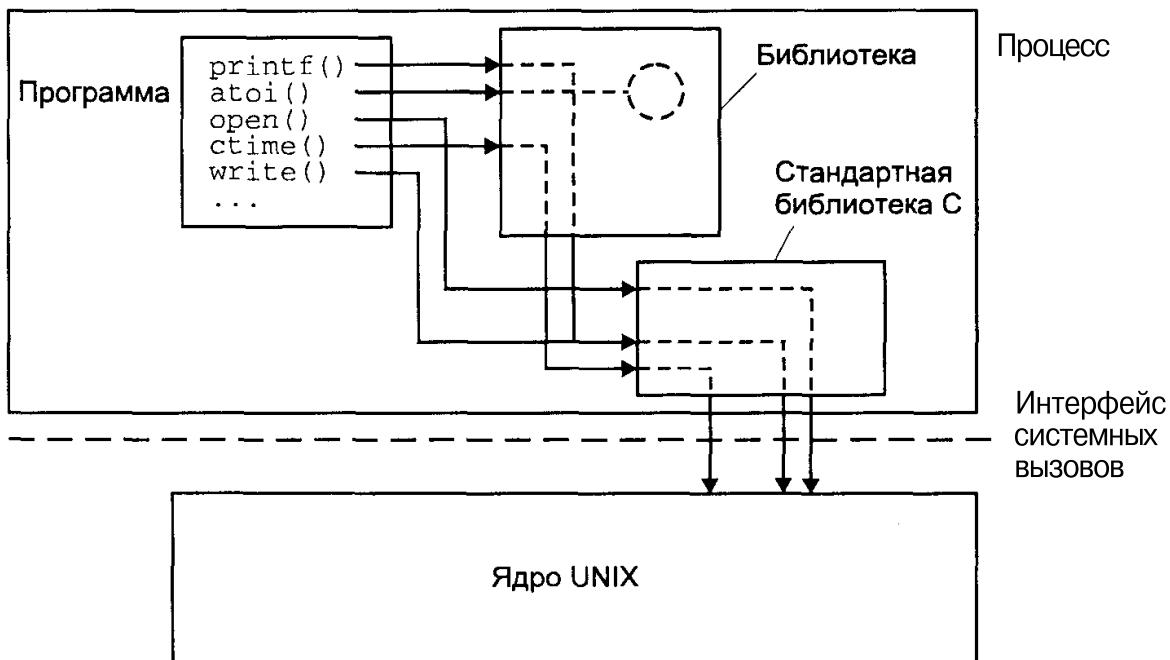
дачи имеют возможность воспользоваться базовыми услугами, предоставляемыми UNIX. Эти точки входа получили название *системных вызовов* (system calls). Системный вызов, таким образом, определяет функцию, выполняемую ядром операционной системы от имени процесса, выполнившего вызов, и является интерфейсом самого низкого уровня взаимодействия прикладных процессов с ядром. Седьмая редакция UNIX включала около 50 системных вызовов, современные версии, например, SVR4, предлагаю более 120.

Системные вызовы обычно документированы в разделе 2 электронного справочника. В среде программирования UNIX они определяются как функции С, независимо от фактической реализации вызова функции ядра операционной системы. В UNIX используется подход, при котором каждый системный вызов имеет соответствующую функцию (или функции) с тем же именем, хранящуюся в стандартной библиотеке языка С (в дальнейшем эти функции будем для простоты называть системными вызовами). Функции библиотеки выполняют необходимое преобразование аргументов и вызывают требуемую процедуру ядра, используя различные приемы. Заметим, что в этом случае библиотечный код выполняет только роль оболочки, в то время как фактические инструкции расположены в ядре операционной системы.

Помимо системных вызовов программисту предлагается большой набор функций общего назначения. Эти функции не являются точками входа в ядро операционной системы, хотя в процессе выполнения многие из них выполняют системные вызовы. Например, функция *printf(3S)* использует системный вызов *write(2)* для записи данных в файл, в то время как функции *strcpy(3C)* (копирование строки) или *atoi(3C)* (преобразование символа в его числовое значение) вообще не прибегают к услугам операционной системы. Функции, о которых идет речь, хранятся в стандартных библиотеках С и наряду с системными вызовами составляют основу среды программирования в UNIX. Подробное описание этих функций приведено в разделе 3 электронного справочника.

Таким образом, часть библиотечных функций является "надстройкой" над системными вызовами, обеспечивающей более удобный способ получения системных услуг. В качестве примера рассмотрим процесс получения текущей даты и времени. Соответствующий системный вызов *time(2)* возвращает время в секундах, прошедшее с момента Epoch: 1 января 1970 года. Дополнительная интерпретация этого значения, такая как преобразование в вид, удобный для восприятия (дата и время) с учетом временной зоны, осуществляется библиотечными функциями (*ctime(3C)*, *localtime(3C)* и т. д.). К этим функциям можно отнести функции библиотеки ввода/вывода, функции распределения памяти, часть функций управления процессами и т. д.

На рис. 2.1 показана схема взаимодействия приложения с ядром операционной системы при использовании системных вызовов и библиотечных функций.



**Рис. 2.1.** Системные вызовы и библиотечные функции

## Обработка ошибок

В предыдущем разделе мы обсудили разницу между системными вызовами и библиотечными функциями. Они также различаются по способу передачи процессу информации об ошибке, произошедшей во время выполнения системного вызова или функции библиотеки.

Обычно в случае возникновения ошибки системные вызовы возвращают `-1` и устанавливают значение переменной `errno`, указывающее причину возникновения ошибки. Так, например, существует более десятка причин завершения вызова `open(2)` с ошибкой, и все они могут быть определены с помощью переменной `errno`. Файл заголовков `<errno.h>` содержит коды ошибок, значения которых может принимать переменная `errno`, с краткими комментариями.

Библиотечные функции, как правило, не устанавливают значение переменной `errno`, а код возврата различен для разных функций. Для уточнения возвращаемого значения библиотечной функции необходимо обратиться к электронному справочнику `man(1)`.

Поскольку базовым способом получения услуг ядра являются системные вызовы, рассмотрим более подробно обработку ошибок в этом случае.

Переменная `errno` определена следующим образом:

```
external int errno;
```

Следует обратить внимание, что значение `errno` не обнуляется следующим нормально завершившимся системным вызовом. Таким образом, значение `errno` имеет смысл только после системного вызова, который завершился с ошибкой.

Стандарт ANSI C определяет две функции, помогающие сообщить причину ошибочной ситуации: `strerror(3C)` и `perror(3C)`.

Функция `strerror(3C)` имеет вид:

```
#include <string.h>
char *strerror(int errnum);
```

Функция принимает в качестве аргумента `errnum` номер ошибки и возвращает указатель на строку, содержащую сообщение о причине ошибочной ситуации.

Функция `perror(3C)` объявлена следующим образом:

```
#include <errno.h>
#include <stdio.h>
void perror(const char *s);
```

Функция выводит в стандартный поток сообщений об ошибках информацию об ошибочной ситуации, основываясь на значении переменной `errno`. Стока `s`, передаваемая функции, предваряет это сообщение и может служить дополнительной информацией, например содержа название функции или программы, в которой произошла ошибка.

Следующий пример иллюстрирует использование этих двух функций:

```
#include <errno.h>
#include <stdio.h>
main(int argc, char *argv[])
{
    fprintf(stderr, "ENOMEM: %s\n", strerror(ENOMEM));
    errno = ENOEXEC;
    perror(argv[0]);
}
```

Запустив программу, мы получим следующий результат на экране:

```
$ a.out
ENOMEM: Not enough space
a.out: Exec format error
```

Эти функции используются, в частности, командным интерпретатором и большинством стандартных утилит UNIX. Например:

```
$ rm does_not_exist
does_not_exist: No such file or directory      ошибка ENOENT
$ pg do_not_read
do_not_read: Permission denied                  ошибка EACCESS
$
```

В табл. 2.1 приведены наиболее общие ошибки системных вызовов, включая сообщения, которые обычно выводят функции *strerror(3C)* и *perror(3C)*, а также их краткое описание.

**Таблица 2.1.** Некоторые ошибки системных вызовов

<b>Код ошибки и сообщение</b>	<b>Описание</b>
E2BIG	Размер списка аргументов, переданных системному вызову <i>exec(2)</i> , плюс размер экспортируемых переменных окружения превышает ARG_MAX байт
EACCES	Попытка доступа к файлу с недостаточными правами для данного класса (определяемого эффективным UID и GID процесса и соответствующими идентификаторами файла)
EAGAIN	Превышен предел использования некоторого ресурса, например, переполнена таблица процессов или пользователь превысил ограничение по количеству процессов с одинаковым UID. Причиной также может являться недостаток памяти или превышение соответствующего ограничения (см. раздел "Ограничения" далее в этой главе)
EALREADY	Попытка операции с неблокируемым объектом, уже обслуживающим некоторую операцию
Operation already in progress	
EBADF	Попытка операции с файловым дескриптором, не адресующим никакой файл; также попытка операции чтения или записи с файловым дескриптором, полученным при открытии файла на запись или чтение, соответственно
Bad file number	
EBADFD	Файловый дескриптор не адресует открытый файл или попытка операции чтения с файловым дескриптором, полученным при открытии файла только на запись
File descriptor in bad state	
EBUSY	Попытка монтирования устройства (файловой системы), которое уже примонтировано; попытка размонтировать файловую систему, имеющую открытые файлы; попытка обращения к недоступным ресурсам (семафоры, блокираторы и т. п.)
Device busy	
ECHILD	Вызов функции <i>wait(2)</i> процессом, не имеющим дочерних процессов или процессов, для которых уже был сделан вызов <i>wait(2)</i>
No child processes	
EDQUOT	Попытка записи в файл, создание каталога или файла при превышении квоты пользователя на дисковые блоки, попытка создания файла при превышении пользовательской квоты на число inode
Disk quota exceeded	

Таблица 2.1 (продолжение)

Код ошибки и сообщение	Описание
EEXIST File exists	Имя существующего файла использовано в недопустимом контексте, например, сделана попытка создания символьской связи с именем уже существующего файла
EFAULT Bad address	Аппаратная ошибка при попытке использования системой аргумента функции, например, в качестве указателя передан недопустимый адрес
EFBIG File too large	Размер файла превысил установленное ограничение RLIMIT_FSIZE или максимально допустимый размер для данной файловой системы (см. раздел "Ограничения" далее в этой главе)
EINPROGRESS Operation now in progress	Попытка длительной операции (например, установление сетевого соединения) для неблокируемого объекта
EINTR Interrupted system call	Получение асинхронного сигнала, например, сигнала SIGINT или SIGQUIT, во время обработки системного вызова. Если выполнение процесса будет продолжено после обработки сигнала, прерванный системный вызов завершится с этой ошибкой
EINVAL Invalid argument	Передача неверного аргумента системному вызову. Например, размонтирование устройства (файловой системы), которое не было примонтировано. Другой пример — передача номера несуществующего сигнала системному вызову <i>kill(2)</i>
EIO I/O error	Ошибка ввода/вывода физического устройства
EISDIR Is a directory	Попытка операции, недопустимой для каталога, например, запись в каталог с помощью вызова <i>write(2)</i>
ELOOP Number of symbolic links encountered during path name traversal exceeds MAXSYMLINKS	При попытке трансляции имени файла было обнаружено недопустимо большое число символьских связей, превышающее значение MAXSYMLINKS
EMFILE Too many open files	Число открытых файлов для процесса превысило максимальное значение OPEN MAX
ENAMETOOLONG File name too long	Длина полного имени файла (включая путь) превысила максимальное значение PATH MAX
ENFILE File table overflow	Переполнение файловой таблицы
ENODEV No such device	Попытка недопустимой операции для устройства. Например, попытка чтения устройства только для записи или операция для несуществующего устройства

**Таблица 2.1 (продолжение)**

<b>Код ошибки и сообщение</b>	<b>Описание</b>
ENOENT No such file or directory	Файл с указанным именем не существует или отсутствует каталог, указанный в полном имени файла
ENOEXEC Exec format error	Попытка запуска на выполнение файла, который имеет права на выполнение, но не является файлом допустимого исполняемого формата
ENOMEM Not enough space	При попытке запуска программы ( <i>exec(2)</i> ) или размещения памяти ( <i>brk(2)</i> ) размер запрашиваемой памяти превысил максимально возможный в системе
ENOMSG No message of desired type	Попытка получения сообщения определенного типа, которого не существует в очереди (см. раздел "Сообщения" в главе 3)
ENOSPC No space left on device	Попытка записи в файл или создания нового каталога при отсутствии свободного места на устройстве (в файловой системе)
ENOSR Out of stream resources	Отсутствие очередей или головных модулей при попытке открытия устройства STREAMS. Это состояние является временным. После освобождения соответствующих ресурсов другими процессами операция может пройти успешно
ENOSTR Not a stream device	Попытка применения операции, определенной для устройств типа STREAMS (например системного вызова <i>putmsg(2)</i> или <i>getmsg(2)</i> ), для устройства другого типа
ENOTDIR Not a directory	В операции, предусматривающей в качестве аргумента имя каталога, было указано имя файла другого типа (например, в пути для полного имени файла)
ENOTTY Inappropriate ioctl for device	Попытка системного вызова <i>ioctl(2)</i> для устройства, которое не является символьным
EPERM Not owner	Попытка модификации файла, способом, разрешенным только владельцу и суперпользователю и запрещенным остальным пользователям. Попытка операции, разрешенной только суперпользователю
EPIPE Broken pipe	Попытка записи в канал (pipe), для которого не существует процесса, принимающего данные. В этой ситуации процессу обычно отправляется соответствующий сигнал. Ошибка возвращается при игнорировании сигнала

**Таблица 2.1 (окончание)**

Код ошибки и сообщение	Описание
EROFS Read-only file system	Попытка модификации файла или каталога для устройства (файловой системы), примонтированного только на чтение
ESRCH No such process	Процесс с указанным PID не существует в системе

## Создание программы

Создание любой программы обычно начинается с базовой идеи (но не всегда), разработки ее блок-схемы (современные программисты часто пропускают этот этап), интерфейса пользователя (весьма ответственный процесс) и написания исходного текста. Далее следуют этапы компиляции и отладки.

В этом разделе рассмотрен процесс создания приложения, написанного на языке С и разработанного для операционной системы UNIX. Предвидя обвинения в архаизме, мы все-таки остановимся на добротном ANSI С и базовой среде разработки UNIX, во-первых, полагая, что старый друг лучше новых двух, а во-вторых потому, что объектом нашего обсуждения все же является UNIX, а не современные средства создания приложений. Заметим также, что язык программирования С является "родным" языком UNIX, поскольку ядро операционной системы написано на этом языке<sup>1</sup>. Это, безусловно, не ограничивает возможности других языков и технологий программирования, которые сегодня, наверное, используются даже чаще, чем обсуждаемый нами традиционный подход.

Опустим также процесс рождения базовой идеи и разработку блок-схем, полагая, что все это уже сделано. Итак, начнем с исходного текста будущей программы.

## Исходный текст

Исходные тексты программы, разработанной для UNIX, по большому счету мало отличаются от текстов приложений, создаваемых для других операционных систем. Можно сказать уверенно, что синтаксис языка определяется не операционной системой. Все, что вам потребуется, это хорошее знание самого языка и особенностей системы UNIX, а именно — ее системных вызовов.

<sup>1</sup> Несмотря на то, что многие современные версии UNIX (особенно коммерческие) поставляются без исходных текстов, основная часть кода ядра в них получена путем компиляции С-модулей.

Во-первых, не забудьте включить в исходный текст необходимые файлы заголовков. Во-вторых, уточните синтаксис вызова библиотечных и системных функций. В-третьих, используйте их по назначению. В-четвертых, не пренебрегайте комментариями.

В этом (за исключением, пожалуй, четвертого совета) вам помогут электронный справочник *man(1)*, ваш опыт, и, надеюсь, эта книга.

## Заголовки

Использование системных функций обычно требует включения в текст программы файлов заголовков, содержащих определения функций — число передаваемых аргументов, типы аргументов и возвращаемого значения. Большинство системных файлов заголовков расположены в каталогах **/usr/include** или **/usr/include/sys**. Если вы планируете использовать малоизвестную системную функцию, будет нeliшним изучить соответствующий раздел электронного справочника *man(1)*. Там же, помимо описания формата функции, возвращаемого значения и особых ситуаций, вы найдете указание, какие файлы заголовков следует включить в программу.

Файлы заголовков включаются в программу с помощью директивы `#include`. При этом, если имя файла заключено в угловые скобки (`<>`), это означает, что поиск файла будет производиться в общепринятых каталогах хранения файлов заголовков. Если же имя файла заголовка заключено в кавычки, то используется явно указанное абсолютное или относительное имя файла.

Например, системный вызов *creat(2)* служащий для создания обычного файла, объявлен в файле `<fcntl.h>` следующим образом:

```
linclude <sys/types.h>
ttinclude <sys/stat.h>
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

Включение в исходный текст прототипа системного вызова *creat(2)* позволяет компилятору произвести дополнительную проверку правильности использования этой функции, а именно — числа аргументов и их типов. Можно заметить, что наряду со стандартными типами языка C, например `char`, для второго аргумента *creat(2)* используется производный тип — `mode_t`. В ранних версиях UNIX большинство системных вызовов использовали стандартные типы, например, *creat(2)* для второго аргумента охотно принимала тип `int`. Производные типы переменных, имеющие окончание `_t`, которые вы в большом количестве встретите при программировании в UNIX, получили название примитивов системных данных. Большинство этих типов определены в файле `<sys/types.h>`, а их назначение заключается в улучшении переносимости написанных программ. Вместо конкретных типов данных, каковыми являются `int`, `char` и т. п., приложению предлага-

ется набор системных типов, гарантированно неизменных в контексте системных вызовов. Другими словами, во всех версиях UNIX сегодня и спустя десять лет, системный вызов *creat(2)* в качестве второго аргумента будет принимать переменную типа *mode\_t*. Фактический размер переменных этого типа может быть разным для различных версий системы, но это отразится в изменении соответствующего файла заголовков и потребует только перекомпиляции вашей программы.

Среда программирования UNIX определяется несколькими стандартами, обсуждавшимися во введении, и может незначительно различаться для разных версий системы. В частности, стандарты ANSI C, POSIX.1 и XPG4, определяют названия и назначения файлов заголовков, приведенных в табл. 2.2.

**Таблица 2.2.** Стандартные файлы заголовков

Файл заголовка	Назначение
<code>&lt;assert.h&gt;</code>	Содержит прототип функции <i>assert(3C)</i> , используемой для диагностики
<code>&lt;cpio.h&gt;</code>	Содержит определения, используемые для файловых архивов <i>cpio(1)</i>
<code>&lt;ctype.h&gt;</code>	Содержит определения символьных типов, а также прототипы функций определения классов символов (ASCII, печатные, цифровые и т. д.) — <i>isascii(3C)</i> , <i>isprint(3C)</i> , <i>isdigit(3C)</i> и т. д.
<code>&lt;dirent.h&gt;</code>	Содержит определения структур данных каталога, а также прототипы функций работы с каталогами <i>opendir(3C)</i> , <i>readdir(3C)</i> и т. д.
<code>&lt;errno.h&gt;</code>	Содержит определения кодов ошибок (см. раздел "Обработка ошибок" в начале главы)
<code>&lt;fcntl.h&gt;</code>	Содержит прототипы системных вызовов <i>fcntl(2)</i> , <i>open(2)</i> и <i>creat(2)</i> , а также определения констант и структур данных, необходимых при работе с файлами
<code>&lt;float.h&gt;</code>	Содержит определения констант, необходимых для операций с плавающей точкой
<code>&lt;ftw.h&gt;</code>	Содержит прототипы функций, используемых для сканирования дерева файловой системы (file tree walk) <i>ftw(3C)</i> и <i>nftw(3C)</i> , а также определения используемых констант
<code>&lt;grp.h&gt;</code>	Содержит прототипы функций и определения структур данных, используемых для работы с группами пользователей: <i>getgrnam(3C)</i> , <i>getgrent(3C)</i> , <i>getgrgid(3C)</i> и т. д.
<code>&lt;langinfo.h&gt;</code>	Содержит определения языковых констант: дни недели, названия месяцев и т. д., а также прототип функции <i>langinfo(3C)</i>

Таблица 2.2 (продолжение)

Файл заголовка	Назначение
<limits.h>	Содержит определения констант, определяющих значения ограничений для данной реализации: минимальные и максимальные значения основных типов данных, максимальное значение файловых связей, максимальная длина имени файла и т. д.
<locale.h>	Содержит определения констант, используемых для создания пользовательской среды, зависящей от языковых и культурных традиций (форматы дат, денежные форматы и т. д.), а также прототип функции <i>setlocale(3C)</i>
<math.h>	Содержит определения математических констант (я, е, $\sqrt{2}$ и т. д.)
<nl_types.h>	Содержит определения для каталогов сообщений (message catalog), а также прототипы функций <i>catopen(3C)</i> и <i>catclose(3C)</i>
<pwd.h>	Содержит определение структуры файла паролей /etc/passwd, а также прототипы функций работы с ним: <i>getpwnam(3C)</i> , <i>getpwent(3C)</i> , <i>getpwuid(3C)</i> и т. д.
<regex.h>	Содержит определения констант и структур данных, используемых в регулярных выражениях, а также прототипы функций для работы с ними: <i>regcomp(3C)</i> , <i>regexec(3C)</i> и т. д.
<search.h>	Содержит определения констант и структур данных, а также прототипы функций, необходимых для поиска: <i>hsearch(3C)</i> , <i>hcreate(3C)</i> , <i>hdestroy(3C)</i>
<setjmp.h>	Содержит прототипы функций перехода <i>setjmp(3C)</i> , <i>sigsetjmp(3C)</i> , <i>longjmp(3C)</i> , <i>siglongjmp(3C)</i> , а также определения связанных с ними структур данных
<signal.h>	Содержит определения констант и прототипы функций, необходимых для работы с сигналами: <i>sigsetops(3C)</i> , <i>sigemptyset(3C)</i> , <i>sigaddset(3C)</i> и т. д. (см. раздел "Сигналы" далее в этой главе)
<stdarg.h>	Содержит определения, необходимые для поддержки списков аргументов переменной длины
<stddef.h>	Содержит стандартные определения (например <i>size_t</i> )
<stdio.h>	Содержит определения стандартной библиотеки ввода/вывода
<stdlib.h>	Содержит определения стандартной библиотеки
<string.h>	Содержит прототипы функций работы со строками <i>string(3C)</i> , <i>strcasestr(3C)</i> , <i>strcat(3C)</i> , <i>strcpy(3C)</i> и т. д.
<tar.h>	Содержит определения, используемые для файловых архивов <i>tar(1)</i>

Таблица 2.2 (продолжение)

Файл заголовка	Назначение
<termios.h>	Содержит определения констант, структур данных и прототипы функций для обработки терминального ввода/вывода
<time.h>	Содержит определения типов, констант и прототипы функций для работы со временем и датами: <i>time(2)</i> , <i>ctime(3C)</i> , <i>localtime(3C)</i> , <i>tzset(3C)</i> , а также определения, относящиеся к таймерам <i>getitimer(2)</i> , <i>setitimer(2)</i> . Таймеры будут рассмотрены в главе 3
<ulimit.h>	Содержит определения констант и прототип системного вызова <i>ulimit(2)</i> для управления ограничениями, накладываемыми на процесс. См. также раздел "Ограничения" далее в этой главе
<unistd.h>	Содержит определения системных символьных констант, а также прототипы большинства системных вызовов
<utime.h>	Содержит определения структур данных и прототип системного вызова <i>utime(2)</i> для работы с временными характеристиками файла (временем доступа и модификации)
<sys/ipc.h>	Содержит определения, относящиеся к системе межпроцессного взаимодействия (IPC), которые рассматриваются в главе 3
<sys/msg.h>	Содержит определения, относящиеся к (сообщениям) подсистеме IPC. См. также раздел "Сообщения" главы 3
<sys/resource.h>	Содержит определения констант и прототипы системных вызовов управления размерами ресурсов, доступных процессу: <i>getrlimit(2)</i> и <i>setrlimit(2)</i> . Более подробно ограничения на ресурсы обсуждаются в разделе "Ограничения" далее в этой главе
<sys/sem.h>	Содержит определения, относящиеся к (семафорам) подсистеме IPC. См. также раздел "Семафоры" главы 3
<sys/shm.h>	Содержит определения, относящиеся к (разделяемой памяти) подсистеме IPC. См. также раздел "Разделяемая память" главы 3
<sys/stat.h>	Содержит определения структур данных и прототипы системных вызовов, необходимых для получения информации о файле: <i>stat(2)</i> , <i>lstat(2)</i> , <i>fstat(2)</i> . Подробнее эти системные вызовы рассмотрены в разделе "Метаданные файла" далее в этой главе
<sys/times.h>	Содержит определения структур данных и прототипа системного вызова <i>times(2)</i> , служащего для получения статистики выполнения процесса (времени выполнения в режиме ядра, задачи и т. д.)
<sys/types.h>	Содержит определения примитивов системных данных

Таблица 2.2 (окончание)

Файл заголовка	Назначение
<sys/utsname.h>	Содержит определения структур данных и прототип системного вызова <i>uname(2)</i> , используемого для получения имен системы (компьютера, операционной системы, версии и т.д.)
<sys/wait.h>	Содержит определения констант и прототипы системных вызовов <i>wait(2)</i> , <i>waitpid(2)</i> , используемых для синхронизации выполнения родственных процессов

## Компиляция

Процедура создания большинства приложений является общей и приведена на рис. 2.2.



Рис. 2.2. Схема компиляции программы

Первой фазой является стадия компиляции, когда файлы с исходными текстами программы, включая файлы заголовков, обрабатываются компилятором *cc(1)*. Параметры компиляции задаются либо с помощью файла **makefile** (или **Makefile**), либо явным указанием необходимых опций компилятора в командной строке. В итоге компилятор создает набор промежуточных объектных файлов. Традиционно имена созданных объектных файлов имеют суффикс ".o".

На следующей стадии эти файлы с помощью редактора связей *ld(1)* связываются друг с другом и с различными библиотеками, включая стандартную библиотеку по умолчанию и библиотеки, указанные пользователем в качестве параметров. При этом редактор связей может выполняться в двух режимах: статическом и динамическом, что задается соответствующими опциями. В статическом, наиболее традиционном режиме связываются все объектные модули и статические библиотеки (их имена имеют суффикс ".a"), производится разрешение всех внешних ссылок модулей и создается единый исполняемый файл, содержащий весь необходимый для выполнения код. Во втором случае, редактор связей по возможности подключает *разделяемые библиотеки* (имена этих библиотек имеют суффикс ".so"). В результате создается исполняемый файл, к которому в процессе запуска на выполнение будут подключены все разделяемые объекты. В обоих случаях по умолчанию создается исполняемый файл с именем **a.out**.

Для достаточно простых задач все фазы автоматически выполняются вызовом команды:

```
$ make prog
```

или эквивалентной ей

```
$ cc -o prog prog.c
```

которые создают исполняемый файл с именем **prog**. В этом случае умалчиваемое имя исполняемого файла (**a.out**) изменено на **prog** с помощью опции **-o**.

Впрочем, указанные стадии можно выполнять и раздельно, с использованием команд *cc(1)* и *ld(1)*. Заметим, что на самом деле команда *cc(1)* является программной оболочкой и компилятора и редактора связей, которую и рекомендуется использовать при создании программ.

Проиллюстрируем процесс создания более сложной программы с помощью конкретных вызовов команд.

```
$ cc -c file1.c file2.c
```

Создадим промежуточные объектные файлы **file1.o** и **file2.o**

```
$ cc -o prog file1.o file2.o -lnsl
```

Создадим исполняемый файл с именем **prog**, используя промежуточные объектные файлы и библиотеку **libnsl.a** или **libnsl.so**

## Форматы исполняемых файлов

Виртуальная память процесса состоит из нескольких *сегментов* или *областей* памяти. Размер, содержимое и расположение сегментов в памяти определяется как самой программой, например, использованием библиотек, размером кода и данных, так и форматом исполняемого файла этой программы. В большинстве современных операционных систем UNIX используются два стандартных формата исполняемых файлов — COFF (Common Object File Format) и ELF (Executable and Linking Format).

Описание форматов исполняемых файлов может показаться лишним, однако представление о них необходимо для описания базовой функциональности ядра операционной системы. В частности, информация, хранящаяся в исполняемых файлах форматов COFF и ELF позволяет ответить на ряд вопросов весьма важных для работы приложения и системы в целом:

- Какие части программы необходимо загрузить в память?
- Как создается область для неинициализированных данных?
- Какие части процесса должны быть сохранены в дисковой области свопинга (специальной области дискового пространства, предназначеннной для временного хранения фрагментов адресного пространства процесса), например, при замещении страниц, а какие могут быть при необходимости считаны из файла, и таким образом не требуют сохранения?
- Где в памяти располагаются инструкции и данные программы?
- Какие библиотеки необходимы для выполнения программы?
- Как связаны исполняемый файл на диске, образ программы в памяти и дисковая область свопинга?

На рис. 2.3 приведена базовая структура памяти для процессов, загруженных из исполняемых файлов форматов COFF и ELF, соответственно. Хотя расположение сегментов различается для этих двух форматов, основные компоненты одни и те же. Оба процесса имеют сегменты кода (text), данных (data), стека (stack). Как видно из рисунка, размер сегментов данных и стека может изменяться, а направление этого изменения определяется форматом исполняемого файла. Размер стека автоматически изменяется операционной системой, в то время как управление размером сегмента данных производится самим приложением. Эти вопросы мы подробно обсудим в разделе "Выделение памяти" далее в этой главе.

Сегмент данных включает инициализированные данные, копируемые в память из соответствующих разделов исполняемого файла, и неинициализированные данные, которые заполняются нулями перед началом выполнения процесса. Неинициализированные данные часто называют сегментом BSS.

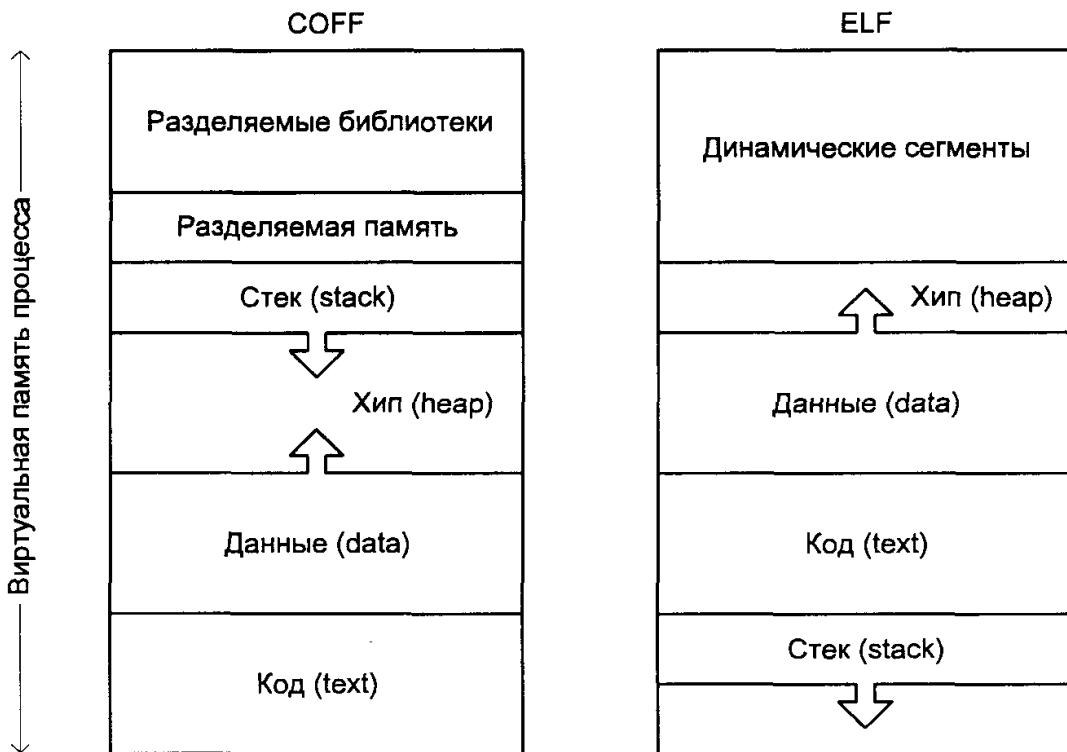


Рис. 2.3. Исполняемые образы программы форматов COFF и ELF

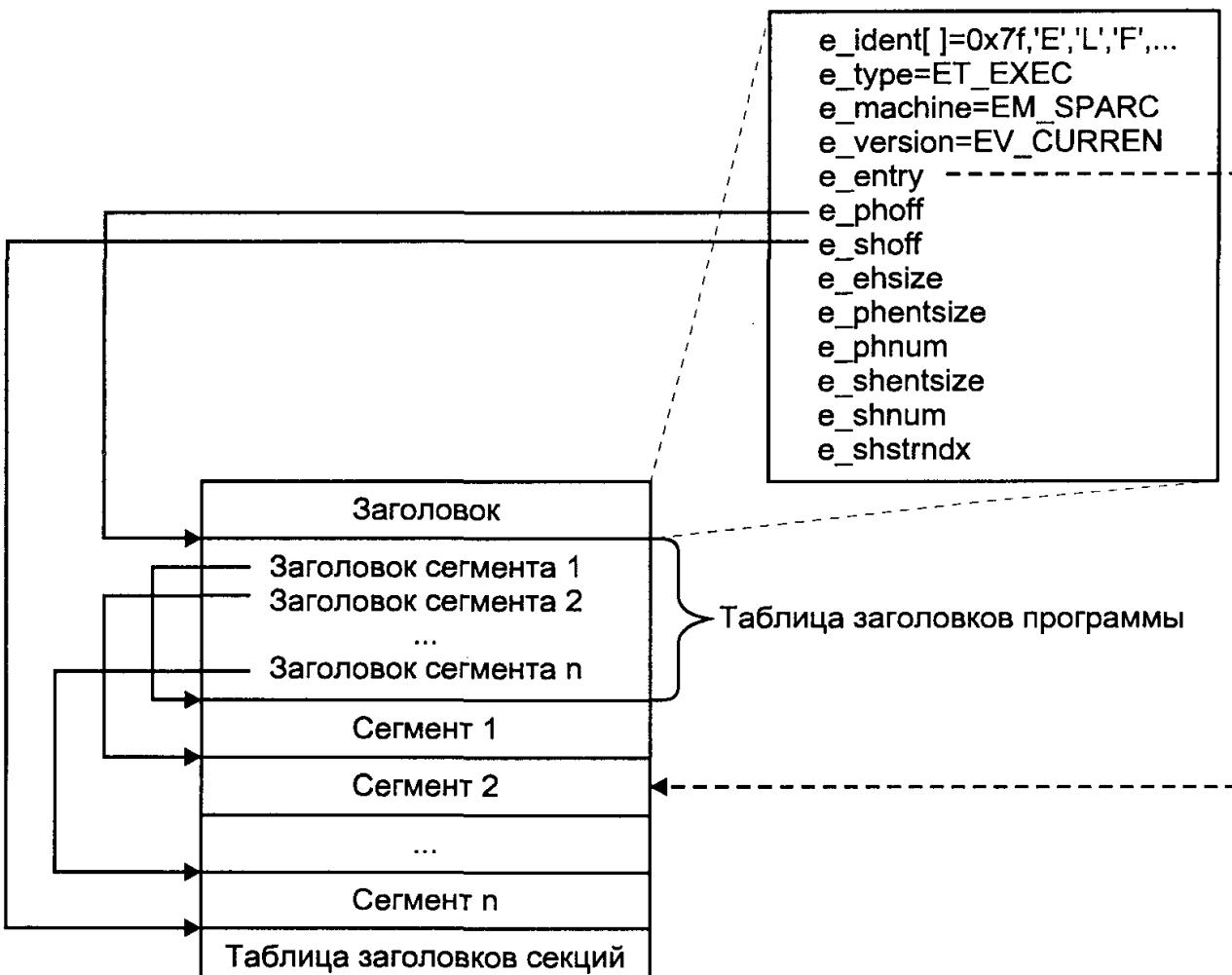
## Формат ELF

Формат ELF имеет файлы нескольких типов, которые до сих пор мы называли по-разному, например, исполняемый файл или объектный файл. Тем не менее стандарт ELF различает следующие типы:

1. *Перемещаемый файл* (relocatable file), хранящий инструкции и данные, которые могут быть связаны с другими объектными файлами. Результатом такого связывания может быть исполняемый файл или разделяемый объектный файл.
2. *Разделяемый объектный файл* (shared object file) также содержит инструкции и данные, но может быть использован двумя способами. В первом случае, он может быть связан с другими перемещаемыми файлами и разделяемыми объектными файлами, в результате чего будет создан новый объектный файл. Во втором случае, при запуске программы на выполнение операционная система может динамически связать его с исполняемым файлом программы, в результате чего будет создан исполняемый образ программы. В последнем случае речь идет о разделяемых библиотеках.
3. *Исполняемый файл* хранит полное описание, позволяющее системе создать образ процесса. Он содержит инструкции, данные, описание необходимых разделяемых объектных файлов, а также необходимую символьную и отладочную информацию.

На рис. 2.4 приведена структура исполняемого файла, с помощью которого операционная система может создать образ программы и запустить программу на выполнение.

Заголовок имеет фиксированное расположение в файле. Остальные компоненты размещаются в соответствии с информацией, хранящейся в заголовке. Таким образом заголовок содержит общее описание структуры файла, расположение отдельных компонентов и их размеры.



**Рис. 2.4.** Структура исполняемого файла в формате ELF

Поскольку заголовок ELF-файла определяет его структуру, рассмотрим его более подробно (табл. 2.4).

**Таблица 2.3.** Поля заголовка ELF-файла

Поле	Описание
<code>e_ident</code> [ ]	Массив байт, каждый из которых определяет некоторую общую характеристику файла: формат файла (ELF), номер версии, архитектуру системы (32-разрядная или 64-разрядная) и т. д.
<code>e_type</code>	Тип файла, поскольку формат ELF поддерживает несколько типов

**Таблица 2.3 (продолжение)**

Поле	Описание
e_machine	Архитектура аппаратной платформы, для которой создан данный файл. В табл. 2.4 приведены возможные значения этого поля
e_version	Номер версии ELF-формата. Обычно определяется как EV_CURRENT (текущая), что означает последнюю версию
e_entry	Виртуальный адрес, по которому системой будет передано управление после загрузки программы (точка входа)
e_phoff	Расположение (смещение от начала файла) таблицы заголовков программы
e_shoff	Расположение таблицы заголовков секций
e_ehsize	Размер заголовка
e_phentsize	Размер каждого заголовка программы
e_phnum	Число заголовков программы
e_shentsize	Размер каждого заголовка сегмента (секции)
e_shnum	Число заголовков сегментов (секций)
e_shstrndx	Расположение сегмента, содержащего таблицу строк

**Таблица 2.4.** Значения поля e\_machine заголовка ELF-файла

Значение	Аппаратная платформа
EM M32	AT&T WE 32100
EM SPARC	Sun SPARC
EM 386	Intel 80386
EM 68K	Motorola 68000
EM 88K	Motorola 88000
EM 486	Intel 80486
EM 860	Intel i860
EM MIPS	MIPS RS3000 Big-Endian
EM MIPS RS3 LE	MIPS RS3000 Little-Endian
EM RS6000	RS6000
EM PA RISC	PA-RISC
EM nCUBE	nCUBE
EM VPP500	Fujitsu VPP500
EM SPARC32PLUS	Sun SPARC 32+

Информация, содержащаяся в таблице заголовков программы, указывает ядру, как создать образ процесса из сегментов. Большинство сегментов

копируются (отображаются) в память и представляют собой соответствующие сегменты процесса при его выполнении, например, сегменты кода или данных.

Каждый заголовок сегмента программы описывает один сегмент и содержит следующую информацию:

- Тип сегмента и действия операционной системы с данным сегментом
- Расположение сегмента в файле
- Стартовый адрес сегмента в виртуальной памяти процесса
- Размер сегмента в файле
- Размер сегмента в памяти
- Флаги доступа к сегменту (запись, чтение, выполнение)

Часть сегментов имеет тип LOAD, предписывающий ядру при запуске программы на выполнение создать соответствующие этим сегментам структуры данных, называемые *областями*, определяющие непрерывные участки виртуальной памяти процесса и связанные с ними атрибуты. Сегмент, расположение которого в ELF-файле указано в соответствующем заголовке программы, будет отображен в созданную область, виртуальный адрес начала которой также указан в заголовке программы. К сегментам такого типа относятся, например, сегменты, содержащие инструкции программы (код) и ее данные. Если размер сегмента меньше размера области, неиспользованное пространство может быть заполнено нулями. Такой механизм, в частности используется при создании неинициализированных данных процесса (BSS). Подробнее об областях мы поговорим в главе 3.

В сегменте типа INTERP хранится программный интерпретатор. Данный тип сегмента используется для программ, которым необходимо динамическое связывание. Суть динамического связывания заключается в том, что отдельные компоненты исполняемого файла (разделяемые объектные файлы) подключаются не на этапе компиляции, а на этапе запуска программы на выполнение. Имя файла, являющегося *динамическим редактором связей*, хранится в данном сегменте. В процессе запуска программы на выполнение ядро создает образ процесса, используя указанный редактор связей. Таким образом, первоначально в память загружается не исходная программа, а динамический редактор связей. На следующем этапе динамический редактор связей совместно с ядром UNIX создают полный образ исполняемого файла. Динамический редактор загружает необходимые разделяемые объектные файлы, имена которых хранятся в отдельных сегментах исходного исполняемого файла, и производит требуемое размещение и связывание. В заключение управление передается исходной программе.

Наконец, завершает файл таблица заголовков *разделов* или *секций* (section). Разделы (секции) определяют разделы файла, используемые для связывания с другими модулями в процессе компиляции или при динамическом связывании. Соответственно, заголовки содержат всю необходимую информацию

для описания этих разделов. Как правило разделы содержат более детальную информацию о сегментах. Так, например, сегмент кода может состоять из нескольких разделов, таких как хэш-таблица для хранения индексов используемых в программе символов, раздел инициализационного кода программы, таблица связывания, используемая динамическим редактором, а также раздел, содержащий собственно инструкции программы.

Мы еще вернемся к формату ELF в главе 3 при обсуждении организации виртуальной памяти процесса, а пока перейдем к следующему распространенному формату — COFF.

### Формат COFF

На рис. 2.5 приведена структура исполняемого файла формата COFF. Исполняемый файл содержит два основных заголовка — заголовок COFF и стандартный заголовок системы UNIX — a.out. Далее следуют заголовки разделов и сами разделы файла, в которых хранятся инструкции и данные программы. Наконец, в файле также хранится символьная информация, необходимая для отладки.

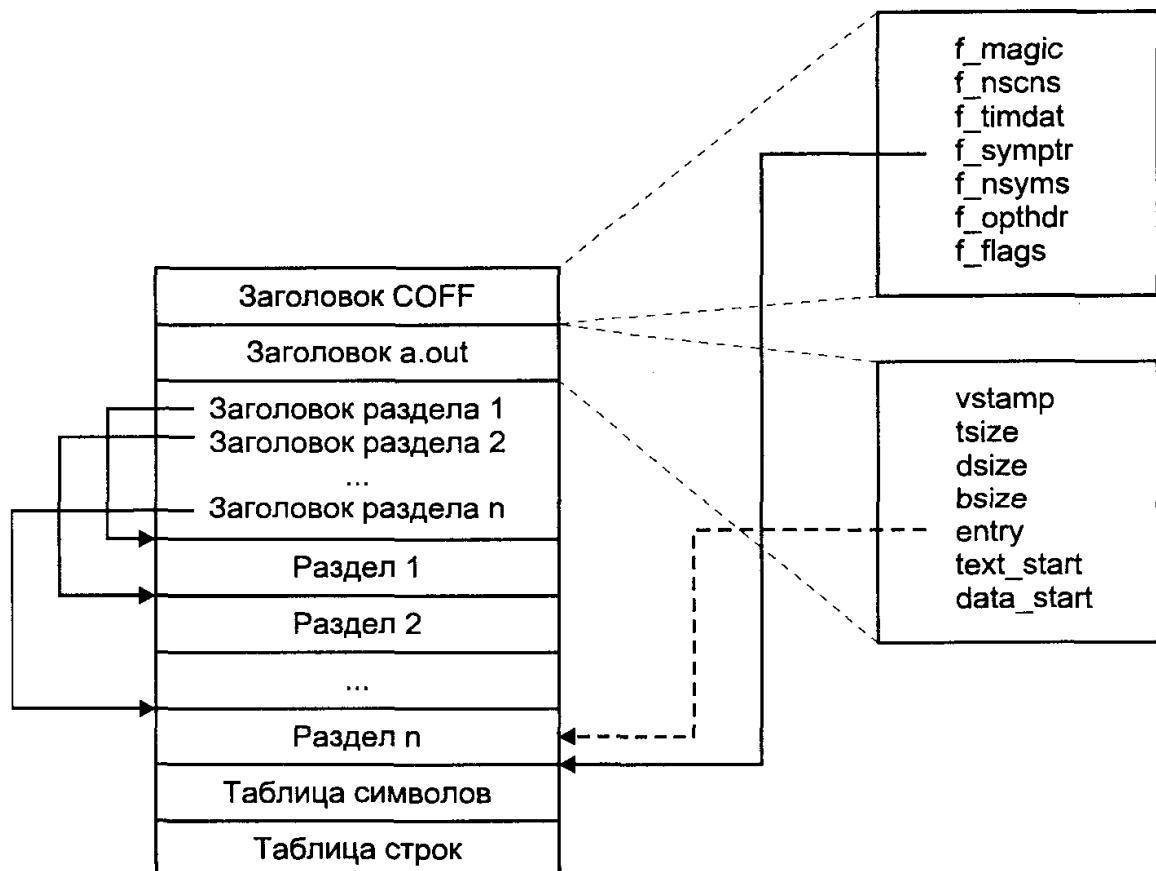


Рис. 2.5. Структура исполняемого файла в формате COFF

В файле находятся только инициализированные данные. Поскольку неинициализированные данные всегда заполняются нулями при загрузке

программы на выполнение, для них необходимо хранить только размер и расположение в памяти.

Символьная информация состоит из *таблицы символов* (symbol table) и *таблицы строк* (string table). В первой таблице хранятся символы, их адреса и типы. Например, мы можем определить, что символ `locptr` является указателем и его виртуальный адрес равен `0x7feh0`. Далее, используя этот адрес, мы можем выяснить значение символа для выполняющегося процесса. Записи таблицы символов имеют фиксированный размер. Если длина символа превышает восемь знаков, его имя хранится во второй таблице — таблице строк. Обычно обе эти таблицы присутствуют в объектных и исполняемых файлах, если они явно не удалены, например, командой `strip(l)`.

Как и в случае ELF-файла, заголовок содержит общую информацию, позволяющую определить местоположение остальных компонентов (табл. 2.5).

**Таблица 2.5.** Поля заголовка COFF-файла

Поле	Описание
<code>f_magic</code>	Аппаратная платформа, для которой создан файл
<code>f_nscns</code>	Количество разделов в файле
<code>f_timdat</code>	Время и дата создания файла
<code>f_symptr</code>	Расположение таблицы символов в файле
<code>f_nsyms</code>	Количество записей в таблице символов
<code>f_opthdr</code>	Размер заголовка <code>a.out</code>
<code>f_flags</code>	Флаги, указывающие на тип файла, присутствие символьной информации и т. д.

Заголовок COFF присутствует в исполняемых файлах, промежуточных объектных файлах и библиотечных архивах. Каждый исполняемый файл кроме заголовка COFF содержит заголовок `a.out`, хранящий информацию, необходимую ядру системы для запуска программы<sup>2</sup> (табл. 2.6).

**Таблица 2.6.** Поля заголовка `a.out`

Поле	Описание
<code>vstamp</code>	Номер версии заголовка
<code>tsize</code>	Размер раздела инструкций (text)
<code>dsize</code>	Размер инициализированных данных (data)
<code>bsize</code>	Размер неинициализированных данных (bss)

<sup>2</sup> В SCO UNIX заголовок `a.out` самого ядра используется программой начальной загрузки `/boot` для запуска ядра и передачи ему управления при инициализации системы.

**Таблица 2.6 (продолжение)**

Поле	Описание
entry	Точка входа программы
text_start	Адрес в начала сегмента инструкций виртуальной памяти
data_start	Адрес в начала сегмента данных виртуальной памяти

Все файлы формата COFF имеют один или более разделов, каждый из которых описывается своим заголовком. В заголовке хранится имя раздела (.text, .data, .bss или любое другое, установленное соответствующей директивой ассемблера), размер раздела, его расположение в файле и виртуальный адрес после запуска программы на выполнение. Заголовки разделов следуют сразу за заголовком файла.

Таблицы символов и строк являются основой системы отладки. *Символом* является любая переменная, имя функции или метка, определенные в программе.

Каждая запись в таблице символов хранит имя символа, его виртуальный адрес, номер раздела, в котором определен символ, тип, класс хранения (автоматический, регистровый и т. д.). Если имя символа занимает больше восьми байт, то оно хранится в таблице строк. В этом случае в поле имени символа указывается смещение имени символа в таблице строк.

С помощью символьной информации можно определить виртуальный адрес некоторого символа. Одним из очевидных применений этой возможности является использование символьной информации в программах-отладчиках. Эта возможность используется некоторыми программами, например, утилитой *ps(l)*, отображающей состояние процессов в системе.

## Выполнение программы в операционной системе UNIX

Выполнение программы начинается с создания в памяти ее образа и связанных с процессом структур ядра операционной системы, инициализации и передаче управления инструкциям программы. Завершение программы ведет к освобождению памяти и соответствующих структур ядра. Образ программы в памяти содержит, как минимум, сегменты инструкций и данных, созданные компилятором, а также стек для хранения *автоматических* переменных при выполнении программы.

## Запуск С-программы

Функция *main()* является первой функцией, определенной пользователем (т. е. явно описанной в исходном тексте программы), которой будет пер-

дано управление после создания соответствующего окружения запускаемой на выполнение программы. Традиционно функция *main()* определяется следующим образом:

```
main(int argc, char *argv[], char *envp[]);
```

Первый аргумент (*argc*) определяет число параметров, переданных программе, включая ее имя.

Указатели на каждый из параметров передаются в массиве *argv* [ ], таким образом, через *argv* [0] адресуется строка, содержащая имя программы, *argv* [1] указывает на первый параметр и т. д. до *argv* [*argc*-1].

Массив *envp* [ ] содержит указатели на переменные окружения, передаваемые программе. Каждая переменная представляет собой строку вида *имя\_переменной*=*значение\_переменной*. Мы уже познакомились с переменными окружения в главе 1, когда обсуждали командный интерпретатор. Сейчас же мы остановимся на их программной "анатомии".

Стандарт ANSI C определяет только два первых аргумента функции *main()* — *argc* и *argv*. Стандарт POSIX.1 определяет также аргумент *envp*, хотя рекомендует передачу окружения программы производить через глобальную переменную *environ*, как это показано на рис. 2.6:

```
extern char *environ;
```

Рекомендуется следовать последнему формату передачи для лучшей переносимости программ на другие платформы UNIX.

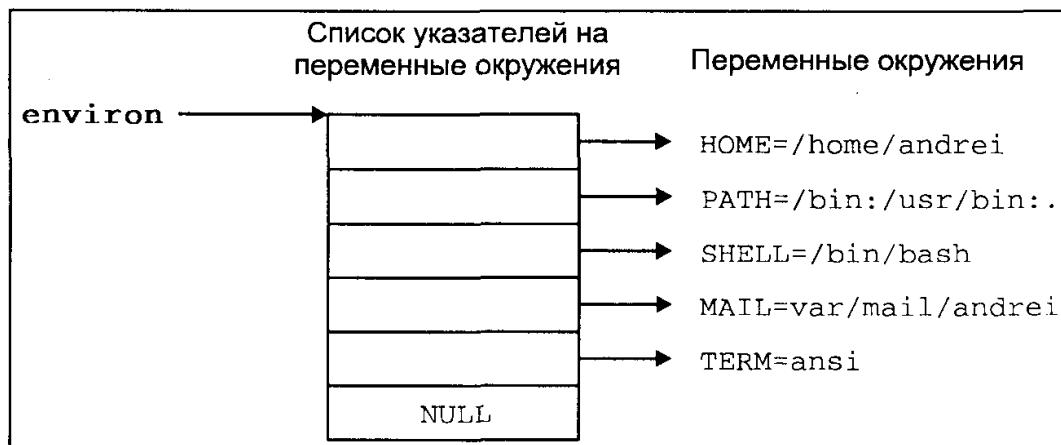


Рис. 2.6. Передача переменных окружения

Приведем пример программы, соответствующую стандарту POSIX.1, которая выводит значения всех аргументов, переданных функции *main()*: число переданных параметров, сами параметры и значения первых десяти переменных окружения.

```
ttinclude <stddef.h>
extern char *environ;
main(int argc, char *argv[])
```

```

int i;
printf("число параметров, переданных
       программе %s равно %d\n", argv[0], argc-1);
for (i=1; i<argc; i++)
    printf("argv[%d] = %s\n", i, argv[i]);
for (i=0; i<8; i++)
    if (environ[i] != NULL)
        printf ("environ[%d] : %s\n",
               i, environ[i]);
}

```

В результате компиляции будет создан исполняемый файл программы (по умолчанию **a.out**). Запустив его, мы увидим следующую информацию:

```

$ a.out first second 3
число параметров, переданных программе a.out равно 3
argv[1] = first
argv[2] = second
argv[3] = 3
environ[0]  LOGNAME=andy
environ[1]  MAIL=/var/mail/andy
environ[2]  LD_LIBRARY_PATH=/usr/openwin/lib:/usr/ucb/lib
environ[3]  PAGER=/usr/bin/pg
environ[4]  TERM=vt100
environ[5]  PATH=/usr/bin:/bin:/etc:/usr/sbin:/sbin:/usr/ccs/
bin:/usr/local/bin
environ[6]  HOME=/home/andy
environ[7]  SHELL=/usr/local/bin/bash

```

Максимальный объем памяти для хранения параметров и переменных окружения программы ограничен величиной **ARG\_MAX**, определенной в файле **<limits.h>**. Это и другие системные ограничения могут быть получены с помощью функции **sysconf(2)**.

Для получения и установки значений конкретных переменных окружения используются две функции: **getenv(3C)** и **putenv(3C)**:

```

ttinclude <stdlib.h>
char *getenv(const char *name);

```

возвращает значение переменной окружения name, а

```

int putenv(const char *string);

```

помещает переменную и ее значение (**var\_name=var\_value**) в окружение программы.

В качестве примера приведем программу, похожую по своей функциональности на предыдущую, которая выборочно выводит значения переменных и устанавливает новые значения по желанию пользователя.

```

ttinclude <stddef.h>
tinclude <stdlib.h>

```

```

#include <stdio.h>
main(int argc, char *argv[])
{
    char *term;
    char buf[200], var[200];
    /*Проверим, определена ли переменная TERM*/
    if ((term= getenv("TERM")) == NULL)
    /*Если переменная не определена, получим от пользователя ее значение и
    поместим переменную в окружение программы*/
    {
        printf("переменная TERM не определена, введите значение: ");
        gets(buf);
        sprintf(var, "TERM=%s",buf);
        putenv(var);
        printf("%s\n", var);
    }
    else
    /*Если переменная TERM определена, предоставим пользователю возможность
    изменить ее значение, после чего поместим ее в окружение процесса*/
    {
        printf("TERM=%s. Change? [N] ", getenv("TERM"));
        gets(buf);
        if (buf[0] =='Y' || buf[0] =='y')
        {
            printf("TERM=");
            gets(buf);
            sprintf(var, "TERM=%s",buf);
            putenv(var);
            printf("new %s\n", var);
        }
    }
}

```

Сначала программа проверяет, определена ли переменная TERM. Если переменная TERM не определена, пользователю предлагается ввести ее значение. Если же переменная TERM определена, пользователю предлагается изменить ее значение, после чего новое значение помещается в окружение программы.

Запуск этой программы приведет к следующим результатам:

```

$ a.out
TERM=ansi. Change? [N]y
TERM=vt100
new TERM=vt100
$
```

К сожалению, введенное значение переменной будет действительно только для данного процесса и порожденных им процессов: если после завершения программы **a.out** вывести значение TERM, то видно, что оно не изменилось:

```

$ echo $TERM
ansi
$
```

Наследование окружения программы мы обсудим в разделе "Создание и управление процессами" далее в этой главе.

Переменные окружения, как и параметры, позволяют передавать программе некоторую информацию. Однако если программа является интерактивной, основную информацию она, скорее всего, будет получать непосредственно от пользователя. В связи с этим встает вопрос: каким образом программа узнает, где находится пользователь, чтобы правильно считывать и выводить информацию? Другими словами, программе необходимо знать, с каким терминальным устройством работает пользователь, запустивший ее.

Обычно при запуске программы на выполнение из командной строки shell автоматически устанавливает для нее три стандартных потока ввода/вывода: для ввода данных, для вывода информации и для вывода сообщений об ошибках. Начальную ассоциацию этих потоков (их файловых дескрипторов) с конкретными устройствами производит *терминальный сервер* (в большинстве систем это процесс *getty(1M)*), который открывает специальный файл устройства, связанный с терминалом пользователя, и получает соответствующие дескрипторы. Эти потоки наследует командный интерпретатор shell и передает их запускаемой программе. При этом shell может изменить стандартные направления (по умолчанию все три потока связаны с терминалом пользователя), если пользователь указал на это с помощью специальных директив перенаправления потока (>, <, >>, <<) см. главу 1, раздел "Пользовательская среда UNIX"). Раздел "Группы и сеансы" внесет окончательную ясность в этот вопрос при описании управляющего терминала.

Такой механизм позволяет программисту не задумываться о местонахождении пользователя, и в то же время обеспечить получение и передачу данных именно запустившему данную программу пользователю.

Завершая разговор о запуске программ, заметим, что при компиляции программы редактор связей устанавливает точку входа в программу, указывающую на библиотечную функцию *\_start()*. Эта функция инициализирует процесс, создавая кадр стека, устанавливая значения переменных и, в конечном итоге, вызывая функцию *main()*.

## Завершение С-программы

Существует несколько способов завершения программы. Основными являются возврат из функции *main()*<sup>3</sup> и вызов функций *exit(2)*, оба приводят к завершению выполнения задачи. Заметим, что процесс может завершиться по не зависящим от него обстоятельствам, например, при получе-

<sup>3</sup> Начальная функция запуска программы на выполнение *\_start()* написана таким образом, что *exit(2)* вызывается автоматически при возврате из функции *main()*. В языке С она имеет следующий вид: `exit(main(argc, argv))`.

ния сигнала, действие по умолчанию для большинства из которых приводит к завершению выполнения процесса<sup>4</sup> (см. раздел "Сигналы" далее в этой главе). В этом случае функция *exit(2)* будет вызвана ядром от имени процесса.

Системный вызов *exit(2)* выглядит следующим образом:

```
#include <unistd.h>
void exit(int status);
```

Аргумент *status*, передаваемый функции *exit(2)*, возвращается родительскому процессу и представляет собой код возврата программы. По соглашению программа возвращает 0 в случае успеха и другую величину в случае неудачи. Значение кода неудачи может иметь дополнительную трактовку, определяемую самой программой. Например, программа *grep(l)*, выполняющая поиск заданных подстрок в файлах, определяет следующие коды возврата:

- 0      совпадение было найдено
- 1      совпадений найдено не было
- 2      синтаксическая ошибка или недоступны файлы поиска

Наличие кода возврата позволяет программам взаимодействовать друг с другом. Например, следующая программа (назовем ее *fail*) может являться условием неудачи и использоваться в соответствующих синтаксических конструкциях *shell*:

```
main()
{
    exit(1);
}
$ fail
$ echo $?          Выведем код возврата программы fail
i
$ fail || echo fail    Конструкция shell, использующая условие неудачи fail
fail
```

Помимо передачи кода возврата, функция *exit(2)* производит ряд действий, в частности выводит буферизированные данные и закрывает потоки ввода/вывода. Альтернативой ей является функция *\_exit(2)*, которая не производит вызовов библиотеки ввода/вывода, а сразу вызывает системную функцию завершения ядра. Более подробно о процедурах завершения процесса см. раздел "Создание и управление процессами".

Задача может зарегистрировать *обработчики выхода* (exit handler), — функции, которые вызываются после вызова *exit(2)*, но до окончательного за-

<sup>4</sup> В английском языке такое завершение выполнения называется более откровенно — "убийство процесса".

вершения процесса. Эти обработчики, вызываемые по принципу LIFO (последний зарегистрированный обработчик будет вызван первым), запускаются только при "добровольном" завершении процесса. Например, при получении процессом сигнала обработчики выхода вызываться не будут. Для обработки таких ситуаций следует использовать специальные функции — *обработчики сигналов* (см. раздел "Сигналы" далее в этой главе).

Обработчики выхода регистрируются с помощью функции *atexit(3C)*:

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Функцией *atexit()* может быть зарегистрировано до 32 обработчиков.

На рис. 2.7 проиллюстрированы возможные варианты запуска и завершения программы, написанной на языке С.

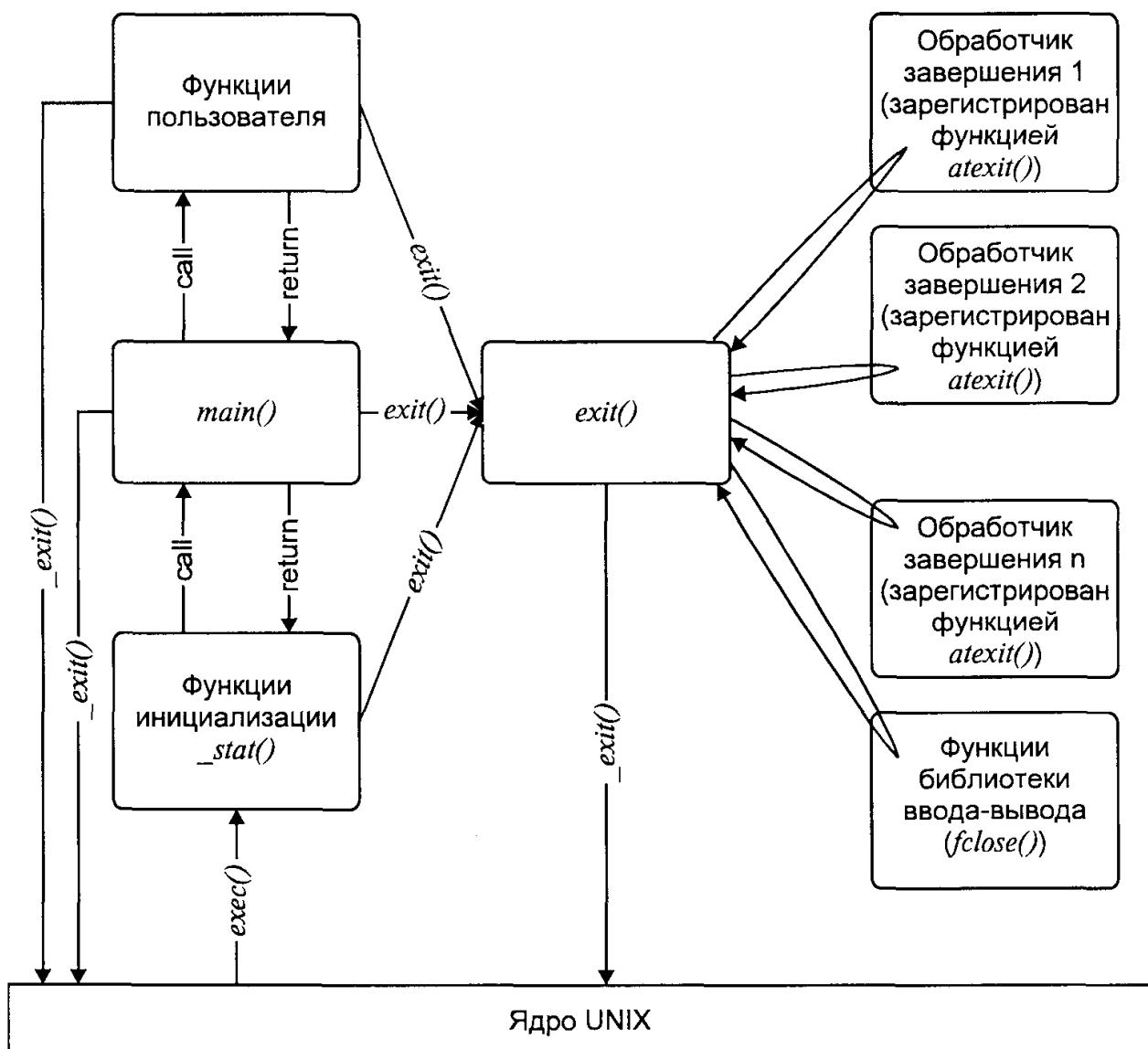


Рис. 2.7. Запуск и завершение С-программы

## Работа с файлами

В среде программирования UNIX существуют два основных интерфейса для файлового ввода/вывода:

1. Интерфейс системных вызовов, предлагающий системные функции низкого уровня, непосредственно взаимодействующие с ядром операционной системы.
2. Стандартная библиотека ввода/вывода, предлагающая функции буферизированного ввода/вывода.

Второй интерфейс является "надстройкой" над интерфейсом системных вызовов, предлагающей более удобный способ работы с файлами.

В следующих разделах будут рассмотрены:

- оба интерфейса, и особенно первый, поскольку именно он представляет набор базовых услуг ядра;
- программный интерфейс управления жесткими и символическими связями файла;
- функции изменения владельцев файла и прав доступа;
- метаданные файла;
- пример программы, выводящей на экран наиболее существенную информацию о файле, подобно тому, как это делает утилита *ls(1)*.

### Основные системные функции для работы с файлами

В табл. 2.7 приведены основные системные функции работы с файлами, являющиеся образами системных вызовов в программе С.

Функции более высокого уровня, предлагаемые стандартной библиотекой ввода/вывода, которые в конечном счете используют описанные здесь системные вызовы, рассматриваются в следующем разделе.

**Таблица 2.7.** Основные системные функции работы с файлами

Системная функция	Описание
<i>open(2)</i>	Служит для получения доступа на чтение и/или запись к указанному файлу. Если файл существует, он открывается, и процессу возвращается файловый дескриптор, адресующий дальнейшие операции с файлом. Если файл не существует, он может быть создан
<i>creat(2)</i>	Служит для создания файла
<i>close(2)</i>	Закрывает файловый дескриптор, связанный с предварительно открытым файлом
<i>dup(2)</i>	Возвращает дубликат файлового дескриптора

Таблица 2.7 (продолжение)

Системная функция	Описание
<i>dup2(2)</i>	Возвращает дубликат файлового дескриптора, но позволяет явно указать его значение
<i>lseek(2)</i>	Устанавливает файловый указатель на определенное место файла. Дальнейшие операции чтения/записи будут производиться, начиная с этого смещения
<i>read(2)</i>	Производит чтение указанного количества байтов из файла
<i>readv(2)</i>	Производит несколько операций чтения указанного количества байтов из файла
<i>write(2)</i>	Производит запись указанного количества байтов в файл
<i>writev(2)</i>	Производит несколько операций записи указанного количества байтов в файл
<i>pipe(2)</i>	Создает коммуникационный канал, возвращая два файловых дескриптора
<i>fcntl(2)</i>	Обеспечивает интерфейс управления открытым файлом

Кратко рассмотрим каждую из этих функций.

### Функция *open(2)*

Открывает указанный файл для чтения или записи и имеет следующий вид:

```
#include <fcntl.h>
int open(const char *path, int oflag, mode_t mode);
```

Первый аргумент (*path*) является указателем на имя файла. Это имя может быть как абсолютным (начинающимся с корневого каталога */*), так и относительным (указанным относительно текущего каталога). Аргумент *oflag* указывает на режим открытия файла и представляет собой побитное объединение флагов, приведенных в табл. 2.8, с помощью операции ИЛИ. Напомним, что если права доступа к файлу не разрешают указанного режима работы с файлом, операция открытия файла будет запрещена, и функция *open(2)* завершится с ошибкой (*errno=EACCESS*). Аргумент *mode*, определяющий права доступа к файлу, используется только при создании файла (как показано в табл. 2.8, функция *open(2)* может использоваться и для создания файла) и рассматривается при описании функции *creat(2)* в разделе "Права доступа" этой главы.

Таблица 2.8. Флаги, определяющие режим открытия файла

Флаг	Описание
<i>O_RDONLY</i>	Открыть файл только для чтения
<i>O_WRONLY</i>	Открыть файл только для записи

Таблица 2.8 (продолжение)

Флаг	Описание
O_RDWR	Открыть файл для чтения и записи
O_APPEND	Производить добавление в файл, т. е. устанавливать файловый указатель на конец файла перед каждой записью в файл
O_CREAT	Если указанный файл уже существует, этот флаг не принимается во внимание. В противном случае, создается файл, атрибуты которого установлены по умолчанию (см. разделы "Владельцы файлов" и "Права доступа к файлу" в главе 1), или с помощью аргумента mode
O_EXCL	Если указан совместно с O_CREAT, то вызов <i>open(2)</i> завершится с ошибкой, если файл уже существует
O_NOCTTY	Если указанный файл представляет собой терминал, не позволяет ему стать управляющим терминалом
O_SYNC	Все записи в файл, а также соответствующие им изменения в метаданных файла будут сохранены на диске до возврата из вызова <i>write(2)</i>
O_TRUNC	Если файл существует и является обычным файлом, его длина будет установлена равной 0
O_NONBLOCK	Изменяет режим выполнения операций <i>read(2)</i> и <i>write(2)</i> для этого файла на неблокируемый. При невозможности произвести запись или чтение, например, если отсутствуют данные, соответствующие вызовы завершатся с ошибкой EAGAIN

Если операция открытия файла закончилась удачно, то будет возвращен файловый дескриптор — указатель на файл, использующийся в последующих операциях чтения, записи и т. д. Значение файлового дескриптора определяется минимальным свободным слотом в таблице дескрипторов процесса. Так, если дескрипторы 0 и 2 уже заняты (указывают на открытые файлы), вызов *open(2)* возвратит значение 1. Это свойство может быть использовано в коде командного интерпретатора при перенаправлении потоков ввода/вывода:

\$ **runme >/home/andrei/run.log**

Фрагмент кода

```
/*Закроем ассоциацию стандартного потока вывода (1) с файлом
(терминалом) */
close (1);
/*Назначим стандартный поток вывода в файл /home/andrei/run.log.
Поскольку файловый дескриптор 1 свободен, мы можем рассчитывать
на его получение.*/
fd = open ("/home/andrei/run.log",
          0 WRONLY I 0 CREATE ] 0 TRUNC);
```

В случае неудачи *open(1)* возвратит -1, а глобальная переменная *errno* будет содержать код ошибки (см. раздел "Обработка ошибок").

Заметим, что только один из флагов *O\_RDONLY*, *O\_WRONLY* и *O\_RDWR* может быть указан в аргументе *oflag*.

Флаг *O\_SYNC* гарантирует, что данные, записанные в файл и связанные с операцией записи изменения метаданных файла, будут сохранены на диске до возврата из функции *write(2)*. Ядро кэширует данные, считываемые или записываемые на дисковое устройство, для ускорения этих операций. Обычно запись данных в файл ограничивается записью в буферный кэш ядра операционной системы, данные из которого впоследствии записываются на диск. По умолчанию возврат из функции *write(2)* происходит после записи в буферный кэш, не дожидаясь записи данных на диск. Более подробно работу буферного кэша мы рассмотрим в главе 4.

Флаг *O\_NOBLOCK* изменяет стандартное поведение функций чтения/записи файла. При указании этого флага возврат из функций *read(2)* и *write(2)* будет происходить немедленно с кодом ошибки и установленным значением *errno* = *EAGAIN*, если ядро не может передать данные при чтении, например, ввиду их отсутствия, или процессу требуется перейти в состояние сна при записи данных.

## Функция *creat(2)*

Функция служит для создания обычного файла или изменения его атрибутов и имеет следующий вид:

```
ttinclude<fcntl.h>
int creat (const char *path, mode_t mode);
```

Как и в случае *open(2)*, аргумент *path* определяет имя файла в файловой системе, а *mode* — устанавливаемые права доступа к файлу. При этом выполняется ряд правил:

- Если идентификатор группы (GID) создаваемого файла не совпадает с эффективным идентификатором группы (EGID) или идентификатором одной из дополнительных групп процесса, бит SGID аргумента *mode* очищается (если он был установлен).
- Очищаются все биты, установленные в маске процесса *umask*.
- Очищается флаг Sticky bit.

Права доступа к файлу обсуждались в главе 1. Более детальная информация приведена в разделе "Права доступа" этой главы.

Если файл уже существует, его длина сокращается до 0, а права доступа и владельцы сохраняются прежними. Вызов *creat(2)* эквивалентен следующему вызову функции *open(2)*:

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

## Функция *close(2)*

Функция *close(2)* разрывает связь между файловым дескриптором и открытым файлом, созданную функциями *creat(2)*, *open(2)*, *dup(2)*, *pipe(2)* или *fcntl(2)*. Функция имеет вид:

```
#include <unistd.h>
int close(int fildes);
```

В случае успеха *close(2)* возвращает нулевое значение, в противном случае возвращается  $-1$ , а значение переменной *errno* указывает на причину неудачи.

Многие программы явно не используют *close(2)* при завершении выполнения. Дело в том, что функция *exit(2)*, вызываемая явно или неявно при завершении выполнения программы, автоматически закрывает открытые файлы.

## Функции *dup(2)* и *dup2(2)*

Функция *dup(2)* используется для дублирования существующего файлового дескриптора:

```
int dup(int fildes);
```

Файловый дескриптор *fildes* должен быть предварительно получен с помощью функций *open(2)*, *creat(2)*, *dup(2)*, *dup2(2)* или *pipe(2)*. В случае успешного завершения функции *dup(2)* возвращается новый файловый дескриптор, свойства которого идентичны свойствам дескриптора *fildes*. Оба указывают на один и тот же файл, одно и то же смещение, начиная с которого будет производиться следующая операция чтения или записи (файловый указатель), и определяют один и тот же режим работы с файлом. Правило размещения нового файлового дескриптора аналогично используемому в функции *open(2)*.

Функция *dup2(2)* делает то же самое, однако позволяет указать номер файлового дескриптора, который требуется получить после дублирования:

```
int dup2(int fildes, int fildes2);
```

Файловый дескриптор, подлежащий дублированию, передается в первом аргументе (*fildes*), а новый дескриптор должен быть равен *fildes2*. Если дескриптор *fildes2* уже занят, сначала выполняется функция *close(fildes2)*.

В качестве примера использования системного вызова *dup2(2)* рассмотрим вариант реализации слияния потоков в командном интерпретаторе *shell*:

```
$ runme >/tmp/file1 2>61
```

Фрагмент кода

```
/*Закроем ассоциацию стандартного потока вывода (1) с файлом
(терминалом) */
```

```
close(1);
/*Назначим стандартный поток вывода в файл /tmp/file1 (fd==1)*/
fd = open("/tmp/file1", O_WRONLY | O_CREATE | O_TRUNC);
/*Выполним слияние потоков*/
dup2(fd, 2);
```

## Функция *lseek(2)*

С файловым дескриптором связан *файловый указатель*, определяющий текущее смещение в файле, начиная с которого будет произведена последующая операция чтения или записи. В свою очередь каждая операция чтения или записи увеличивают значение файлового указателя на число считанных или записанных байт. При открытии файла, файловый указатель устанавливается равным 0 или, если указан флаг O\_APPEND, равным размеру файла. С помощью функции *lseek(2)* можно установить файловый указатель на любое место файла и тем самым обеспечить прямой доступ к любой части файла. Функция имеет следующий вид:

```
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

Интерпретация аргумента *offset* зависит от аргумента *whence*, который может принимать следующие значения:

SEEK_CUR	Указатель смещается на <i>offset</i> байт от текущего положения
SEEK_END	Указатель смещается на <i>offset</i> байт от конца файла
SEEK_SET	Указатель устанавливается равным <i>offset</i>

В случае успеха функция возвращает положительное целое, равное текущему значению файлового указателя.

Относительно системного вызова *lseek(2)* необходимо сделать два замечания. Во-первых, *lseek(2)* не инициирует никакой операции ввода/вывода, лишь изменяя значения файлового указателя в файловой таблице ядра. Во-вторых, смещение, указанное в качестве аргумента *lseek(2)*, может выходить за пределы файла. В этом случае, последующие операции записи приведут к увеличению размера файла и, в то же время, к образованию *дыры* — пространства, формально незаполненного данными. В реальности, дыры заполняются нулями, но могут в ряде случаев привести к неприятным последствиям, с причиной и описанием которых вы сможете ознакомиться в главе 4 при обсуждении внутренней структуры файла.

## Функция *read(2)* и *readv(2)*

Функции *read(2)* и *readv(2)* позволяют считывать данные из файла, на который указывает файловый дескриптор, полученный с помощью функций

*open(2)*, *creat(2)*, *dup(2)*, *dup2(2)*, *pipe(2)* или *fcntl(2)*. Функции имеют следующий вид:

```
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);
#include <sys/types.h>
#include <sys/uio.h>
ssize_t readv(int fildes, struct iovec *iov, int iovcnt);
```

Аргументы, передаваемые функции *read(2)*, указывают, что следует считать *nbyte* байт из файла, связанного с дескриптором *fildes*, начиная с текущего значения файлового указателя. Считанные данные помещаются в буфер приложения, указатель на который передается в аргументе *buf*. После завершения операции значение файлового указателя будет увеличено на *nbyte*.

Функция *readv(2)* позволяет выполнить *iovcnt* последовательных операций чтения за одно обращение к *readv(2)*. Аргумент *iov* указывает на массив структур, каждый элемент которого имеет вид:

<pre>struct {     void *iov_base;     size_t iov_len; } iovec;</pre>	<b>Указатель на начало буфера</b> <b>Размер буфера</b>
--	---

Функция *readv(2)* считывает данные из файла и последовательно размещает их в нескольких буферах, определенных массивом *iov*. Такой характер работы, проиллюстрированный на рис. 2.8, получил название scatter read (от scatter (англ.) — разбрасывать). Общее число считанных байт в нормальной ситуации равно сумме размеров указанных буферов.

## Функции *write(2)* и *writev(2)*

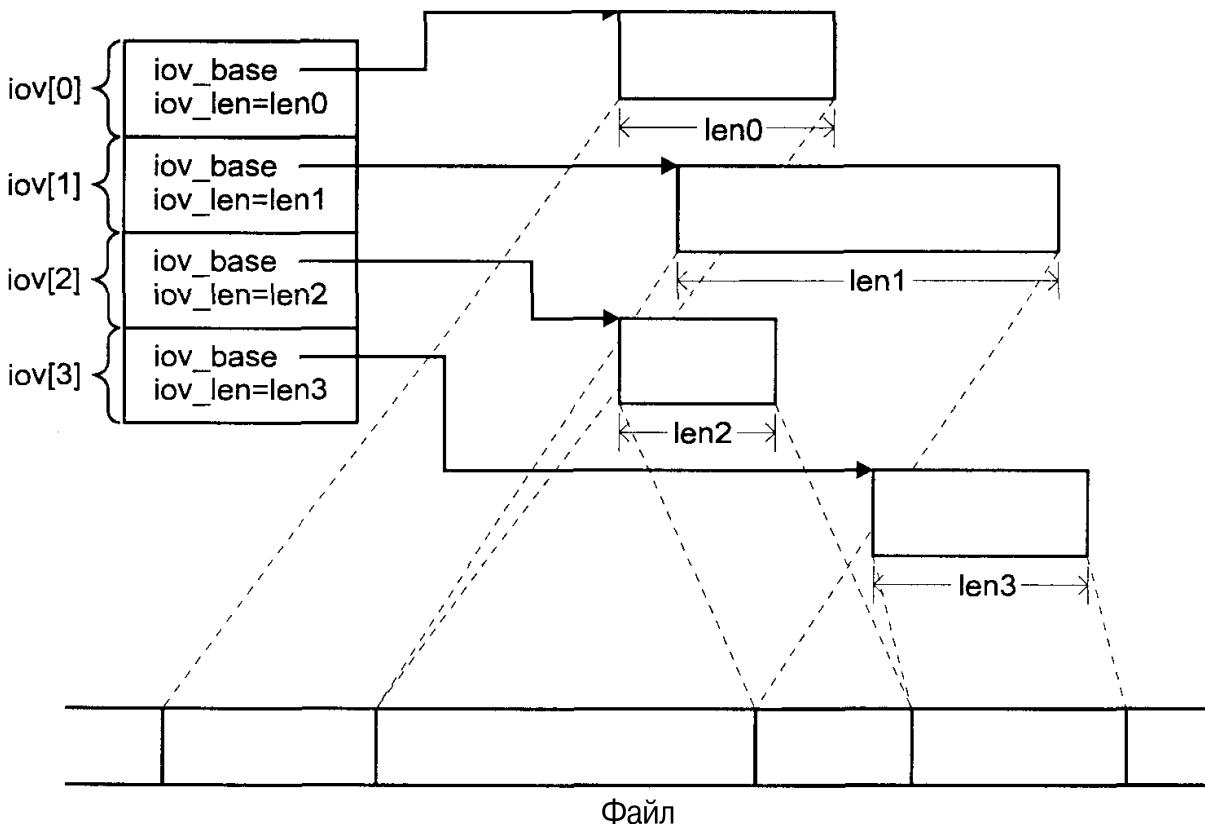
Функции *write(2)* и *writev(2)* очень похожи на функции *read(2)* и *readv(2)*, но используются для записи данных в файл. Функции имеют следующий вид:

```
#include <unistd.h>
ssize_t write(int fildes, void *buf, size_t nbyte);
#include <sys/types.h>
#include <sys/uio.h>
ssize_t writev(int fildes, struct iovec *iov, int iovcnt);
```

Аргументы, передаваемые функции *write(2)*, указывают, что следует записать *nbyte* байт в файл, связанный с дескриптором *fildes*, начиная с текущего значения файлового указателя. Данные для записи находятся в буфере приложения, указанном аргументом *buf*. После завершения операции значение файлового указателя будет увеличено на *nbyte*.

Аналогично функции *readv(2)*, функция *writev(2)* позволяет выполнить *iovcnt* последовательных операций записи за одно обращение к *writev(2)*.

Такая операция ввода/вывода получила название *gather* (собирать), а функции ввода/вывода, использующие набор буферов, — общее название *scatter-gather*.



**Рис. 2.8.** Чтение файла с использованием нескольких буферов

### Функция *pipe(2)*

Функция *pipe(2)* служит для создания одностороннего (симплексного) канала (также называемого анонимным каналом) обмена данными между двумя родственными процессами. Дело в том, что только родственные процессы (например, родительский и дочерний) имеют возможность получить доступ к одному и тому же каналу. Этот аспект станет более понятным в ходе обсуждения в разделе "Создание и управление процессами" далее в этой главе. Функция имеет вид:

```
#include <unistd.h>
int pipe(int fildes[]);
```

Функция возвращает два файловых дескриптора в массиве *fildes* [], причем *fildes[0]* служит для чтения данных из канала, а *fildes[1]* — для записи данных в канал.

Каналы являются одним из способов организации межпроцессного взаимодействия и будут подробно рассмотрены в главе 3. В качестве примера

использования *pipe(2)* можно привести возможность командного интерпретатора — создание программных каналов, рассмотренное в главе 1.

Отметим, что буферизация данных в канале стандартно осуществляется путем выделения дискового пространства в структуре файловой системы. Таким образом, чтение и запись в канал связаны с дисковым вводом/выводом, что, безусловно, сказывается на производительности этого механизма. Современные операционные системы наряду с более совершенными средствами межпроцессного взаимодействия предлагают и более эффективные механизмы каналов. Так, например, SCO UNIX (OpenServer 5.0) обеспечивает работу каналов через специальную файловую систему — HPPS (High Performance Pipe System). С помощью HPPS данные буферизируются в оперативной памяти, что существенно ускоряет операции записи и чтения.

### Функция *fcntl(2)*

После открытия файла и получения ссылки на него в виде файлового дескриптора процесс может производить различные файловые операции. Функция *fcntl(2)* позволяет процессу выполнить ряд действий с файлом, используя его дескриптор, передаваемый в качестве первого аргумента:

```
#include <fcntl.h>
int fcntl(int fildes, int cmd, ...);
```

Функция *fcntl(2)* выполняет действие cmd с файлом, а возможный третий аргумент зависит от конкретного действия:

**F\_DUPFD** Разместить новый файловый дескриптор, значение которого больше или равно значению третьего аргумента. Новый файловый дескриптор будет указывать на тот же открытый файл, что и *fildes*. Действие аналогично вызову функции *dup(2)* или *dup2(2)*:

```
fddup = fcntl(fd, F_DUPFD, fildes2)
```

**F\_GETFD** Возвратить признак сохранения дескриптора при запуске новой программы (выполнении системного вызова *exec(2)*) — флаг *close-on-exec(FD\_CLOEXEC)*. Если флаг установлен, то при вызове *exec(2)* файл, ассоциированный с данным дескриптором, будет закрыт

**F\_SETFD** Установить флаг *close-on-exec* согласно значению, заданному третьим аргументом

**F\_GETFL** Возвратить режим доступа к файлу, ассоциированному с данным дескриптором. Флаги, установленные в возвращаемом значении, полностью соответствуют режимам открытия файла, задаваемым функции *open(2)*. Их значения приведены в табл. 2.8. Рассмотрим пример:

```
oflags = fcntl(fd, F_GETFL, 0);
/* Выделим биты, определяющие режим доступа */
accbits = oflags & O_ACCMODE;
if (accbits == O_RDONLY)
    printf("Файл открыт только для чтения\n");
else if (accbits == O_WRONLY)
    printf("Файл открыт только для записи\n");
else if (accbits == O_RDWR)
    printf("Файл открыт для чтения и записи\n");
```

F_SETFL	Установить режим доступа к файлу согласно значению, переданному в третьем аргументе. Могут быть изменены только флаги <code>0_APPEND</code> , <code>0_NONBLOCK</code> , <code>0_SYNC</code> и <code>0_ASYNC</code> .
F_GETLK	Проверить существование блокирования записи файла. Блокирование записи, подлежащее проверке, описывается структурой <code>flock</code> , указатель на которую передается в качестве третьего аргумента. Если существующие установки не позволяют выполнить блокирование, определенное структурой <code>flock</code> , последняя будет возвращена с описанием текущего блокирования записи. Данная команда не устанавливает блокирование, а служит для проверки его возможности. Более подробно блокирование записей описано в главе 4, в разделе "Блокирование доступа к файлу".
F_SETLK	Установить блокирование записи файла. Структура <code>flock</code> описывает блокирование, и указатель на нее передается в качестве третьего аргумента. При невозможности блокирования <code>fcntl(2)</code> возвращается <b>С ошибкой EACCESS или EAGAIN</b> .
F_SETLKW	Аналогично предыдущему, но при невозможности блокирования по причине уже существующих блокировок, процесс переходит в состояние сна, ожидая, пока последние будут освобождены. Последняя буква <code>W</code> в названии действия означает <code>wait</code> (ждать).

## Стандартная библиотека ввода/вывода

Функции, которые мы только что рассмотрели представляют интерфейс ввода/вывода между приложениями и ядром операционной системы. Хотя их использование напоминает использование библиотечных функций С, по существу они представляют собой лишь "обертки" к функциям ядра UNIX, фактически выполняющим операции ввода/вывода.

Однако программисты редко используют этот интерфейс низкого уровня, предпочитая возможности, предоставляемые стандартной библиотекой ввода/вывода. Функции этой библиотеки обеспечивают *буферизированный ввод/вывод* и более удобный стиль программирования. Для использования функций этой библиотеки в программу должен быть включен файл заголовков `<stdio.h>`. Эти функции входят в стандартную библиотеку С (`libc.so` или `libc.a`), которая, как правило, подключается по умолчанию на этапе связывания.

Вместо использования файлового дескриптора библиотека определяет указатель на специальную структуру данных (структуре `FILE`), называемый *потоком* или *файловым указателем*. Стандартные потоки ввода/вывода обозначаются символическими именами `stdin`, `stdout`, `stderr` соответственно для потоков ввода, вывода и сообщений об ошибках. Они определены следующим образом:

```
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
```

Связь потоков стандартной библиотеки с файловыми дескрипторами приведена в табл. 2.9.

**Таблица 2.9.** Стандартные потоки и их дескрипторы

Файловый дескриптор	Поток (указатель)	Описание
0	stdin	Стандартный ввод
1	stdout	Стандартный вывод
2	stderr	Сообщения об ошибках

**Таблица 2.10.** Наиболее употребительные функции стандартной библиотеки ввода/вывода

Функция	Назначение
<i>fopen(3S)</i>	Открывает файл с указанным именем и возвращает файловый указатель, ассоциированный с данным файлом
<i>fclose(3S)</i>	Закрывает поток, освобождая буферы
<i>fflush(3S)</i>	Очищает буфер потока, открытого на запись
<i>getc(3S)</i>	Считывает символ из потока
<i>putc(3S)</i>	Записывает символ в поток
<i>gets(3S)</i>	Считывает строку из потока
<i>puts(3S)</i>	Записывает строку в поток
<i>fread(3S)</i>	Считывает указанное число байтов из потока (бинарный ввод)
<i>fwrite(3S)</i>	Записывает указанное число байтов в поток (бинарный вывод)
<i>fseek(3S)</i>	Позиционирует указатель в потоке
<i>printf(3S)</i>	Производит форматированный вывод
<i>scanf(3S)</i>	Производит форматированный ввод
<i>fileno(3S)</i>	Возвращает файловый дескриптор данного потока

Выбор между функциями интерфейса системных вызовов и стандартной библиотеки зависит от многих факторов, в частности, степени контроля ввода/вывода, переносимости программы, простоты. Взгляните, например, на следующие эквивалентные строки программы:

```
write (1, "Здравствуй, Мир!\n", 16);
printf ("Здравствуй, Мир!\n");
```

В первой строке сообщение выводится с использованием системной функции *write(2)*, во второй — с помощью библиотечной функции *printf(3S)*. Помимо того, что второй вариант кажется более лаконичным, отметим еще ряд особенностей. В первом варианте пришлось сделать предположение о том, что файловый дескриптор стандартного вывода равен 1, что может оказаться несправедливым для некоторых систем. Также пришлось

явно указать число символов в строке, т. к. *write(2)* не делает никаких предположений о формате вывода, трактуя его как последовательность байтов. В отличие от *write(2)*, *printf(3S)* распознает строки, представляющие собой последовательность символов, заканчивающихся нулем. Функция *printf(3S)* также позволяет отформатировать выводимые данные для представления их в требуемом виде.

Но основным достоинством функций библиотеки является буферизация ввода/вывода, позволяющая минимизировать число системных вызовов *read(2)* и *write(2)*. При открытии файла и создании потока функции библиотеки автоматически размещают необходимые буфера, позволяя приложению не заботиться о них.

Библиотека предоставляет три типа буферизации:

- *Полная буферизация*. В этом случае операция чтения или записи завершается после того, как будет заполнен буфер ввода/вывода. Ввод/вывод для дисковых файлов, как правило, полностью буферизируется. Буфер размещается с помощью функции *malloc(3C)* при первом обращении к потоку для чтения или записи и заполняется системными вызовами *read(2)* или *write(2)*. Это означает, что последующие вызовы *getc(3S)*, *gets(3S)*, *putc(3S)*, *puts(3S)* и т. д. не инициируют обращений к системным функциям, а будут производить чтение или запись из буфера библиотеки. Содержимое буфера очищается (т. е. данные сохраняются на диске) автоматически, либо при вызове функции *fflush(3S)*.
- *Построчная буферизация*. В этом случае библиотека выполняет фактический ввод/вывод (т. е. производит системные вызовы *read(2)* или *write(2)*) построчно при обнаружении конца строки (символа перевода каретки). Такой тип буферизации обычно используется для ассоциированных с терминальными устройствами потоков, которыми, как правило являются стандартные потоки ввода и вывода.
- *Отсутствие буферизации*. В этом случае библиотека не производит никакой буферизации, фактически являясь только программной оболочкой системных вызовов. При этом достигаются минимальные задержки операций чтения и записи, необходимые, например, при выводе сообщений об ошибках. Отсутствие буферизации характерно для стандартного потока вывода сообщений об ошибках.

Характер буферизации может быть изменен с помощью функций:

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

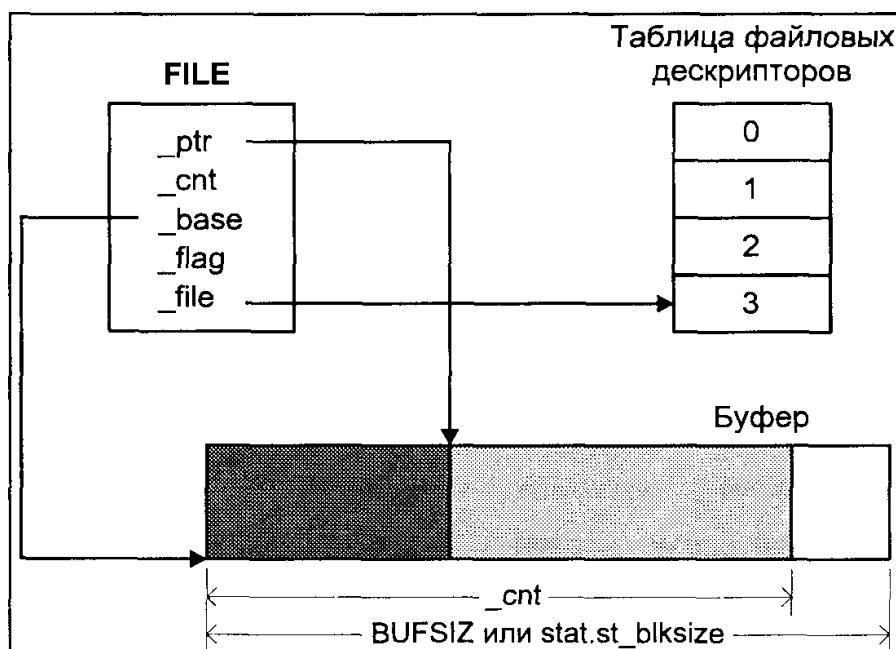
Функция *setbuf(3S)* позволяет включить или отключить буферизацию для потока *stream*. В первом случае *buf* должен указывать на буфер размером *BUFSIZ*, во втором его значение должно быть равно *NULL*.

Функция `setvbuf(3S)` позволяет производить более тонкое управление буферизацией, явно указывая, какой ее тип мы хотим установить. Для этого используется аргумент `type`, который может принимать следующие значения:

<code>_IOFBF</code>	Полная буферизация
<code>_IOLBF</code>	Построчная буферизация
<code>IONBF</code>	Отсутствие буферизации

В случае полной или построчной буферизации аргумент `size` определяет размер буфера, адресованного указателем `buf`.

Каждый поток стандартной библиотеки представлен указателем на структуру `FILE`, показанную на рис. 2.9, в которой хранится указатель на буфер `_base`, указатель на следующий символ, подлежащий чтению или записи `_ptr`, число байт в буфере `_cnt`, указатель на файловый дескриптор `_file`, с которым ассоциирован данный поток, а также флаги состояния потока `_flag`. При создании буфера библиотека выбирает оптимальный размер для данного потока. Обычно этот размер равен значению поля `st_blksize` структуры `stat`, возвращаемой системным вызовом `stat(2)`, рассмотренный в разделе "Метаданные файла" этой главы. Если определить оптимальный размер невозможно, например для каналов или специальных файлов устройств, выбирается стандартное значение `BUFSIZ`, определенное в файле `<stdio.h>`.



**Рис. 2.9.** Структуры данных потока

## Связи

В метаданных каждого файла файловой системы UNIX хранится число связей, определяющее количество имен, которое имеет данный файл. На-

пример, файлы `/etc/init.d/Ip` (или `/etc/Ip`), `etc/rc0.d/K201p`, `/etc/rc2.d/K20Ip` и `/etc/rc2.d/S80Ip` имеют различные имена, но ссылаются на один и тот же физический файл (точнее, метаданные файла) и тем самым обеспечивают доступ к одним и тем же данным. В данном случае число связей файла равно 4. Каждый раз, когда одно из имен файла удаляется, число связей соответственно уменьшается. Когда оно достигнет нуля — данные файла будут удалены. Такой тип связи называется *жесткой*.

Жесткая связь создается с помощью системного вызова `link(2)`:

```
#include <unistd.h>
int link(const char *existing, const char *new);
```

При этом будет образована новая запись каталога с именем `new` и номером `inode` указывающим на метаданные файла `existing`. Также будет увеличено число связей. Этим системным вызовом, в частности, пользуется команда `ln(1)`, рассмотренная в главе 1.

Для удаления жесткой связи используется системный вызов `unlink(2)`:

```
#include <unistd.h>
int unlink(const char *path);
```

Эту функцию вызывает команда `rm(1)` при удалении файла. При этом не обязательно будут удалены данные файла. Заметим, что системный вызов, явно удаляющий данные файла, отсутствует, поскольку у файла может существовать несколько жестких связей, часть из которых может быть недоступна процессу, вызывающему такую функцию (например, одно из имен файла может быть расположено в недоступном каталоге).

В противоположность жестким связям, которые, как отмечалось в главе 1, являются естественным способом адресации данных файла, в UNIX применяются *символические связи*, адресующие не данные файла, а его имя. Например, если файл является символической связью, то в его данных хранится имя файла, данные которого косвенно адресуются.

Символическая связь позволяет косвенно адресовать другой файл файловой системы. Системный вызов `symlink(2)` служит для создания символической связи. Этим вызовом, кстати, пользуется команда `ln -s`.

```
#include <unistd.h>
int symlink(const char *name, const char *symname);
```

После создания символической связи, доступ к целевому файлу `name` может осуществляться с помощью `symname`. При этом, функция `open(2)`, принимая в качестве аргумента имя символической связи, на самом деле открывает целевой файл. Такая особенность называется *следованием символической связи*. Не все системные вызовы обладают этим свойством. Например, системный вызов `unlink(2)`, удаляющий запись в каталоге, действует только на саму символическую связь. В противном случае, мы не имели бы возможности удалить ее. В табл. 2.11 показано, как работают с символическими связями различные системные вызовы.

**Таблица 2.11.** Интерпретация символьской связи различными системными вызовами

Системный вызов	Следует символьской связи	Не следует символьской связи
<i>access(2)</i>	+	
<i>chdir(2)</i>	+	
<i>chmod(2)</i>	+	
<i>chown(2)</i>	+	
<i>lchown(2)</i>		+
<i>creat(2)</i>	+	
<i>exec(2)</i>	+	
<i>link(2)</i>	±	
<i>mkdir(2)</i>	+	
<i>mknod(2)</i>		+
<i>open(2)</i>	+	
<i>readlink(2)</i>		+
<i>rename(2)</i>		+
<i>stat(2)</i>	+	
<i>lstat(2)</i>		+
<i>unlink(2)</i>		+

Для чтения содержимого файла — символьской связи используется системный вызов *readlink(2)*:

```
ttinclude <unistd.h>
int readlink(const char *path, void *buf, size_t bufsiz);
```

Аргумент *path* содержит имя символьской связи. В буфере *buf* размером *bufsiz* возвращается содержимое файла — символьской связи.

Для иллюстрации к вышеприведенным рассуждениям приведем пример программы, которая сначала выводит содержимое символьской связи, а затем — целевого файла, пользуясь в обоих случаях символьским именем:

```
linclude <sys/types.h>
ttinclude<sys/stat.h>
ttinclude <fcntl.h>
#include <stdio.h>
#define BUFSZ 256
/* В качестве аргумента программа принимает
имя символьской связи*/
main(int argc, char *argv[])

```

```

char buf[BUFSZ+1];
int nread, fd;
/*Прочитаем содержимое самой символической связи*/
printf("Читаем символическую связь\n");
nread = readlink(argv[1], buf, BUFSZ);
if (nread < 0)
{
    perror("readlink"); exit(1);
}
/*readlink не завершает строку '\0'*/
buf[nread] = '\0';
printf("Символическая связь:\n %s\n", buf);
/*Теперь прочитаем содержимое целевого файла*/
printf("Читаем целевой файл\n");
fd = open(argv[1], O_RDONLY);
if (fd < 0)
{
    perror("open"); exit(2);
}
nread = read(fd, buf, BUFSIZ);
if (nread < 0)
{
    perror("read"); exit(3);
}
buf[nread] = '\0';
printf("Целевой файл:\n %s\n", buf);
close(fd);
exit(0);
}

```

Перед тем как запустить программу, создадим символическую связь с файлом **unix0.txt**:

```

$ ln -s unix0.txt symlink.txt
$ ls -l
lrwxrwxrwx 1 andy user 10 Jan 6 09:54 symlink.txt -> unix0.txt
-rw-r--r-- 1 andy user 498 Jan 6 09:53 unix0.txt

```

```

$ a.out symlink.txt
Читаем символическую связь
Символическая связь:
unix0.txt
Читаем целевой файл
Целевой файл:

```

Начиная с 1975 года фирма AT&T начала предоставлять лицензии на использование операционной системы как научно-образовательным учреждениям, так и коммерческим организациям. Поскольку основная часть системы поставлялась в исходных текстах, написанных на языке С, опытным программистам не требовалось детальной документации, чтобы разобраться в архитектуре UNIX. С ростом популярности микропроцессоров

## Файлы, отображаемые в памяти

Системный вызов *mmap(2)* предоставляет механизм доступа к файлам, альтернативный вызовам *read(2)* и *write(2)*. С помощью этого вызова процесс имеет возможность отобразить участки файла в собственное адресное пространство. После этого данные файла могут быть получены или записаны путем чтения или записи в память. Функция *mmap(2)* определяется следующим образом:

```
#include <sys/types.h>
#include <sys/mman.h>
caddr_t mmap(caddr_t addr, size_t len, int prot,
             int flags, int fildes, off_t off);
```

Этот вызов задает отображение *len* байтов файла с дескриптором *fildes*, начиная со смещения *off*, в область памяти со стартовым адресом *addr*. Разумеется, перед вызовом *mmap(2)* файл должен быть открыт с помощью функции *open(2)*. Аргумент *prot* определяет права доступа к области памяти, которые должны соответствовать правам доступа к файлу, указанным в системном вызове *open(2)*. В табл. 2.12 приведены возможные значения аргумента *prot* и соответствующие им права доступа к файлу. Возможно логическое объединение отдельных значений *prot*. Так значение *PROT\_READ | PROT\_WRITE* соответствует доступу *O\_RDWR* к файлу.

**Таблица 2.12.** Права доступа к области памяти

Значение аргумента <i>prot</i>	Описание	Права доступа к файлу
<i>PROT_READ</i>	Область доступна для чтения	<i>r</i>
<i>PROT_WRITE</i>	Область доступна для записи	<i>w</i>
<i>PROT_EXEC</i>	Область доступна для исполнения	<i>x</i>
<i>PROT_NONE</i>	Область недоступна	—

Обычно значение *addr* задается равным 0, что позволяет операционной системе самостоятельно выбрать виртуальный адрес начала области отображения. В любом случае, при успешном завершении возвращаемое системным вызовом значение определяет действительное расположение области памяти.

Операционная система округляет значение *len* до следующей страницы виртуальной памяти<sup>5</sup>. Например, если размер файла 96 байтов, а размер страницы 4 Кбайт, то система все равно выделит область памяти размером 4096 байтов. При этом 96 байтов займут собственно данные файла, а остальные 4000 байтов будут заполнены нулями. Процесс может модифици-

Организация виртуальной памяти подробно рассматривается в главе 3.

ровать и оставшиеся 4000 байтов, но эти изменения не отразятся на содержимом файла. При обращении к участку памяти, лежащему за пределами файла, ядро отправит процессу сигнал `SIGBUS`<sup>6</sup>. Несмотря на то что область памяти может превышать фактический размер файла, процесс не имеет возможности изменить его размер.

Использование права на исполнение (`prot = PROT_EXEC`) позволяет процессу определить собственный механизм загрузки кода. В частности, такой подход используется редактором динамических связей при загрузке динамических библиотек, когда библиотека отображается в адресное пространство процесса. Значение `PROT_NONE` позволяет приложению определить собственные механизмы контроля доступа к разделяемым объектам (например, к разделяемой памяти), разрешая или запрещая доступ к области памяти.

Аргумент `flags` определяет дополнительные особенности управления областью памяти. В табл. 2.13 приведены возможные типы отображения, определяемые аргументом `flags`.

**Таблица 2.13.** Типы отображения

Значение аргумента <code>flags</code>	Описание
<code>MAP_SHARED</code>	Область памяти может совместно использоваться несколькими процессами
<code>MAP_PRIVATE</code>	Область памяти используется только вызывающим процессом
<code>MAP_FIXED</code>	Требует выделения памяти, начиная точно с адреса <code>addr</code>
<code>MAP_NORESERVE</code>	Не требует резервирования области свопинга

В случае указания `MAP_PRIVATE`, для процесса, определившего этот тип отображения, будет создана собственная копия страницы памяти, которую он пытается модифицировать. Заметим, что копия будет создана только при вызове операции записи, до этого остальные процессы, определившие тип отображения как `MAP_SHARED` могут совместно использовать одну и ту же область памяти.

Не рекомендуется использовать флаг `MAP_FIXED`, т. к. это не позволяет системе максимально эффективно распределить память. В случае отсутствия этого флага, ядро пытается выделить область памяти, начиная с адреса наиболее близкого к значению `addr`. Если же значение `addr` установлено равным 0, операционная система получает полную свободу в размещении области отображения.

<sup>6</sup> Если быть более точным, сигнал посыпается процессу, когда происходит обращение к странице памяти, на которую не отображается ни один из участков файла. Таким образом, в приведенном примере сигнал процессу не будет отправлен.

Отображение автоматически снимается при завершении процесса. Процесс также может явно снять отображение с помощью вызова *munmap(2)*. Закрытие файла не приводит к снятию отображения. Следует отметить, что снятие отображения непосредственно не влияет на отображаемый файл, т. е. содержимое страниц области отображения не будет немедленно записано на диск. Обновление файла производится ядром согласно алгоритмам управления виртуальной памятью. В то же время в ряде систем существует функция *msync(3C)*, которая позволяет синхронизировать обновление памяти с обновлением файла на диске<sup>7</sup>.

В качестве примера приведем упрощенную версию утилиты *cp(1)*, копирующую один файл в другой с использованием отображения файла в память.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
#include <fcntl.h>
main(int argc, char *argv[])
{
    int fd_src, fd_dst;
    caddr_t addr_src, addr_dst;
    struct stat filestat;
    /*Первый аргумент — исходный файл, второй — целевой*/
    fd_src=open(argv[1], O_RDONLY);
    fd_dst=open(argv[2], O_RDWR | O_CREAT);
    /*Определим размер исходного файла*/
    fstat(fd_src, &filestat);
    /*Сделаем размер целевого файла равным исходному*/
    lseek(fd_dst, filestat.st_size - 1, SEEK_SET);
    write(fd_dst, "", 1);
    /*Зададим отображение*/
    addr_src=mmap((caddr_t)0, filestat.st_size,
                  PROT_READ, MAP_SHARED, fd_src, 0);
    addr_dst=mmap((caddr_t)0, filestat.st_size,
                  PROT_READ | PROT_WRITE, MAP_SHARED, fd_dst, 0);
    /*Копируем области памяти*/
    memcpy(addr_dst, addr_src, filestat.st_size);
    exit(0);
}
```

Поскольку, как обсуждалось выше, с помощью вызова *munmap(2)* нельзя изменить размер файла, это было сделано с помощью вызова *lseek(2)* с по-

<sup>7</sup> На самом деле *msync(3C)* синхронизирует обновление страниц памяти с *вторичной памятью*. Для областей типа *MAP\_SHARED* вторичной памятью является сам файл на диске. Для областей типа *MAP\_PRIVATE* вторичной памятью является область свопинга. Функция *msync(3C)* также позволяет принудительно обновить страницы, так что при следующем обращении к какой-либо из них ее содержимое будет загружено из вторичной памяти.

следующей записью одного байта так, что размер целевого файла стал равным размеру исходного. При этом в целевом файле образуется "дыра", которая, к счастью, сразу же заполняется содержимым копируемого файла.

## Владение файлами

Владелец-пользователь и владелец-группа файла могут быть изменены с помощью системных вызовов *chown(2)*, *fchown(2)* и *lchown(2)*:

```
#include <unistd.h>
#include <sys/types.h>
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fildes, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

Все три вызова работают одинаково за исключением ситуации, когда адресуемый файл является символической связью. В последнем случае вызов *lchown(2)* действует на сам файл — символическую связь, а не на целевой файл (т. е. не следует символической связи). В функциях *chown(2)* и *lchown(2)* файл адресуется по имени, а в *fchown(2)* — по файловому дескриптору. Если значение *owner* или *group* установлено равным -1, соответствующий владелец файла не изменяется.

В версиях BSD UNIX только суперпользователь может изменить владение файлом. Это ограничение призвано, в первую очередь, не допустить "скрытие" файлов под именем другого пользователя, например, при установке квотирования ресурсов файловой системы. Владельца-группу можно изменить только для файлов, которыми вы владеете, причем им может стать одна из групп, членом которой вы являетесь. Эти же ограничения определены и стандартом POSIX.1.

В системах ветви System V эти ограничения являются конфигурируемыми, и в общем случае в UNIX System V пользователь может изменить владельца собственных файлов.

В случае успешного изменения владельцев файла биты SUID и SGID сбрасываются, если процесс, вызвавший *chown(2)* не обладает правами суперпользователя.

## Права доступа

Как уже обсуждалось в предыдущей главе, каждый процесс имеет четыре пользовательских идентификатора — UID, GID, EUID и EGID. В то время как UID и GID определяют реального владельца процесса, EUID и EGID определяют права доступа процесса к файлам в процессе выполнения. В общем случае реальные и эффективные идентификаторы эквивалентны. Это значит, что процесс имеет те же привилегии, что и пользователь, запустивший его. Однако, как уже обсуждалось выше, возникают ситуации, когда процесс должен получить дополнительные привилегии,

чаще всего — привилегии суперпользователя. Это достигается установкой битов SUID и SGID. Примером такого процесса может служить утилита *passwd(1)*, изменяющая пароль пользователя.

Права доступа к файлу могут быть изменены с помощью системных вызовов *chmod(2)* и *fchmod(2)*:

```
iinclude <sys/types.h>
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

Значение аргумента mode определяет устанавливаемые права доступа и дополнительные атрибуты (такие как SUID, SGID и Sticky bit), и создается путем логического объединения различных флагов, представленных в табл. 2.14. Вторая колонка таблицы содержит восьмеричные значения для девяти битов прав доступа (чтение, запись и выполнение для трех классов доступа) и трех битов дополнительных атрибутов.

**Таблица 2.14.** Флаги аргумента mode

Флаг	Биты	Значение
S_ISUID	04000	Установить бит SUID
S_ISGID	020#0	Установить бит SGID, если # равно 7, 5, 3 или 1 Установить обязательное блокирование файла, если # равно 6, 4, 2 или 0
S_ISVTX	01000	Установить Sticky bit
S_IRWXU	00700	Установить право на чтение, запись и выполнение для владельца-пользователя
S_IRUSR	00400	Установить право на чтение для владельца-пользователя
S_IWUSR	00200	Установить право на запись для владельца-пользователя
S_IXUSR	00100	Установить право на выполнение для владельца-пользователя
S_IRWXG	00070	Установить право на чтение, запись и выполнение для владельца-группы
S_IRGRP	00040	Установить право на чтение для владельца-группы
S_IWGRP	00020	Установить право на запись для владельца-группы
S_IXGRP	00010	Установить право на выполнение для владельца-группы
S_IRWXO	00007	Установить право на чтение, запись и выполнение для остальных пользователей
S_IROTH	00004	Установить право на чтение для остальных пользователей
S_IWOTH	00002	Установить право на запись для остальных пользователей
S_IXOTH	00001	Установить право на выполнение для остальных пользователей

Некоторые флаги, представленные в таблице, уже являются объединением нескольких флагов. Так, например, флаг `S_RWXU` эквивалентен `S_IRUSR | S_IWUSR | S_IXUSR`. Значение флага `S_ISGID` зависит от того, установлено или нет право на выполнение для группы (`S_IXGRP`). В первом случае, он будет означать установку SGID, а во втором — обязательное блокирование файла.

Для иллюстрации приведем небольшую программу, создающую файл с полными правами доступа для владельца, а затем изменяющую их. После каждой установки прав доступа в программе вызывается библиотечная функция `system(3S)`, позволяющая запустить утилиту `ls(l)` и отобразить изменение прав доступа и дополнительных атрибутов.

```
ttinclude <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
main ()
{
    int fd;
/*Создадим файл с правами rwx-----*/
    fd = creat("my_file", S_IRUSR|S_IWUSR|S_IXUSR);
    system("ls -l my_file");
/*Добавим флаг SUID*/
    chmod(fd, S_IRWXU|S_ISUID);
    system("ls -l my_file");
/*Установим блокирование записей файла */
    chmod(fd, S_IRWXU|S_ISUID|S_ISGID);
    system("ls -l my_file");
/*Теперь установим флаг SGID*/
    chmod(fd, S_IRWXU|S_ISUID|S_ISGID|S_IXGRP);
    system("ls -l my_file");
}
```

В результате запуска программы на выполнение, получим следующий вывод:

```
$ a.out
rwx----- 1 andy    user    0  Jan  6 19:28  my_file
-rws----- 1 andy    user    0  Jan  6 19:28  my_file
-rws--l--- 1 andy    user    0  Jan  6 19:28  my_file
-rws--s--- 1 andy    user    0  Jan  6 19:28  my_file
```

## Перемещение по файловой системе

Каждый процесс имеет два атрибута, связанных с файловой системой — *корневой каталог* (root directory) и *текущий рабочий каталог* (current working directory). Когда некоторый файл адресуется по имени (например, в системных вызовах `open(2)`, `creat(2)` или `readlink(2)`), ядро системы производит поиск файла, начиная с корневого каталога, если имя файла задано как абсолютное, либо текущего каталога, если имя файла является относи-

тельным. Абсолютное имя файла начинается с символа '/', обозначающего корневой каталог. Все остальные имена файлов являются относительными. Например, имя `/usr/bin/sh` является абсолютным, в то время как `mydir/test1.c` или `../andy/mydir/test1.c` — относительным, при котором фактическое расположение файла в файловой системе зависит от текущего каталога.

Процесс может изменить свой корневой каталог с помощью системного вызова `chroot(2)` или `fchroot(2)`:

```
#include <unistd.h>
int chroot(const char *path);
int fchroot(int fildes);
```

После этого поиск всех адресуемых файлов с абсолютными именами будет производиться, начиная с нового каталога, указанного аргументом `path`. Например, после изменения корневого каталога на домашний каталог пользователя абсолютное имя инициализационного скрипта `.profile` станет `./.profile`<sup>8</sup>.

Изменение корневого каталога может потребоваться, например, при распаковке архива, созданного с абсолютными именами файла, в другом месте файловой системы, либо при работе над большим программным проектом, затрагивающим существенную часть корневой файловой системы. В этом случае для отладочной версии удобно создать собственную корневую иерархию.

Процесс также может изменить и текущий каталог. Для этого используются системные вызовы `chdir(2)` или `fchdir(2)`:

```
#include <unistd.h>
int chdir(const char *path);
int fchdir(int fildes);
```

Например, внутренняя команда командного интерпретатора `cd` может быть реализована следующим кодом:

```
char newdir[PATH_MAX];

/* Предположим, что имя нового каталога, введенного пользователем,
уже находится в переменной newdir*/
if(chdir(newdir) == -1) perror("sh: cd");
```

Изменение корневого каталога разрешено только для администратора системы — суперпользователя. Эта операция таит в себе определенную опасность, т. к. часть утилит операционной системы (если не все) могут оказаться недоступными, в том числе и команда `chroot(1M)`. Таким образом, последствия необдуманного изменения корневого каталога могут стать необратимыми.

## Метаданные файла

Как уже говорилось, каждый файл помимо собственно данных содержит *метаданные*, описывающие его характеристики, например, владельцев, права доступа, тип и размер файла, а также содержащие указатели на фактическое расположение данных файла. Метаданные файла хранятся в структуре *inode*. Часть полей этой структуры могут быть получены с помощью системных вызовов *stat(2)*:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
int fstat(int fildes, struct stat *buf);
```

В качестве аргумента функции принимают имя файла или файловый дескриптор (*fstat(2)*) и возвращают заполненные поля структуры *stat*, которые приведены в табл. 2.15.

**Таблица 2.15.** Поля структуры *stat*

Поле	Значение
mode t st mode	Тип файла и права доступа
ino_t st_ino	Номер <i>inode</i> . Поля <i>st_ino</i> и <i>st_dev</i> однозначно определяют обычные файлы
dev t st dev	Номер устройства, на котором расположен файл (номер устройства файловой системы)
dev_t st_rdev	Для специального файла устройства содержит номер устройства, адресуемого этим файлом
nlink t st nlink	<b>Число жестких связей</b>
uid t st uid	Идентификатор пользователя-владельца файла
gid_t st_gid	Идентификатор группы-владельца файла
off_t st size	Размер файла в байтах. Для специальных файлов устройств это поле не определено
time_t st_atime	Время последнего доступа к файлу
time_t st_mtime	Время последней модификации данных файла
time_t st_ctime	Время последней модификации метаданных файла
long st_blksize	Оптимальный размер блока для операций ввода/вывода. Для специальных файлов устройств и каналов это поле не определено
long st_blocks	Число размещенных 512-байтовых блоков хранения данных. Для специальных файлов устройств это поле не определено

Для определения типа файла служат следующие макроопределения, описанные в файле *<sys/stat.h>*:

Таблица 2.16. Определение типа файла

Макроопределение	Тип файла
<code>S_ISFIFO(mode)</code>	FIFO
<code>S_ISCHR(mode)</code>	Специальный файл символьного устройства
<code>S_ISDIR(mode)</code>	Каталог
<code>S_ISBLK(mode)</code>	Специальный файл блочного устройства
<code>S_ISREG(mode)</code>	Обычный файл
<code>S_ISLNK(mode)</code>	Символическая связь
<code>S_ISSOCK(mode)</code>	Сокет

Все значения времени, связанные с файлом (время доступа, модификации данных и метаданных) хранятся в секундах, прошедших с 0 часов 1 января 1970 года. Заметим, что информация о времени создания файла отсутствует.

Приведенная ниже программа выводит информацию о файле, имя которого передается ей в качестве аргумента:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
main(int argc, char *argv[])
{
    struct stat s;
    char *ptype;
    lstat(argv[1], &s);
/*Определим тип файла*/
    if(S_ISREG(s.st_mode) ) ptype = "Обычный файл";
    else if(S_ISDIR(s.st_mode) ) ptype = "Каталог";
    else if(S_ISLNK(s.st_mode)) ptype = "Симв. связь";
    else if(S_ISCHR(s.st_mode)) ptype = "Симв. устройство";
    else if(S_ISBLK(s.st_mode)) ptype = "Бл.устройство";
    else if(S_ISSOCK(s.st_mode)) ptype = "Сокет";
    else if(S_ISFIFO(s.st_mode)) ptype = "FIFO";
    else ptype = "Неизвестный тип";
/*Выведем информацию о файле*/
/*Его тип*/
    printf("type = %s\n", ptype);
/*Права доступа*/
    printf("perm = %o\n", s.st_mode & S_IAMB);
/*Номер inode*/
    printf("inode = %d\n", s.st_ino);
/*Число связей*/
    printf("nlink = %d\n", s.st_nlink);
/*Устройство, на котором хранятся данные файла*/
    printf("dev = (%d, %d)\n", major(s.st_dev), minor(s.st_dev));
```

```

/*Владельцы файла*/
    printf("UID = %d\n", s.st_uid);
    printf("GID = %d\n", s.st_gid);
/*Для специальных файлов устройств — номера устройства*/
    printf("rdev = (%d, %d)\n", major(s.st_rdev),
           minor(s.st_rdev));
/*Размер файла*/
    printf("size = %d\n", s.st_size);
/*Время доступа, модификации и модификации метаданных*/
    printf("atime = %s", ctime(&s.st_atime));
    printf("mtime = %s", ctime(&s.st_mtime));
    printf("ctime = %s", ctime(&s.st_ctime));
}

```

Программа использует библиотечные функции *major(3C)* и *minor(3C)*, возвращающие, соответственно, старший и младший номера устройства. Функция *ctime(3C)* преобразует системное время в удобный формат.

Запуск программы на выполнение приведет к следующим результатам:

```

$ a.out ftype.c
type = Обычный файл
perm = 644
inode = 13
nlink = 1
dev = (1, 42)
UID = 286
GID = 100
rdev = (0, 0)
size = 1064
atime = Wed Jan 8 17:25:34 1997
mtime = Wed Jan 8 17:19:27 1997
ctime = Wed Jan 8 17:19:27 1997
$ ls -il /tmp/ftype.c
13 -rw-r--r-- 1 andy user 1064 Jan 8 17:19 ftype.c

```

## Процессы

В главе 1 уже упоминались процессы. Однако знакомство ограничивалось пользовательским, или командным интерфейсом операционной системы. В этом разделе попробуем взглянуть на них с точки зрения программиста.

Процессы являются основным двигателем операционной системы. Большинство функций выполняется ядром требованию того или иного процесса. Выполнение этих функций контролируется привилегиями процесса, которые соответствуют привилегиям пользователя, запустившего его.

В этом разделе рассматриваются:

- Идентификаторы процесса

- Программный интерфейс управления памятью: системные вызовы низкого уровня и библиотечные функции, позволяющие упростить управление динамической памятью процесса.
  - Важнейшие системные вызовы, обеспечивающие создание нового процесса и запуск новой программы. Именно с помощью этих вызовов создается существующая популяция процессов в операционной системе и ее функциональность.
  - Сигналы и способы управления ими. Сигналы можно рассматривать как элементарную форму межпроцессного взаимодействия, позволяющую процессам сообщать друг другу о наступлении некоторых событий. Более мощные средства будут рассмотрены в разделе "Взаимодействие между процессами" главы 3.
- Π Группы и сеансы; взаимодействие процесса с пользователем.
- Π Ограничения, накладываемые на процесс, и функции, которые позволяют управлять этими ограничениями.

## Идентификаторы процесса

Вы уже знаете, что каждый процесс характеризуется набором атрибутов и идентификаторов, позволяющих системе управлять его работой. Важнейшими из них являются идентификатор процесса **PID** и идентификатор родительского процесса **PPID**. **PID** является именем процесса в операционной системе, по которому мы можем адресовать его, например, при отправлении сигнала. **PPID** указывает на родственные отношения между процессами, которые (как и в жизни) в значительной степени определяют его свойства и возможности.

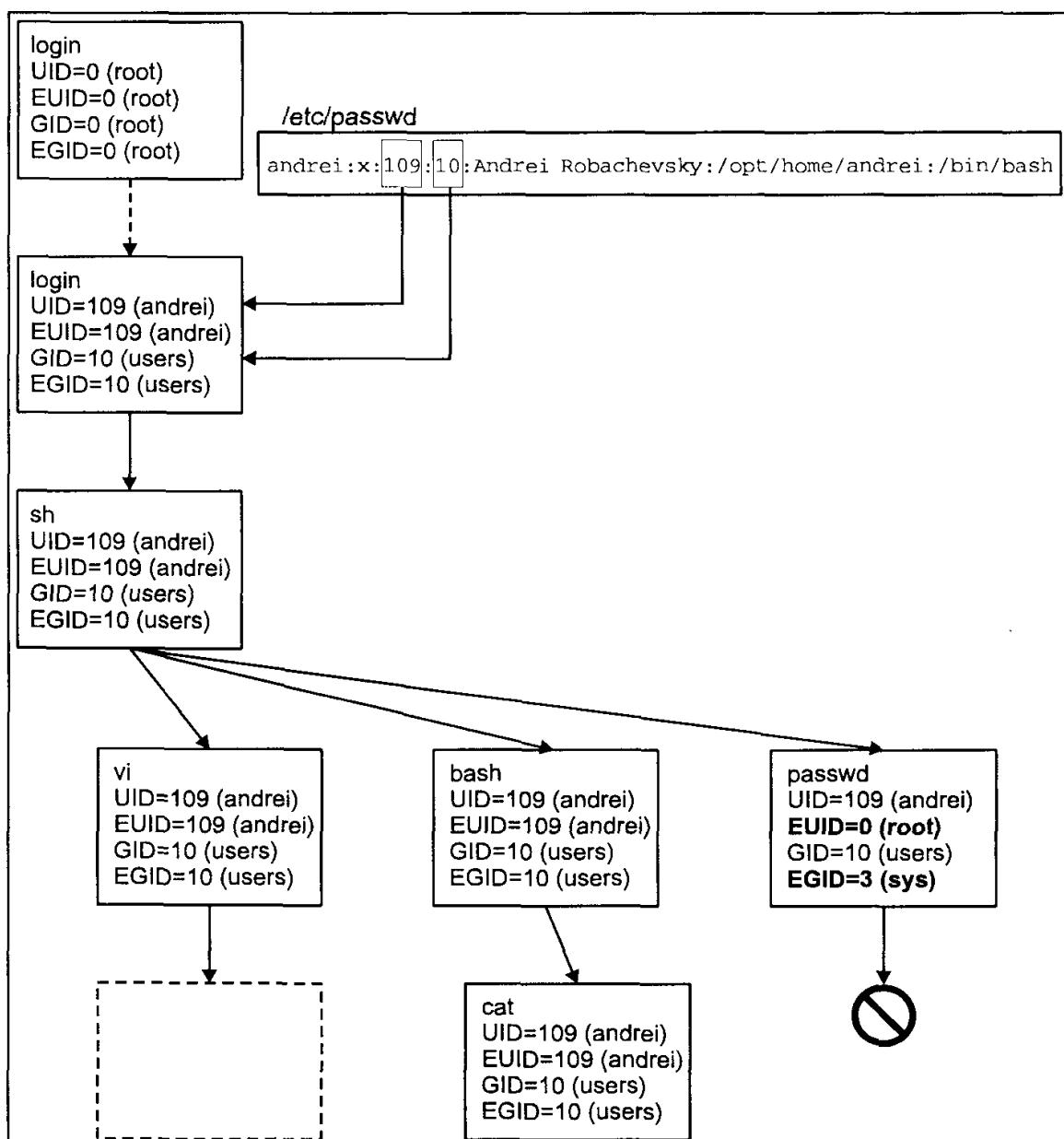
Однако нельзя не отметить еще четыре идентификатора, играющие решающую роль при доступе к системным ресурсам: идентификатор пользователя **UID**, эффективный идентификатор пользователя **EUID**, идентификатор группы **GID** и эффективный идентификатор группы **EGID**. Эти идентификаторы определяют права процесса в файловой системе, и как следствие, в операционной системе в целом. Запуская различные команды и утилиты, можно заметить, что порожденные этими командами процессы полностью отражают права пользователя UNIX. Причина проста — все процессы, которые запускаются, имеют идентификатор пользователя и идентификатор группы. Исключение составляют процессы с установленными флагами **SUID** и **SGID**.

При регистрации пользователя в системе утилита *login(1)* запускает командный интерпретатор, — *login shell*, имя которого является одним из атрибутов пользователя. При этом идентификаторам **UID** (**EUID**) и **GID** (**EGID**) процесса *shell* присваиваются значения, полученные из записи пользователя в файле паролей **/etc/passwd**. Таким образом, командный интерпретатор обладает правами, определенными для данного пользователя.

При запуске программы командный интерпретатор порождает процесс, который наследует все четыре идентификатора и, следовательно, имеет те же права, что и shell. Поскольку в конкретном сеансе работы пользователя в системе прародителем всех процессов является login shell, то и их пользовательские идентификаторы будут идентичны.

Казалось бы, эту стройную систему могут "испортить" утилиты с установленными флагами SUID и SGID. Но не стоит волноваться — как правило, такие программы не позволяют порождать другие процессы, в противном случае, эти утилиты необходимо немедленно уничтожить!

На рис. 2.10 показан процесс наследования пользовательских идентификаторов в рамках одного сеанса работы.



**Рис. 2.10.** Наследование пользовательских идентификаторов

Для получения значений идентификаторов процесса используются следующие системные вызовы:

```
#include <sys/types.h>
#include <unistd.h>
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

Эти функции возвращают для сделавшего вызов процесса соответственно реальный и эффективный идентификаторы пользователя и реальный и эффективный идентификаторы группы.

Процесс также может изменить значения этих идентификаторов с помощью системных вызовов:

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);
int setegid(gid_t egid);
int seteuid(uid_t euid);
int setgid(gid_t gid);
```

Системные вызовы *setuid(2)* и *setgid(2)* устанавливают сразу реальный и эффективный идентификаторы, а системные вызовы *seteuid(2)* и *setegid(2)* — только эффективные.

Ниже приведен фрагмент программы *login(1)*, изменяющей идентификаторы процесса на значения, полученные из записи файла паролей. В стандартной библиотеке имеется ряд функций работы с записями файла паролей, каждая из которых описывается структурой *passwd*, определенной в файле **<pwd.h>**. Поля этой структуры приведены в табл. 2.17.

**Таблица 2.17.** Поля структуры *passwd*

Поле	Значение
char *pw_name	<b>Имя пользователя</b>
char *pw_passwd	Строка, содержащая пароль в зашифрованном виде; из соображения безопасности в большинстве систем пароль хранится в файле /etc/shadow, а это поле не используется
uid_t pw_uid	Идентификатор пользователя
gid_t pw_gid	Идентификатор группы
char *pw_gecos	Комментарий (поле GECOS), обычно реальное имя пользователя и дополнительная информация
char *pw_dir	Домашний каталог пользователя
char *pw_shell	Командный интерпретатор

Функция, которая потребуется для нашего примера, позволяет получить запись файла паролей по имени пользователя. Она имеет следующий вид:

```
linclude <pwd.h>
struct passwd *getpwnam(const char *name);
```

Итак, перейдем к фрагменту программы:

```
struct passwd *pw;
char logname[MAXNAME];
/*Массив аргументов при запуске командного интерпретатора*/
char *arg[MAXARG];
/*Окружение командного интерпретатора*/
char *envir[MAXENV];

/*Проведем поиск записи пользователя с именем logname, которое
было введено на приглашение "login:"*/
pw = getpwnam(logname);
/*Если пользователь с таким именем не найден, повторить
приглашение*/
if ( pw == 0 ) retry();
/*В противном случае установим идентификаторы процесса равными
значениям, полученным из файла паролей и запустим командный
интерпретатор*/
else
{
    setuid(pw->pw_uid);
    setgid(pw->pw_gid);
    execve(pw->pw_shell, arg, envir);
}
```

Вызов *execve(2)* запускает на выполнение программу, указанную в первом аргументе. Мы рассмотрим эту функцию в разделе "Создание и управление процессами" далее в этой главе.

## Выделение памяти

При обсуждении формата исполняемых файлов и образа программы в памяти мы отметили, что сегменты данных и стека могут изменять свои размеры. Если для стека операцию выделения памяти операционная система производит автоматически, то приложение имеет возможность управлять ростом сегмента данных, выделяя дополнительную память из хипа (heap — куча). Рассмотрим этот программный интерфейс.

Память, которая используется сегментами данных и стека, может быть выделена несколькими различными способами как во время создания процесса, так и динамически во время его выполнения. Существует четыре способа выделения памяти:

1. Переменная объявлена как глобальная, и ей присвоено начальное значение в исходном тексте программы, например:

```
char ptype = "Unknown file type";
```

Строка `ptype` размещается в сегменте инициализированных данных исполняемого файла, и для нее выделяется соответствующая память при создании процесса.

2. Значение глобальной переменной неизвестно на этапе компиляции, например:

```
char ptype[32];
```

В этом случае место в исполняемом файле для `ptype` не резервируется, но при создании процесса для данной переменной выделяется необходимое количество памяти, заполненной нулями, в сегменте BSS.

3. Переменные автоматического класса хранения, используемые в функциях программы, используют стек. Память для них выделяется при вызове функции и освобождается при возврате. Например:

```
fund ()  
{  
    int a;  
    char *b;  
    static int c = 4;
```

В данном примере переменные `a` и `b` размещаются в сегменте стека. Переменная `c` размещается в сегменте инициализированных данных и загружается из исполняемого файла либо во время создания процесса, либо в процессе загрузки страниц по требованию. Более подробно страничный механизм описан в главе 3.

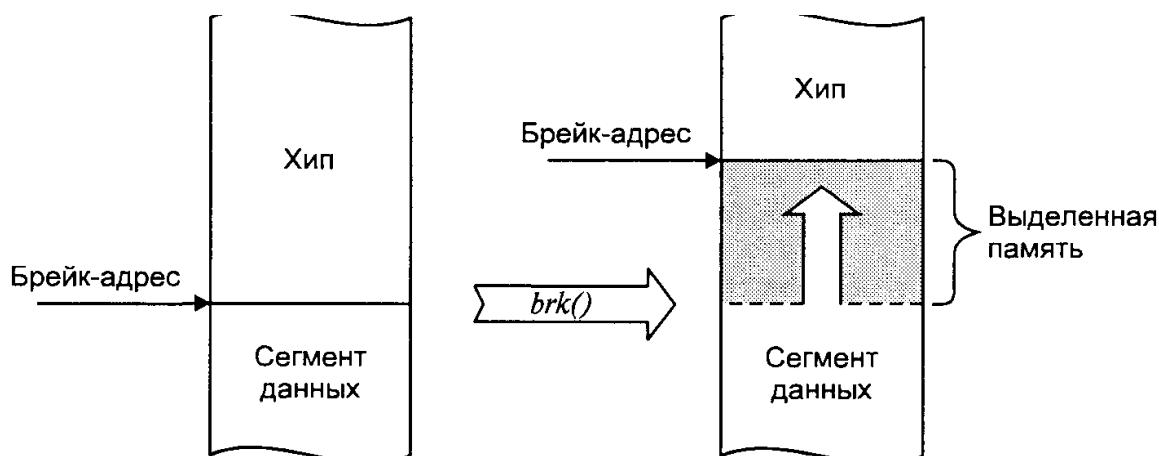
4. Выделение памяти явно запрашивается некоторыми системными вызовами или библиотечными функциями. Например, библиотечная функция `malloc(3С)` запрашивает выделение дополнительной памяти, которая в дальнейшем используется для динамического размещения данных. Функция `ctime(3С)`, предоставляющая системное время в удобном формате, также требует выделения памяти для размещения строки, содержащей значения текущего времени, указатель на которую возвращается программе.

Напомним, что дополнительная память выделяется из хипа (heap) — области виртуальной памяти, расположенной рядом с сегментом данных, размер которой меняется для удовлетворения запросов на размещение. Следующий за сегментом данных адрес называется *разделительным* или *брейк-адресом* (break address). Изменение размера сегмента данных по существу заключается в изменении брейк-адреса. Для изменения его значе-

ния UNIX предоставляет процессу два системных вызова — *brk(2)* и *sbrk(2)*.

```
#include <unistd.h>
int brk(void *endds);
void *sbrk(int incr);
```

Системный вызов *brk(2)* позволяет установить значение брейк-адреса равным *endds* и, в зависимости от его значения, выделяет или освобождает память (рис. 2.11). Функция *sbrk(2)* изменяет значение брейк-адреса на величину *incr*. Если значение *incr* больше 0, происходит выделение памяти, в противном случае, память освобождается<sup>9</sup>.



**Рис. 2.11.** Динамическое выделение памяти с помощью *brk(2)*

Существуют четыре стандартные библиотечные функции, предназначенные для динамического выделения/освобождения памяти.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nelem, size_t elsize);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

Функция *malloc(3C)* выделяет указанное аргументом *size* число байтов.

Функция *calloc(3C)* выделяет память для указанного аргументом *nelem* числа объектов, размер которых *elsize*. Выделенная память инициализируется нулями.

Функция *realloc(3C)* изменяет размер предварительно выделенной области памяти (увеличивает или уменьшает, в зависимости от знака аргумента *size*). Увеличение размера может привести к перемещению всей области в

<sup>9</sup> Заметим, что в некоторых системах дополнительная память выделяется (или освобождается) в порциях, кратных размеру страницы. Например, выделение всего 100 байтов на самом деле приведет к выделению 4096 байтов, если размер страницы равен 4К.

другое место виртуальной памяти, где имеется необходимое свободное непрерывное виртуальное адресное пространство.

Функция *free(3C)* освобождает память, предварительно выделенную с помощью функций *malloc(3C)*, *calloc(3C)* или *realloc(3C)*, указатель на которую передается через аргумент *ptr*.

Указатель, возвращаемый функциями *malloc(3C)*, *calloc(3C)* и *realloc(3C)*, соответствующим образом выровнен, таким образом выделенная память пригодна для хранения объектов любых типов. Например, если наиболее жестким требованием по выравниванию в системе является размещение переменных типа *double* по адресам, кратным 8, то это требование будет распространено на все указатели, возвращаемыми этими функциями.

Упомянутые библиотечные функции обычно используют системные вызовы *sbrk(2)* или *brk(2)*. Хотя эти системные вызовы позволяют как выделять, так и освобождать память, в случае библиотечных функций память реально не освобождается, даже при вызове *free(3C)*. Правда, с помощью функций *malloc(3C)*, *calloc(3C)* или *realloc(3C)* можно снова выделить и использовать эту память и снова освободить ее, но она не передается обратно ядру, а остается в пуле *malloc(3C)*.

Для иллюстрации этого положения приведем небольшую программу, выделяющую и освобождающую память с помощью функций *malloc(3C)* и *free(3C)*, соответственно. Контроль действительного значения брейк-адреса осуществляется с помощью системного вызова *sbrk(2)*:

```
#include <unistd.h>
#include <stdlib.h>
main()
{
    char *obrk;
    char *nbrk;
    char *naddr;
    /*Определим текущий брейк-адрес*/
    obrk = sbrk(0);
    printf("Текущий брейк-адрес= 0x%x\n", obrk);
    /*Выделим 64 байта из хипа*/
    naddr = malloc(64);
    printf("malloc(64)\n");
    /*Определим новый брейк-адрес*/
    nbrk = sbrk(0);
    printf("Новый адрес области malloc= 0x%x,
брейк-адрес= 0x%x (увеличение на %d байтов)\n",
           naddr, nbrk, nbrk - obrk);
    /*"Освободим" выделенную память и проверим, что произошло на
самом деле*/
    free(naddr);
    printf("free(0x%x)\n", naddr);
    obrk = sbrk(0);
```

```

    printf("Новый брейк-адрес= 0x%x (увеличение на %d байтов)\n",
obrk, obrk + nbrk);
}

```

Откомпилируем и запустим программу:

```

$ a.out
Текущий брейк-адрес= 0x20ac0
malloc (64)
Новый адрес области malloc = 0x20ac8, брейк-адрес = 0x22ac0
(увеличение на 8192 байтов)
free (0x20ac8)
Новый брейк-адрес = 0x22ac0 (увеличение на 0 байтов)
$
```

Как видно из вывода программы, несмотря на освобождение памяти функцией *free(3C)*, значение брейк-адреса не изменилось. Также можно заметить, что функция *malloc(3C)* выделяет больше памяти, чем требуется. Дополнительная память выделяется для необходимого выравнивания и для хранения внутренних данных *malloc(3C)*, таких как размер области, указатель на следующую область и т. п.

## Создание и управление процессами

Работая в командной строке shell вы, возможно, не задумывались, каким образом запускаются программы. На самом деле каждый раз порождается новый процесс, а затем загружается программа. В UNIX эти два этапа четко разделены. Соответственно система предоставляет два различных системных вызова: один для создания процесса, а другой для запуска новой программы.

Новый процесс порождается с помощью системного вызова *fork(2)*:

```

#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);

```

*Порожденный*, или *дочерний* процесс, хотя это кажется странным, является точной копией процесса, выполнившего этот вызов, или *родительского* процесса. В частности, дочерний процесс наследует такие атрибуты родителя, как:

- Я идентификаторы пользователя и группы,
- Я переменные окружения,
- Я диспозицию сигналов и их обработчики,
- Я ограничения, накладываемые на процесс,
- Я текущий и корневой каталог,
- Я маску создания файлов,
- Я все файловые дескрипторы, включая файловые указатели,
- Я управляющий терминал.

Более того, виртуальная память дочернего процесса не отличается от образа родительского: такие же сегменты кода, данных, стека, разделяемой

памяти и т. д. После возврата из вызова *fork(2)*, который происходит и в родительский и в дочерний процессы, оба начинают выполнять одну и ту же инструкцию.

Легче перечислить немногочисленные различия между этими процессами, а именно:

- дочернему процессу присваивается уникальный идентификатор PID,
- идентификаторы родительского процесса PPID у этих процессов различны,
- дочерний процесс свободен от сигналов, ожидающих доставки,
- значение, возвращаемое системным вызовом *fork(2)* различно для родителя и потомка.

Последнее замечание требует объяснения. Как уже говорилось, возврат из функции *fork(2)* происходит как в родительский, так и в дочерний процесс. При этом возвращаемое родителю значение равно PID дочернего процесса, а дочерний, в свою очередь, получает значение, равное 0. Если *fork(2)* возвращает -1, то это свидетельствует об ошибке (естественно, в этом случае возврат происходит только в процесс, выполнивший системный вызов).

В возвращаемом *fork(2)* значении заложен большой смысл, поскольку оно позволяет определить, кто является родителем, а кто — потомком, и соответственно разделить функциональность. Поясним это на примере:

```
int pid;
pid = fork();
if (pid == -1
    perror("fork");
    exit(1);
}
if (pid == 0)
    /*Эта часть кода выполняется дочерним процессом*/
    printf("Потомок\n");
}
else
{
    /*Эта часть кода выполняется родительским процессом*/
    printf("Родитель\n");
}
```

Таким образом, порождение нового процесса уже не кажется абсолютно бессмысленным, поскольку родитель и потомок могут параллельно выполнять различные функции. В данном случае, это вывод на терминал различных сообщений, однако можно представить себе и более сложные при-

ложении. В частности, большинство серверов, одновременно обслуживающих несколько запросов, организованы именно таким образом: при поступлении запроса порождается процесс, который и выполняет необходимую обработку. Родительский процесс является своего рода супервизором, принимающим запросы и распределяющим их выполнение. Очевидным недостатком такого подхода является то, что вся функциональность по-прежнему заложена в одном исполняемом файле и, таким образом, ограничена.

UNIX предлагает системный вызов, предназначенный исключительно для запуска программ, т. е. загрузки другого исполняемого файла. Это системный вызов *exec(2)*, представленный на программном уровне несколькими модификациями:

```
#include <unistd.h>
int execl (const char *path, const char *arg0, ...,
            const char *argn, char * /*NULL*/);
int execv (const char *path, char *const argv[]);
int execle (const char *path, char *const arg0 [], ...,
            const char *argn, char * /*NULL*/, char *const envp[]);
int execve (const char *path, char *const argv [],
            char *const envp[]);
int execlp (const char *file, const char *arg0, ...,
            const char *argn, char * /*NULL*/);
int execvp (const char *file, char *const argv[]);
```

Все эти функции по существу являются надстройками системного вызова *execve(2)*, который в качестве аргументов получает имя запускаемой программы (исполняемого файла), набор аргументов и список переменных окружения. После выполнения *execve(2)* не создается новый процесс, а образ существующего полностью заменяется на образ, полученный из указанного исполняемого файла. На рис. 2.12 показано, как связаны между собой приведенные выше функции.

В отличие от вызова *fork(2)*, новая программа наследует меньше атрибутов. В частности, наследуются:

- идентификаторы процесса PID и PPID,
- идентификаторы пользователя и группы,
- эффективные идентификаторы пользователя и группы (в случае, если для исполняемого файла не установлен флаг SUID или SGID),
- П ограничения, накладываемые на процесс,
- О текущий и корневой каталоги,
- П маска создания файлов,
- П управляющий терминал,
- файловые дескрипторы, для которых не установлен флаг FD\_CLOEXEC.

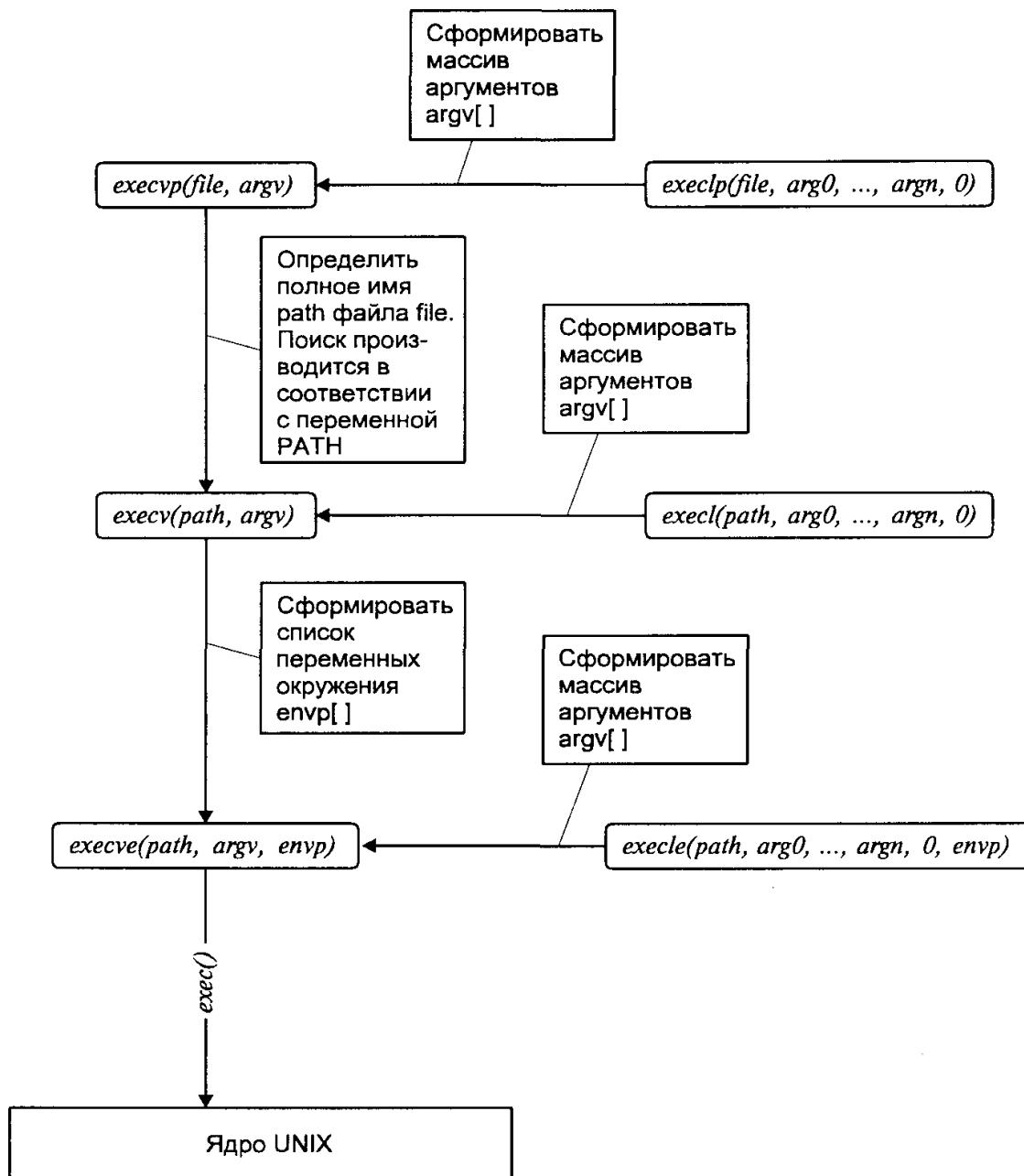


Рис. 2.12. Семейство функций exec(2)

Наследование характеристик процесса играет существенную роль в работе операционной системы. Так наследование идентификаторов владельцев процесса гарантирует преемственность привилегий и, таким образом, неизменность привилегий пользователя при работе в UNIX. Наследование файловых дескрипторов позволяет установить направления ввода/вывода для нового процесса или новой программы. Именно так действует командный интерпретатор. Мы вернемся к вопросу о наследовании в главе 3.

В главе 1 уже говорилось о частом объединении вызовов `fork(2)` и `exec(2)`, получившем специальное название `fork-and-exec`. Таким образом загружается подавляющее большинство программ, которые выполняются в системе.

При порождении процесса, который впоследствии может загрузить новую программу, "родителю" может быть небезынтересно узнать о завершении выполнения "потомка". Например, после того как запущена утилита *ls(1)*, командный интерпретатор приостанавливает свое выполнение до завершения работы утилиты и только после этого выдает свое приглашение на экран. Можно привести еще множество ситуаций, когда процессам необходимо синхронизировать свое выполнение с выполнением других процессов. Одним из способов такой синхронизации является обработка родителем сигнала *SIGCHLD*, отправляемого ему при "смерти" потомка. Механизм сигналов мы рассмотрим в следующем разделе. Сейчас же остановимся на другом подходе.

Операционная система предоставляет процессу ряд функций, позволяющих ему контролировать выполнение потомков. Это функции *wait(2)*, *waitid(2)* и *waitpid(2)*:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);
int waitid(idtype_t idtype, id_t id,
           siginfo_t *infop, int options);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Первый из этих вызовов *wait(2)* обладает самой ограниченной функциональностью — он позволяет заблокировать выполнение процесса, пока кто-либо из его непосредственных потомков не прекратит существование. Вызов *wait(2)* немедленно возвратит состояние уже завершившегося дочернего процесса в переменной *stat\_loc*, если последний находится в состоянии зомби. Значение *stat\_loc* может быть проанализировано с помощью следующих макроопределений:

<i>WIFEXITED(status)</i>	Возвращает истинное (ненулевое) значение, если процесс завершился нормально.
<i>WEXITSTATUS(status)</i>	Если <i>WIFEXITED(status)</i> не равно нулю, определяет код возврата завершившегося процесса (аргумент функции <i>exit(2)</i> ).
<i>WIFSIGNalled(status)</i>	Возвращает истину, если процесс завершился по сигналу.
<i>WTERMSIG(status)</i>	Если <i>WIFSIGNalled(status)</i> не равно нулю, определяет номер сигнала, вызвавшего завершение выполнения процесса.
<i>WCOREDUMP(status)</i>	Если <i>WIFSIGNalled(status)</i> не равно нулю, макрос возвращает истину в случае создания файла <b>core</b> .

Системный вызов *waitid(2)* предоставляет больше возможностей для контроля дочернего процесса. Аргументы *idtype* и *id* определяют, за какими именно из дочерних процессов требуется следить:

## Значение аргумента **idtype**      Описание

P_PID	<i>waitid(2)</i> блокирует выполнение процесса, следя за потомком, PID которого равен id.
P_PGID	<i>waitid(2)</i> блокирует выполнение процесса, следя за потомками, идентификаторы группы которых равны id.
P_ALL	<i>waitid(2)</i> блокирует выполнение процесса, следя за всеми непосредственными потомками.

Аргумент **options** содержит флаги, объединенные логическим ИЛИ, определяющие, за какими изменениями в состоянии потомков следует *waitid(2)*:

## Флаги аргумента **options**      Описание

WEXITED	Предписывает ожидать завершения выполнения процесса.
WTRAPPED	Предписывает ожидать ловушки (trap) или точки останова (breakpoint) для трассируемых процессов.
WSTOPPED	Предписывает ожидать останова процесса из-за получения сигнала.
WCONTINUED	Предписывает вернуть статус процесса, выполнение которого было продолжено после останова.
WNOHANG	Предписывает завершить свое выполнение, если отсутствует статусная информация (т. е. отсутствует ожидаемое событие).
WNOWAIT	Предписывает получить статусную информацию, но не уничтожать ее, оставив дочерний процесс в состоянии ожидания.

Аргумент **infop** указывает на структуру **siginfo\_t**, которая будет заполнена информацией о потомке. Мы рассмотрим эту структуру в следующем разделе.

Функция *waitpid(2)*, как и функции *wait(2)* и *waitid(2)*, позволяет контролировать определенное множество дочерних процессов.

В заключение для иллюстрации описанных в этом разделе системных вызовов приведем схему работы командного интерпретатора при запуске команды.

```
/*Вывести приглашение shell*/
write(1, "$ ", 2);
/*Считать пользовательский ввод*/
get input(inputbuf);
```

```

/*Произвести разбор ввода: выделить команду and и ее аргументы arg[]*/
parse_input(inputbuf, and, arg);
/*Породить процесс*/
pid = fork();
if (pid == 0)
{
/*Запустить программу*/
execvp(cmd, arg);
/*При нормальном запуске программы эта часть кода выполняться уже не будет — можно смело выводить сообщение об ошибке*/
pexit(cmd);
}
else
/*Родительский процесс (shell) ожидает завершения выполнения потомка*/
wait(&status);

```

## Сигналы

Сигнал является способом передачи уведомления о некотором произошедшем событии между процессами или между ядром системы и процессами. Сигналы можно рассматривать, как простейшую форму межпроцессного взаимодействия, хотя на самом деле они больше напоминают программные прерывания, при которых нарушается нормальное выполнение процесса.

Сигналы появились уже в ранних версиях UNIX, но их реализация не была достаточно надежной. Сигнал мог быть "потерян", возникали также определенные сложности с отключением (блокированием) сигналов на время выполнения критических участков кода. В последующие версии системы, как BSD, так и System V, были внесены изменения, позволившие реализовать *надежные (reliable) сигналы*. Однако модель сигналов, принятая в версиях BSD, была несовместима с моделью версий System V. В настоящее время стандарт POSIX.1 вносит определенность в интерфейс надежных сигналов.

Прежде всего, каждый сигнал имеет уникальное символьное имя и соответствующий ему номер. Например, сигнал прерывания, посылаемый процессу при нажатии пользователем клавиши *<Del>* или *<Ctrl>+<C>*, имеет имя **SIGINT**. Сигнал, генерируемый комбинацией *<Ctrl>+<\>*, называется **SIGQUIT**. Седьмая редакция UNIX насчитывала 15 различных сигналов, а в современных версиях их число увеличилось вдвое.

Сигнал может быть отправлен процессу либо ядром, либо другим процессом с помощью системного вызова *kill(2)*:

```

ttinclude <sys/types.h>
#include <signal.h>

int kill(pid t pid, int sig);

```

Аргумент `pid` адресует процесс, которому посыпается сигнал. Аргумент `sig` определяет тип отправляемого сигнала.

К генерации сигнала могут привести различные ситуации:

- Ядро отправляет процессу (или группе процессов) сигнал при нажатии пользователем определенных клавиш или их комбинаций. Например, нажатие клавиши `<Del>` (или `<Ctrl>+<C>`) приведет к отправке сигнала `SIGINT`, что используется для завершения процессов, вышедших из-под контроля<sup>10</sup>.
  - Аппаратные *особые ситуации*, например, деление на 0, обращение к недопустимой области памяти и т. д., также вызывают генерацию сигнала. Обычно эти ситуации определяются аппаратурой компьютера, и ядру посыпается соответствующее уведомление (например, в виде прерывания). Ядро реагирует на это отправкой соответствующего сигнала процессу, который находился в стадии выполнения, когда произошла особая ситуация.
- П Определенные программные состояния системы или ее компонентов также могут вызвать отправку сигнала. В отличие от предыдущего случая, эти условия не связаны с аппаратной частью, а имеют чисто программный характер. В качестве примера можно привести сигнал `SIGALRM`, отправляемый процессу, когда срабатывает таймер, ранее установленный с помощью вызова `alarm(2)`.

С помощью системного вызова `kill(2)` процесс может послать сигнал как самому себе, так и другому процессу или группе процессов. В этом случае процесс, посылающий сигнал, должен иметь те же реальный и эффективный идентификаторы, что и процесс, которому сигнал отправляется. Разумеется, данное ограничение не распространяется на процессы, обладающие привилегиями суперпользователя. Такие процессы имеют возможность отправлять сигналы любым процессам системы.

Как уже говорилось в предыдущей главе, процесс может выбрать одно из трех возможных действий при получении сигнала:

- П игнорировать сигнал,
- П перехватить и самостоятельно обработать сигнал,
- П позволить действие по умолчанию.

Текущее действие при получении сигнала называется *диспозицией сигнала*.

Напомним, что сигналы `SIGKILL` и `SIGSTOP` невозможно ни игнорировать, ни перехватить. Сигнал `SIGKILL` является силовым методом завершения выполнения "непослушного" процесса, а от работоспособности `SIGSTOP` зависит функционирование системы управления заданиями.

<sup>10</sup>Сигналы этого рода генерируются драйвером терминала. Настройка терминального драйвера позволяет связать условие генерации сигнала с любой клавишей.

Условия генерации сигнала и действие системы по умолчанию приведены в табл. 2.18. Как видно из таблицы, при получении сигнала в большинстве случаев по умолчанию происходит завершение выполнения процесса. В ряде случаев в текущем рабочем каталоге процесса также создается файл **core** (в таблице такие случаи отмечены как "Завершить+core"), в котором хранится образ памяти процесса. Этот файл может быть впоследствии проанализирован программой-отладчиком для определения состояния процесса непосредственно перед завершением. Файл **core** не будет создан в следующих случаях:

- исполняемый файл процесса имеет установленный бит SUID, и реальный владелец-пользователь процесса не является владельцем-пользователем исполняемого файла;
- исполняемый файл процесса имеет установленный бит SGID, и реальный владелец-группа процесса не является владельцем-группой исполняемого файла;
- П процесс не имеет права записи в текущем рабочем каталоге;
- Д размер файла **core** слишком велик (превышает допустимый предел **RLIMIT\_CORE**, см. раздел "Ограничения" далее в этой главе).

**Таблица 2.18.** Сигналы

Название	Действие по умолчанию	Значение
	Завершить+core	Сигнал отправляется, если процесс вызывает системный вызов <i>abort(2)</i> .
	Завершить	Сигнал отправляется, когда срабатывает таймер, ранее установленный с помощью системных вызовов <i>alarm(2)</i> или <i>setitimer(2)</i> .
	Завершить+core	Сигнал свидетельствует о некоторой аппаратной ошибке. Обычно этот сигнал отправляется при обращении к допустимому виртуальному адресу, для которого отсутствует соответствующая физическая страница. Другой случай генерации этого сигнала упоминался при обсуждении файлов, отображаемых в память (сигнал отправляется процессу при попытке обращения к странице виртуальной памяти, лежащей за пределами файла).
	Игнорировать	Сигнал, посыпаемый родительскому процессу при завершении выполнения его потомка.
	Завершить+core	Сигнал свидетельствует о попытке обращения к недопустимому адресу или к области памяти, для которой у процесса недостаточно привилегий.
	Завершить+core	Сигнал свидетельствует о возникновении особых ситуаций, таких как деление на 0 или переполнение операции с плавающей точкой.

Таблица 2.18 (продолжение)

Название	Действие по умолчанию	Значение
SIGHUP	Завершить	<p>Сигнал посыпается лидеру сеанса, связанному с управляющим терминалом, когда ядро обнаруживает, что терминал отсоединился (потеря линии). Сигнал также посыпается всем процессам текущей группы при завершении выполнения лидера.</p> <p>Этот сигнал иногда используется в качестве простейшего средства межпроцессного взаимодействия. В частности, он применяется для сообщения демонам о необходимости обновить конфигурационную информацию. Причина выбора именно сигнала SIGHUP заключается в том, что демон по определению не имеет управляющего терминала и, соответственно, обычно не получает этого сигнала.</p>
SIGILL	Завершить+core	Сигнал посыпается ядром, если процесс попытался выполнить недопустимую инструкцию.
SIGINT	Завершить	Сигнал посыпается ядром всем процессам текущей группы при нажатии клавиши прерывания (<Del> или <Ctrl>+<C>).
SIGKILL	Завершить	Сигнал, при получении которого выполнение процесса завершается. Этот сигнал нельзя ни перехватить, ни игнорировать.
SIGPIPE	Завершить	Сигнал посыпается при попытке записи в канал или сокет, получатель данных которого завершил выполнение (закрыл соответствующий дескриптор).
SIGPOLL	Завершить	Сигнал отправляется при наступлении определенного события для устройства, которое является опрашиваемым.
SIGPWR	Игнорировать	Сигнал генерируется при угрозе потери питания. Обычно он отправляется, когда питание системы переключается на источник бесперебойного питания (UPS).
SIGQUIT	Завершить+core	Сигнал посыпается ядром всем процессам текущей группы при нажатии клавиш <Ctrl>+< >.
SIGSTOP	Остановить	Сигнал отправляется всем процессам текущей группы при нажатии пользователем клавиш <Ctrl>+<Z>. Получение сигнала вызывает останов выполнения процесса.
	Завершить+core	Сигнал отправляется ядром при попытке недопустимого системного вызова.

Таблица 2.18 (окончание)

Название	Действие по умолчанию	Значение
SIGTERM	Завершить	Сигнал обычно представляет своего рода предупреждение, что процесс вскоре будет уничтожен. Этот сигнал позволяет процессу соответствующим образом "подготовиться к смерти" — удалить временные файлы, завершить необходимые транзакции и т. д. Команда <i>kill(1)</i> по умолчанию отправляет именно этот сигнал.
SIGTTIN	Остановить	Сигнал генерируется ядром (драйвером терминала) при попытке процесса фоновой группы осуществить чтение с управляющего терминала.
SIGTTOU	Остановить	Сигнал генерируется ядром (драйвером терминала) при попытке процесса фоновой группы осуществить запись на управляющий терминал.
SIGUSR1	Завершить	Сигнал предназначен для прикладных задач как простейшее средство межпроцессного взаимодействия.
SIGUSR2	Завершить	Сигнал предназначен для прикладных задач как простейшее средство межпроцессного взаимодействия.

Простейшим интерфейсом к сигналам UNIX является устаревшая, но по-прежнему поддерживаемая в большинстве систем функция *signal(3C)*. Эта функция позволяет изменить диспозицию сигнала, которая по умолчанию устанавливается ядром UNIX. Порожденный вызовом *fork(2)* процесс наследует диспозицию сигналов от своего родителя. Однако при вызове *exec(2)* диспозиция всех перехватываемых сигналов будет установлена на действие по умолчанию. Это вполне естественно, поскольку образ новой программы не содержит функции-обработчика, определенной диспозицией сигнала перед вызовом *exec(2)*. Функция *signal(3C)* имеет следующее определение:

```
#include <signal.h>
void (*signal (int sig, void (*disp) (int))) (int);
```

Аргумент *sig* определяет сигнал, диспозицию которого нужно изменить. Аргумент *disp* определяет новую диспозицию сигнала, которой может быть определенная пользователем функция-обработчик или одно из следующих значений:

SIG_DFL	Указывает ядру, что при получении процессом сигнала необходимо вызвать системный обработчик, т. е. выполнить действие по умолчанию.
SIG_IGN	Указывает, что сигнал следует игнорировать. Напомним, что не все сигналы можно игнорировать.

В случае успешного завершения *signal(3C)* возвращает предыдущую диспозицию — это может быть функция-обработчик сигнала или системные значения *SIG\_DFL* или *SIG\_IGN*. Возвращаемое значение может быть использовано для восстановления диспозиции в случае необходимости.

Использование функции *signal(3C)* подразумевает семантику устаревших или ненадежных сигналов. Процесс при этом имеет весьма слабые возможности управления сигналами. Во-первых, процесс не может заблокировать сигнал, т. е. отложить получение сигнала на период выполнения критического участка кода. Во-вторых, каждый раз при получении сигнала, его диспозиция устанавливается на действие по умолчанию. Данная функция и соответствующая ей семантика сохранены для поддержки старых версий приложений. В связи с этим в новых приложениях следует избегать использования функции *signal(3C)*. Тем не менее для простейшей иллюстрации использования сигналов, приведенный ниже пример использует именно этот интерфейс:

```
#include <signal.h>
/*Функция-обработчик сигнала*/
static void sig_hndlr(int signo)
{
/*Восстановим диспозицию*/
    signal(SIGINT, sig_hndlr);
    printf("Получен сигнал SIGINT\n");

main()
{
/*Установим диспозицию*/
    signal(SIGINT, sig_hndlr);
    signal(SIGUSR1, SIG_DFL);
    signal(SIGUSR2, SIG_IGN);
/*Бесконечный цикл*/
    while(1)
        pause();
}
```

В этом примере изменена диспозиция трех сигналов: *SIGINT*, *SIGUSR1* и *SIGUSR2*. При получении сигнала *SIGINT* вызывается обработчик *sig\_hndlr()*, при получении сигнала *SIGUSR1* производится действие по умолчанию (процесс завершает работу), а сигнал *SIGUSR2* игнорируется. После установки диспозиции сигналов процесс запускает бесконечный цикл, в процессе которого вызывается функция *pause(2)*. При получении сигнала, который не игнорируется, *pause(2)* возвращает значение *-1*, а переменная *errno* устанавливается равной *EINTR*. Заметим, что каждый раз при получении сигнала *SIGINT* мы вынуждены восстанавливать требуемую диспозицию, в противном случае получение следующего сигнала этого типа вызвало бы завершение выполнения процесса (действие по умолчанию).

При запуске программы, получим следующий результат:

```
$ a.out &
[1] 8365
$ kill -SIGINT 8365
Получен сигнал SIGINT
$ kill -SIGUSR2 8365
$ kill -SIGUSR1 8365
[1]+ User Signal 1
a.out
$
```

PID порожденного процесса

Сигнал SIGINT перехвачен

Сигнал SIGUSR2 игнорируется

**Сигнал SIGUSR1 вызывает завершение выполнения процесса**

Для отправления сигналов процессу использована команда *kill(1)*, описанная в предыдущей главе.

### Надежные сигналы

Стандарт POSIX.1 определил новый набор функций управления сигналами, основанный на интерфейсе 4.2BSD UNIX и лишенный рассмотренных выше недостатков.

Модель сигналов, предложенная POSIX, основана на понятии *набора сигналов* (signal set), описываемого переменной типа *sigset\_t*. Каждый бит этой переменной отвечает за один сигнал. Во многих системах тип *sigset\_t* имеет длину 32 бита, ограничивая количество возможных сигналов числом 32.

Следующие функции позволяют управлять наборами сигналов:

```
#include<signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(sigset_t *set, int signo);
```

В отличие от функции *signal(3C)*, изменяющей диспозицию сигналов, данные функции позволяют модифицировать структуру данных *sigset\_t*, определенную процессом. Для управления непосредственно сигналами используются дополнительные функции, которые мы рассмотрим позже.

Функция *sigemptyset(3C)* инициализирует набор, очищая все биты. Если процесс вызывает *sigfillset(3C)*, то набор будет включать все сигналы, известные системе. Функции *sigaddset(3C)* и *sigdelset(3C)* позволяют добавлять или удалять сигналы набора. Функция *sigismember(3C)* позволяет проверить, входит ли указанный параметром *signo* сигнал в набор.

Вместо функции *signal(3C)* стандарт POSIX.1 определяет функцию *sigaction(2)*, позволяющую установить диспозицию сигналов, узнать ее текущее значение или сделать и то и другое одновременно. Функция имеет следующее определение:

```
#include <signal.h>
int sigaction (int sig, const struct sigaction *act,
               struct sigaction *oact);
```

Вся необходимая для управлением сигналами информация передается через указатель на структуру `sigaction`, имеющую следующие поля:

<code>void (*sa_handler) ()</code>	Обработчик сигнала <code>sig</code>
<code>void (*sa_sigaction) (int, siginfo_t *, void *)</code>	Обработчик сигнала <code>sig</code> при установленном флаге <code>SA_SIGINFO</code>
<code>sigset_t sa_mask</code>	Маска сигналов
<code>int sa_flags</code>	Флаги

Поле `sa_handler` определяет действие, которое необходимо предпринять при получении сигналов, и может принимать значения `SIG_IGN`, `SIG_DFL` или адреса функции-обработчика. Если значение `sa_handler` или `sa_sigaction` не равны `NULL`, то в поле `sa_mask` передается набор сигналов, которые будут добавлены к маске сигналов перед вызовом обработчика. Каждый процесс имеет установленную маску сигналов, определяющую сигналы, доставка которых должна быть заблокирована. Если определенный бит маски установлен, соответствующий ему сигнал будет заблокирован. После возврата из функции-обработчика значение маски возвращается к исходному значению. Заметим, что сигнал, для которого установлена функция-обработчик, также будет заблокирован перед ее вызовом. Такой подход гарантирует, что во время обработки, последующее поступление определенных сигналов будет приостановлено до завершения функции. Как правило, UNIX не поддерживает очередей сигналов, и это значит, что блокировка нескольких однотипных сигналов в конечном итоге вызовет доставку лишь одного.

Поле `sa_flags` определяет флаги, модифицирующие доставку сигнала. Оно может принимать следующие значения:

<code>SA_ONSTACK</code>	Если определена функция-обработчик сигнала, и с помощью функции <code>sigaltstack(2)</code> задан альтернативный стек для функции-обработчика, то при обработке сигнала будет использоваться этот стек. Если флаг не установлен, будет использоваться обычный стек процесса.
<code>SA_RESETHAND*</code>	Если определена функция-обработчик, то диспозиция сигнала будет изменена на <code>SIG_DFL</code> , и сигнал не будет блокироваться при запуске обработчика. Если флаг не установлен, диспозиция сигнала остается неизменной.
<code>SA_NODEFER</code>	Если определена функция-обработчик, то сигнал блокируется на время обработки только в том случае, если он явно указан в поле <code>sa_mask</code> . Если флаг не установлен, в процессе обработки данный сигнал автоматически блокируется.

\* Данные флаги не определены для UNIX BSD.

SA_RESTART	Если определена функция-обработчик, ряд системных вызовов, выполнение которых было прервано полученным сигналом, будут автоматически перезапущены после обработки сигнала <sup>11</sup> . Если флаг не установлен, системный вызов возвратит ошибку EINTR.
SA_SIGINFO*	Если диспозиция указывает на перехват сигнала, вызывается функция, адресованная полем sa_sigaction. Если флаг не установлен, вызывается обработчик sa_handler.
SA_NOCLDWAIT*	Если указанный аргументом sig сигнал равен SIGCHLD, при завершении потомки не будут переходить в состояние зомби. Если процесс в дальнейшем вызовет функции wait(2), wait3(2), waitid(2) или waitpid(2), их выполнение будет блокировано до завершения работы всех потомков данного процесса.
SA_NOCLDSTOP*	Если указанный аргументом sig сигнал равен SIGCHLD, указанный сигнал не будет отправляться процессу при завершении или останове любого из его потомков.

В системах UNIX BSD 4.x структура sigaction имеет следующий вид:

```
struct sigaction {
    void (*sa_handler)();
    sigset_t    sa_mask;
    int    sa_flags;
};
```

где функция-обработчик определена следующим образом:

```
void handler(int signo, int code, struct sigcontext *scp);
```

В первом аргументе signo содержится номер сигнала, code определяет дополнительную информацию о причине поступления сигнала, а scp указывает на контекст процесса.

Для UNIX System V реализована следующая возможность получения более полной информации о сигнале. Если установлен флаг SA\_SIGINFO, то при получении сигнала sig будет вызван обработчик, адресованный полем sa\_sigaction. Помимо номера сигнала, обычно передаваемого обработчику сигнала, ему будет переданы указатель на структуру siginfo\_t, содержащую информацию о причинах получения сигнала, а также указатель на структуру ucontext\_t, содержащую контекст процесса.

Структура siginfo\_t определена в файле `<siginfo.h>` и включает следующие поля:

int si_signo	<b>Номер сигнала</b>
int si_errno	<b>Номер ошибки</b>
int si_code	<b>Причина отправления сигнала</b>

<sup>11</sup> К таким системным вызовам, в частности, относятся `read(2)` и `write(2)` для медленных устройств, таких как терминалы, а также `ioctl(2)`, `fcntl(2)`, `wait(2)` и `waitpid(2)`.

В поле `si_signo` хранится номер сигнала. Поле `si_code` имеет следующий смысл: если его значение меньше или равно нулю, значит сигнал был отправлен прикладным процессом, в этом случае структура `siginfo_t` содержит также следующие поля:

<code>pid_t si_pid</code>	Идентификатор процесса PID
<code>uid_t si_uid</code>	Идентификатор пользователя UID

которые адресуют процесс, пославший сигнал; если значение `si_code` больше нуля, то оно указывает на причину отправления сигнала. Список возможных значений `si_code` для некоторых сигналов, соответствующих полю `si_signo`, приведен в табл. 2.19

**Таблица 2.19.** Значения поля `si_code` структуры `siginfo_t` для некоторых сигналов

Значение поля <code>si_signo</code>	Значение поля <code>si_code</code>	Описание
<code>SIGILL</code>	Попытка выполнения недопустимой инструкции	
	<code>ILL_ILLOPC</code>	Недопустимый код операции (opcode)
	<code>ILL_ILLOPN</code>	Недопустимый операнд
	<code>ILL_ILLADR</code>	Недопустимый режим адресации
	<code>ILL_ILLTRP</code>	Недопустимая ловушка (trap)
	<code>ILL_PRVOPC</code>	Привилегированный код операции
	<code>ILL_PRVREG</code>	Привилегированный регистр
	<code>ILL_COPROC</code>	Ошибка сопроцессора
	<code>ILL_BADSTK</code>	Ошибка внутреннего стека
<code>SIGFPE</code>	Особая ситуация операции с плавающей точкой	
	<code>FPE_INTDIV</code>	Целочисленное деление на ноль
	<code>FPE_INTOVF</code>	Целочисленное переполнение
	<code>FPE_FLTDIV</code>	Деление на ноль с плавающей точкой
	<code>FPE_FLTOVF</code>	Переполнение с плавающей точкой
	<code>FPE_FLTUND</code>	Потеря точности с плавающей точкой (underflow)
	<code>FPE_FLTRES</code>	Неоднозначный результат операции с плавающей точкой
	<code>FPE_FLTINV</code>	Недопустимая операция с плавающей точкой
	<code>FPE_FLTSUB</code>	Индекс вне диапазона

Таблица 2.19 (продолжение)

Значение поля si_signo	Значение поля si_code	Описание
SIGSEGV	Нарушение сегментации	
	SEGV MAPPER	Адрес не отображается на объект
	SEGV ACCERR	Недостаточно прав на отображаемый объект
SIGBUS	Ошибка адресации	
	BUSADR ALN	Недопустимое выравнивание адреса
	BUSADRERR	Несуществующий физический адрес
	BUSOBJERR	Аппаратная ошибка, связанная с объектом
SIGTRAP	Ловушка	
	TRAP_BRKPT	Процессом достигнута точка останова
	TRAP_TRACE	Ловушка трассирования процесса
SIGCHLD	Завершение выполнения дочернего процесса	
	CLD_EXITED	Дочерний процесс завершил выполнение
	CLD_KILLED	Дочерний процесс был "убит"
	CLD_DUMPED	Ненормальное завершение дочернего процесса
	CLD_TRAPPED	Трассируемый дочерний процесс находится в ловушке
	CLD_STOPPED	Выполнение дочернего процесса было остановлено
	CLD_CONTINUED	Выполнение остановленного дочернего процесса было продолжено
SIGPOLL	Событие на опрашиваемом устройстве	
	POLL_IN	Поступили данные для ввода
	POLL_OUT	Свободны буферы данных
	POLL_MSG	Сообщение ожидает ввода
	POLL_ERR	Ошибка ввода/вывода
	POLL_PRI	Высокоприоритетные данные ожидают ввода
	POLL_HUP	Устройство отключено

Уже отмечалось, что при получении сигнала от пользовательского процесса структура `siginfo_t` содержит дополнительные поля (табл. 2.20).

**Таблица 2.20.** Дополнительные поля структуры `siginfo_t`

Значение поля <code>si_signo</code>	Дополнительные поля		Значение
<code>SIGILL</code>	<code>caddr_t</code>	<code>si_addr</code>	Адрес недопустимой инструкции
<code>SIGFPE</code>			
<code>SIGSEGV</code>	<code>caddr_t</code>	<code>si_addr</code>	Адрес недопустимой области памяти
<code>SIGBUS</code>			
<code>SIGCHLD</code>	<code>pid_t</code> <code>int</code>	<code>si_pid</code> <code>si_status</code>	Идентификатор дочернего процесса Код возврата сигнала
<code>SIGPOLL</code>	<code>long</code>	<code>si_band</code>	Ошибка канала (для модулей STREAMS)

Установить маску сигналов или получить текущую маску можно с помощью функции `sigprocmask(2)`:

```
#include <signal.h>
int sigprocmask(int how, sigset_t *set, sigset_t *oset);
```

Маска сигналов изменяется в соответствии с аргументом `how`, который может принимать следующие значения:

<code>SIG_UNBLOCK</code>	Результирующая маска получится путем объединения текущей маски и набора <code>set</code>
<code>SIG_SETMASK</code>	Сигналы набора <code>set</code> будут удалены из текущей маски
	Текущая маска будет заменена на набор <code>set</code>

Если указатель `set` равен `NULL`, то аргумент `how` игнорируется. Если аргумент `oset` не равен `NULL`, то в набор, адресованный этим аргументом, помещается текущая маска сигналов.

Функция `sigpending(2)` используется для получения набора заблокированных сигналов, ожидающих доставки:

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Список сигналов, ожидающих доставки, возвращается в наборе, адресованном аргументом `set`.

Системный вызов `sigsuspend(2)` замещает текущую маску набором, адресованным аргументом `set`, и приостанавливает выполнение процесса до получения сигналов, диспозиция которых установлена либо на завершение выполнения процесса, либо на вызов функции-обработчика сигнала.

```
ttinclude <signal.h>
int sighuspend(const sigset_t *set);
```

При получении сигнала, завершающего выполнение процесса, возврата из функции *sigsuspend(2)* не происходит. Если же диспозиция полученного сигнала установлена на вызов функции-обработчика, возврат из *sigsuspend(2)* происходит сразу после завершения обработки сигнала. При этом восстанавливается маска, существовавшая до вызова *sigsuspend(2)*.

Заметим, что в BSD UNIX вызов *signal(3)* является упрощенным интерфейсом к более общей функции *sigaction(2)*, в то время как в ветви System V *signal(3)* подразумевает использование старой семантики ненадежных сигналов.

В заключение для иллюстрации изложенных соображений, приведем версию функции *signal()*, позволяющую использовать надежные сигналы. Похожая реализация используется в BSD UNIX. С помощью этой "надежной" версии мы повторим пример, рассмотренный нами выше, в измененном виде.

```
ttinclude <signal.h>
ttinclude <sys/types.h>
ttinclude <sys/stat.h>
ttinclude <fcntl.h>
ttinclude <unistd.h>
/* Вариант "надежной" функции signal() */
void (*mysignal (int signo, void (*hndlr) (int))) (int)
{
    struct sigaction    act, oact;
    /* Установим маску сигналов */
    act.sa_handler = hndlr;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo != SIGALRM)
        act.sa_flags |= SA_RESTART;
    /* Установим диспозицию */
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
/* Функция-обработчик сигнала */
static void sig_hndlr(int signo)
{
    /* Эта часть кода нам уже не нужна
    mysignal (SIGINT, sig_hndlr);
    */
    printf ("Получен сигнал SIGINT\n");
}
```

```
/*Установим диспозицию*/
mysignal(SIGINT, sig_hndlr);
mysignal(SIGUSR1, SIG_DFL);
mysignal(SIGUSR2, SIG_IGN);
/*Бесконечный цикл*/
while(1)
    pause();
}
```

Заметим, что при использовании надежных сигналов, не нужно восстанавливать диспозицию в функции-обработчике при получении сигнала.

## Группы и сеансы

После создания процесса ему присваивается уникальный идентификатор, возвращаемый системным вызовом *fork(2)* родительскому процессу. Дополнительно ядро назначает процессу *идентификатор группы процессов* (process group ID). *Группа процессов* включает один или более процессов и существует, пока в системе присутствует хотя бы один процесс этой группы. Временной интервал, начинающийся с создания группы и заканчивающийся, когда последний процесс ее покинет, называется *временем жизни группы*. Последний процесс может либо завершить свое выполнение, либо перейти в другую группу.

Многие системные вызовы могут быть применены как к единичному процессу, так и ко всем процессам группы. Например, системный вызов *kill(2)* может отправить сигнал как одному процессу, так и всем процессам указанной группы. Точно так же функция *waitpid(2)* позволяет родительскому процессу ожидать завершения конкретного процесса или любого процесса группы.

Каждый процесс, помимо этого, является членом *сеанса* (session), являющегося набором одной или нескольких групп процессов. Понятие сеанса было введено в UNIX для логического объединения процессов, а точнее, групп процессов, созданных в результате регистрации и последующей работы пользователя в системе. Таким образом, термин "сеанс работы" в системе тесно связан с понятием сеанса, описывающего набор процессов, которые порождены пользователем за время пребывания в системе.

Процесс имеет возможность определить идентификатор собственной группы процессов или группы процесса, который является членом того же сеанса. Для этого используются два системных вызова: *getpgrp(2)* и *getpgid(2)*:

```
ttinclude<sys/types.h>
ttinclude<unistd.h>
pid_t getpgrp(void);
pid_t getpgid(pid_t pid);
```

Аргумент `pid`, который передается функции `getpgid(2)`, адресует процесс, идентификатор группы которого требуется узнать. Если этот процесс не принадлежит к тому же сеансу, что и процесс, сделавший системный вызов, функция возвращает ошибку.

Системный вызов `setpgid(2)` позволяет процессу стать членом существующей группы или создать новую группу.

```
#include <sys/types.h>
#include <unistd.h>

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

Функция устанавливает идентификатор группы процесса `pid` равным `pgid`. Процесс имеет возможность установить идентификатор группы для себя и для своих потомков (дочерних процессов). Однако процесс не может изменить идентификатор группы для дочернего процесса, который выполнил системный вызов `exec(2)`, запускающий на выполнение другую программу.

Если значения обоих аргументов равны, то создается новая группа с идентификатором `pgid`, а процесс становится *лидером* (group leader) этой группы. Поскольку именно таким образом создаются новые группы, их идентификаторы гарантированно уникальны. Заметим, что группа не удаляется при завершении ее лидера, пока в нее входит хотя бы один процесс.

Идентификатор сеанса можно узнать с помощью функции `getsid(2)`:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getsid(pid_t pid);
```

Как и в случае с группой, идентификатор `pid` должен адресовать процесс, являющийся членом того же сеанса, что и процесс, вызвавший `getsid(2)`. Заметим, что эти ограничения не распространяются на процессы, имеющие привилегии суперпользователя.

Вызов функции `setsid(2)` приводит к созданию нового сеанса:

```
#include <sys/types.h>
#include <unistd.h>

pid_t setsid(void);
```

Новый сеанс создается лишь при условии, что процесс не является лидером какого-либо сеанса. В случае успеха процесс становится лидером сеанса и лидером новой группы.

Понятия группы и сеанса тесно связаны с терминалом или, точнее, с драйвером терминала. Каждый сеанс может иметь один ассоциированный терминал, который называется *управляющим терминалом* (controlling terminal), а группы, созданные в данном сеансе, наследуют этот управ-

ляющий терминал. Наличие управляющего терминала позволяет ядру контролировать стандартный ввод/вывод процессов, а также дает возможность отправить сигнал всем процессам ассоциированной с терминалом группы, например, при его отключении. Типичным примером является регистрация и работа пользователя в системе. При входе в систему терминал пользователя становится управляющим для лидера сеанса (в данном случае для командного интерпретатора `shell`) и всех процессов, порожденных лидером (в данном случае для всех процессов, которые запускает пользователь из командной строки интерпретатора). При выходе пользователя из системы `shell` завершает свою работу и таким образом отключается от управляющего терминала, что вызывает отправление сигнала `SIGHUP` всем незавершенным процессам текущей группы. Это гарантирует, что после завершения работы пользователя в системе не останется запущенных им процессов<sup>12</sup>.

## Текущие и фоновые группы процессов

Как было показано, для каждого управляющего терминала существует сеанс, включающий одну или несколько групп процессов. Одна из этих групп является *текущей* (foreground group), а остальные *фоновыми* (background group)<sup>13</sup>. Сигналы `SIGINT` и `SIGQUIT`, которые генерируются драйвером терминала, посылаются всем процессам текущей группы. Попытка процессов фоновых групп осуществить доступ к управляющему терминалу, как правило, вызывает отправление им сигналов `SIGSTOP`, `SIGTTIN` ИЛИ `SIGTTOUT`.

Рассмотрим следующие команды:

```
$ find / -name foo &
$ cat | sort
```

При этом происходит чтение ввода пользователя с клавиатуры (`cat(1)`) и сортировка введенных данных (`sort(1)`). Если интерпретатор поддерживает управление заданиями, оба процесса, созданные для программ `cat(1)` и `sort(1)`, будут помещены в отдельную группу. Это подтверждается выводом команды `ps(1)`:

<sup>12</sup> Тем не менее в системе будут продолжать выполняться процессы, запущенные в фоновом режиме. Это утверждение также не справедливо для демонов — процессов, являющихся членами сеанса, не имеющего управляющего терминала. Система не имеет возможности автоматического отправления сигнала `SIGHUP` таким процессам при выходе пользователя, и они будут продолжать выполняться даже после завершения пользователем работы в **UNIX**. Для "превращения" процесса в демона, он должен воспользоваться функцией `setsid(2)` и создать новый сеанс, лидером которого он автоматически окажется и который не будет ассоциирован с управляющим терминалом. Эти вопросы будут более подробно обсуждены при иллюстрации программы-демона далее в этой главе.

<sup>13</sup> Наличие текущей и фоновых групп процессов в сеансе работы пользователя зависит от возможности командного интерпретатора управлять заданиями (job control). При отсутствии этой возможности все процессы будут выполняться в той же группе, что и `shell`.

```
$ ps -efj | egrep "PID|andy"
UID  PID  PPID  PGID  SID  C  STIME    TTY      TIME  CMD
andy 2436 2407 2435 2407 1 15:51:30  tty01  0:00  sort
andy 2431 2407 2431 2407 0 15:51:25  tty01  0:00  find / -name foo
andy 2407 2405 2407 2407 0 15:31:09  tty01  0:00  -sh
andy 2435 2407 2435 2407 0 15:51:30  tty01  0:00  cat
```

Все четыре процесса (*sh*, *find*, *cat* и *sort*) имеют один и тот же идентификатор сеанса, связанного с управляющим терминалом *tty01*. Процессы *cat(1)* и *sort(1)* принадлежат одной группе, идентификатор которой (2435) отличен от идентификатора группы командного интерпретатора (2407). То же самое можно сказать и о процессе *find(l)*, который является лидером отдельной группы (2431). Можно также заметить, что процессы *sh(1)*, *find(1)* и *cat(1)* являются лидерами групп, а *sh(1)* еще и лидером сеанса.

Хотя команда *ps(l)* не указывает, какие группы являются фоновыми, а какая текущей, синтаксис команд позволяет утверждать, что командный интерпретатор помещает *cat(l)* и *sort(l)* в текущую группу. Это, во-первых, позволяет процессу *cat(l)* читать данные со стандартного потока ввода, связанного с терминалом *tty01*. Во-вторых, пользователь имеет возможность завершить выполнение обоих процессов путем нажатия клавиши *<Del>* (или *<Ctrl>+<C>*), что вызовет генерацию сигнала *SIGINT*. Получение процессами этого сигнала вызовет завершение их выполнения (действие по умолчанию), если, конечно, процесс не установил игнорирование *SIGINT*. На рис. 2.13 представлена схема взаимодействия управляющего терминала, сеанса и групп процессов для приведенного выше примера. Более детально взаимосвязь между терминалом и процессами рассмотрена в следующей главе.

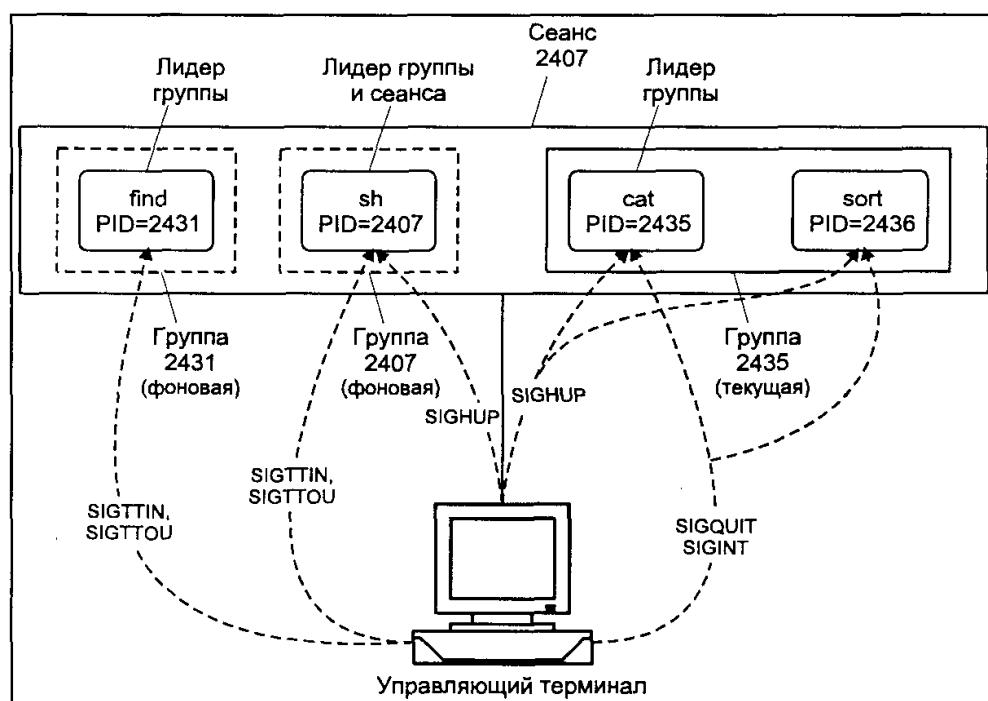


Рис. 2.13. Связь между управляющим терминалом, сеансом и группами

Если командный интерпретатор не поддерживает управление заданиями, оба процесса станут членами той же группы, что и сам shell. В этом случае командный интерпретатор должен позаботиться об игнорировании сигналов SIGINT и SIGQUIT, чтобы допустимые действия пользователя (такие как нажатие клавиши *<Del>* или *<Ctrl>+<C>*) не привели к завершению выполнения shell и выходу из системы.

## Ограничения

UNIX является многозадачной системой. Это значит, что несколько процессов конкурируют между собой при доступе к различным ресурсам. Для "справедливого" распределения разделяемых ресурсов, таких как память, дисковое пространство и т. п., каждому процессу установлен набор ограничений. Эти ограничения не носят общесистемного характера, как, например, максимальное число процессов или областей, а устанавливаются для каждого процесса отдельно. Для получения информации о текущих ограничениях и их изменения пред назначены системные вызовы *getrlimit(2)* и *setrlimit(2)*:

```
#include <sys/time.h>
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, const struct rlimit *rlp);
```

Аргумент *resource* определяет вид ресурса, для которого мы хотим узнать или изменить ограничения процесса. Структура *rlimit* состоит из двух полей:

```
rlim_t rlim_cur;
rlim_t rlim_max;
```

определяющих, соответственно, *изменяемое* (soft) и *жесткое* (hard) ограничение. Первое определяет текущее ограничение процесса на данный ресурс, а второе — максимальный возможный предел потребления ресурса. Например, изменяемое ограничение на число открытых процессом файлов может составлять 64, в то время как жесткое ограничение равно 1024.

Любой процесс может изменить значение текущего ограничения вплоть до максимально возможного предела. Жесткое ограничение может быть изменено в сторону увеличения предела потребления ресурса только процессом с привилегиями суперпользователя. Обычные процессы могут только уменьшить значение жесткого ограничения. Обычно ограничения устанавливаются при инициализации системы и затем наследуются порожденными процессами (хотя в дальнейшем могут быть изменены).

Вообще говоря, максимальный возможный предел потребления ресурса может иметь бесконечное значение. Для этого необходимо установить значение *rlim\_max* равным **RLIM\_INFINITY**. В этом случае физические ограничения системы (например, объем памяти и дискового пространства) будут определять реальный предел использования того или иного ресурса.

Различные ограничения и связанные с ними типы ресурсов приведены в табл. 2.21.

**Таблица 2.21.** Ограничения процесса (значения аргумента resource)

Ограничение	Тип ресурса	Эффект
RLIMIT_CORE	Максимальный размер создаваемого файла core, содержащего образ памяти процесса. Если предел установлен равным 0, файл core создаваться не будет.	После создания файла core запись в этот файл будет остановлена при достижении предельного размера.
RLIMIT_CPU	Максимальное время использования процессора в секундах.	При превышении предела процессу отправляется сигнал SIGXCPU.
RLIMIT_DATA	Максимальный размер сегмента данных процесса в байтах, т. е. максимальное значение смещения брейк-адреса.	При достижении этого предела последующие вызовы функции <i>brk(2)</i> завершатся с ошибкой ENOMEM.
RLIMIT_FSIZE	Максимальный размер файла, который может создать процесс. Если значение этого предела равно 0, процесс не может создавать файлы.	При достижении этого предела процессу отправляется сигнал SIGXFSZ. Если сигнал перехватывается или игнорируется процессом, последующие попытки увеличить размер файла закончатся С Ошибкой EFBIG.
RLIMIT_NOFILE	Максимальное количество назначенных файловых дескрипторов процесса.	При достижении этого предела, последующие попытки получить новый файловый дескриптор закончатся с ошибкой EMFILE.
RLIMIT_STACK	Максимальный размер стека процесса.	При попытке расширить стек за установленный предел отправляется сигнал SIGSEGV. Если процесс перехватывает или игнорирует сигнал и не использует альтернативный стек с помощью функции <i>sigaltstack(2)</i> , диспозиция сигнала устанавливается на действие по умолчанию перед отправкой процессу.
RLIMIT_VMEM	Максимальный размер отображаемой памяти процесса в байтах. (Предел определен в версиях System V.)	При достижении этого предела последующие вызовы <i>brk(2)</i> или <i>mmap(2)</i> завершатся с ошибкой ENOMEM.
RLIMIT_NPROC	Максимальное число процессов с одним реальным UID. Определяет максимальное число процессов, которые может запустить пользователь. (Предел определен в версиях BSD UNIX.)	При достижении этого предела, последующие вызовы <i>fork(2)</i> для порождения нового процесса завершатся с ошибкой EAGAIN.

Таблица 2.21 (окончание)

Ограничение	Тип ресурса	Эффект
RLIMIT_RSS	Максимальный размер в байтах резидентной части процесса (RSS — Resident Set Size). Определяет максимальное количество физической памяти, предоставляемой процессу. (Предел определен в версиях BSD UNIX.)	Если система ощущает недостаток памяти, ядро освободит память за счет процессов, превысивших свой RSS.
RLIMIT_MEMLOCK	Максимальный объем физической памяти (физических страниц) в байтах, который процесс может заблокировать с помощью системного вызова <i>mlock(2)</i> . (Предел определен в версиях BSD UNIX.)	При превышении предела системный вызов <i>mlock(2)</i> завершится с ошибкой EAGAIN.

В заключение приведем пример программы, выводящий на экран установленные ограничения для процесса:

```
#include <sys/types.h>
#include <sys/resource.h>
/*Процедура вывода на экран текущего и максимального пределов
потребления ресурса resource*/
void disp_limit(int resource, char *rname)
{
    struct rlimit rlm;
    getrlimit(resource, &rlm);
    printf("%-13s ", rname);
    /*Значение изменяемого ограничения*/
    if (rlm.rlim_cur == RLIM_INFINITY)
        printf("infinite    ");
    else
        printf ("%10ld ", rlm.rlim_cur);
    /*Значение жесткого ограничения*/
    if (rlm.rlim_max == RLIM_INFINITY)
        printf ("infinite \n");
    else
        printf("%10ld\n", rlm.rlim_max);
}

disp_limit (RLIMIT_CORE, "RLIMIT_CORE");
disp_limit(RLIMIT_CPU, "RLIMIT_CPU");
disp_limit(RLIMIT_DATA, "RLIMIT_DATA");
disp_limit(RLIMIT_FSIZE, "RLIMIT_FSIZE");
disp_limit(RLIMIT_NOFILE, "RLIMIT_NOFILE");
disp_limit(RLIMIT_STACK, "RLIMIT_STACK");
```

```

/* BSD */
#ifndef RLIMIT_NPROC
    disp_limit(RLIMIT_NPROC, "RLIMIT_NPROC");
#endif
/* BSD */
#ifndef RLIMIT_RSS
    disp_limit(RLIMIT_RSS, "RLIMIT_RSS");
#endif
/* BSD */
#ifndef RLIMIT_MEMLOCK
    disp_limit(RLIMIT_MEMLOCK, "RLIMIT_MEMLOCK");
#endif
/* System V */
#ifndef RLIMIT_VMEM
    disp_limit(RLIMIT_VMEM, "RLIMIT_VMEM");
#endif
}

```

Запуск программы под управлением операционной системы Solaris 2.5 даст следующие результаты:

```

$ a.out
RLIMIT_CORE      infinite      infinite
RLIMIT_CPU       infinite      infinite
RLIMIT_DATA      2147479552   2147479552
RLIMIT_FSIZE     infinite      infinite
RLIMIT_NOFILE    64           1024
RLIMIT_STACK     8388608     2147479552
RLIMIT_VMEM      infinite      infinite

```

## Примеры программ

В качестве заключительной иллюстрации к обсуждавшимся выше вопросам приводятся фрагменты двух приложений, которые в достаточной степени демонстрируют практическое применение программного интерфейса UNIX. Заметим, что приведенные примеры не являются законченными программами — во многих местах участки кода намеренно опущены, а функциональность сведена к минимуму. Задачей являлось показать принцип взаимодействия программ с операционной системой и идеологию программирования в UNIX. Рассмотрим два диаметрально противоположных приложения — неинтерактивную программу-демон и интерактивный командный интерпретатор.

## Демон

Демоны играют важную роль в работе операционной системы. Достаточно будет сказать, что возможность терминального входа пользователей в сис-

тему, доступ по сети, использование системы печати и электронной почты, — все это обеспечивается соответствующими демонами — неинтерактивными программами, составляющими собственные сеансы (и группы) и не принадлежащими ни одному из пользовательских сеансов (групп).

Некоторые демоны работают постоянно, наиболее яркий пример такого демона — процесс *init(1M)*, являющийся прародителем всех прикладных процессов в системе. Другими примерами являются *cron(1M)*, позволяющий запускать программы в определенные моменты времени, *inetd(1M)*, обеспечивающий доступ к сервисам системы из сети, и *sendmail(1M)*, обеспечивающий получение и отправку электронной почты.

При описании взаимодействия процессов с терминалом и пользователем в разделе "Группы и сеансы", отмечалось особое место демонов, которые не имеют управляющего терминала. Теперь в отношении демонов можно сформулировать ряд правил, определяющих их нормальное функционирование, которые необходимо учитывать при разработке таких программ:

1. Демон не должен реагировать на сигналы управления заданиями, посылаемые ему при попытке операций ввода/вывода с управляющим терминалом. Начиная с некоторого времени, демон снимает ассоциацию с управляющим терминалом, но на начальном этапе запуска ему может потребоваться вывести то или иное сообщение на экран.
2. Необходимо закрыть все открытые файлы (файловые дескрипторы), особенно стандартные потоки ввода/вывода. Многие из этих файлов представляют собой терминальные устройства, которые должны быть закрыты, например, при выходе пользователя из системы. Предполагается, что демон остается работать и после того, как пользователь "покинул" UNIX.
3. Необходимо снять его ассоциацию с группой процессов и управляющим терминалом. Это позволит демону избавиться от сигналов, генерируемых терминалом (SIGINT или SIGHUP), например, при нажатии определенных клавиш или выходе пользователя из системы.
4. Сообщения о работе демона следует направлять в специальный журнал с помощью функции *syslog(3)*, — это наиболее корректный способ передачи сообщений от демона.
5. Необходимо изменить текущий каталог на корневой. Если этого не сделать, а текущий каталог, допустим, находится на примонтированной файловой системе, последнюю нельзя будет размонтировать. Самым надежным выбором является корневой каталог, всегда принадлежащий корневой файловой системе.

Приведем скелет программы-демона:

```
#include <stdio.h>
#include <syslog.h>
#include <signal.h>
#include <sys/types.h>
```

```

#include <sys/param.h>
#include <sys/resource.h>

main(int argc, char **argv)
{
    int fd;
    struct rlimit      flim;
    /*Если родительский процесс – init, можно не беспокоиться за
    терминальные сигналы. Если нет – необходимо игнорировать сигналы,
    связанные с вводом/выводом на терминал фонового процесса:
    SIGTTOU, SIGTTIN, SIGTSTP*/
    if (getppid() != 1)
    {
        signal(SIGTTOU, SIG_IGN);
        signal(SIGTTIN, SIG_IGN);
        signal(SIGTSTP, SIG_IGN);

        /*Теперь необходимо организовать собственную группу и сеанс, не
        имеющие управляющего терминала. Однако лидером группы и сеанса
        может стать процесс, если он еще не является лидером. Поскольку
        предыстория запуска данной программы неизвестна, необходима га-
        рантия, что наш процесс не является лидером. Для этого порождаем
        дочерний процесс. Т. к. его PID уникален, то ни группы, ни сеанса
        с таким идентификатором не существует, а значит нет и лидера. При
        этом родительский процесс немедленно завершает выполнение, по-
        скольку он уже не нужен.

        Существует еще одна причина необходимости порождения дочернего
        процесса. Если демон был запущен из командной строки командного
        интерпретатора shell не в фоновом режиме, последний будет ожидать
        завершения выполнения демона, и таким образом, терминал будет за-
        блокирован. Порождая процесс и завершая выполнение родителя, ими-
        тируем для командного интерпретатора завершение работы демона,
        после чего shell выведет свое приглашение.*/
        if (fork() !=0)
            exit(0); /*Родитель заканчивает работу*/
        /*Дочерний процесс с помощью системного вызова setsid(2)
        становится лидером новой группы, сеанса и не имеет
        ассоциированного терминала14/
        setsid();
    }
    /*Теперь необходимо закрыть открытые файлы. Закроем все возможные
    файловые дескрипторы. Максимальное число открытых файлов получим
    с помощью функции getrlimit(2)*/
    getrlimit(RLIMIT_NOFILE, &flim);
    for (fd = 0; fd < flim.rlim_max; fd++)
        close(fd);
    /*Сменим текущий каталог на корневой*/
    chdir("/");
}

```

<sup>14</sup> Использование вызова *setsid(2)* справедливо для UNIX System V. Для BSD UNIX процесс должен последовательно создать группу, лидером которой он становится, а затем открыть управляющий терминал и с помощью команды *ioctl(2)* TIOCNOTTY отключиться от него.

```

/*Заявим о себе в системном журнале. Для этого сначала установим
опции ведения журнала: каждая запись будет предваряться идентифи-
катором PID демона, при невозможности записи в журнал сообщения
будут выводиться на консоль, источник сообщений определим как
"системный демон" (см. комментарии к функциям ведения журнала ни-
же).*/
    openlog("Скелет демона", LOG_PID | LOG_CONS, LOG_DAEMON);
/*Отметимся/
    syslog(LOG_INFO, "Демон начал плодотворную работу...");
    closelog();

/*Далее следует текст программы, реализующий полезные функции
демона. Эта часть предоставляется читателю для собственной
разработки.*/
}

}

```

В программе использовалось еще не обсуждавшаяся возможность системного журнала сообщений выполняющихся программ. Функцией генерации сообщений является *syslog(3)*, отправляющая сообщение демону системного журнала *syslogd(1M)*, который в свою очередь либо дописывает сообщения в системный журнал, либо выводит на их консоль, либо перенаправляет в соответствии со списком пользователей данной или удаленной системы. Конкретный пункт назначения определяется конфигурационным файлом (*/etc/syslog.conf*). Функция имеет определение:

```

#include <syslog.h>
void syslog (int priority, char *logstring, /* параметры*/...);

```

Каждому сообщению *logstring* назначается приоритет, указанный параметром *priority*. Возможные значения этого параметра включают:

LOG_EMERG	Идентифицирует состояние "паники" в системе. Обычно рассы- ляется всем пользователям.
LOG_ALERT	Идентифицирует ненормальное состояние, которое должно быть исправлено немедленно, например, нарушение целостно- сти системной базы данных.
LOG_CRIT	Идентифицирует критическое событие, например, ошибку дис- кового устройства.
LOG_ERR	Идентифицирует различные ошибки.
LOG_WARNING	Идентифицирует предупреждения.
LOG_NOTICE	Идентифицирует события, которые не являются ошибками, но требуют внимания.
LOG_INFO	Идентифицирует информационные сообщения, как, например, использованное в приведенной программе.
LOG_DEBUG	Идентифицирует сообщение, обычно используемое только при отладке программы.

Последний тип сообщений подсказывает еще одну возможность использования системного журнала — для отладки программ, особенно неинтерактивных.

Строка `logstring` может включать элементы форматирования, такие же, как и в функции `printf(3)`, с одним дополнительным выражением `%m`, которое заменяется сообщением, соответствующим ошибке `errno`. При этом может осуществляться вывод значений дополнительных параметров.

Функция `openlog(3)` позволяет определить ряд опций ведения журнала. Она имеет следующее определение:

```
void openlog(char *ident, int logopt, int facility);
```

Строка `ident` будет предшествовать каждому сообщению программы. Аргумент `logopt` задает дополнительные опции, в том числе:

<code>LOG_PID</code>	Позволяет указывать идентификатор процесса в каждом сообщении. Эта опция полезна при журналировании нескольких демонов с одним и тем же значением <code>ident</code> , например, когда демоны порождаются вызовом <code>fork(2)</code> .
<code>LOG_CONS</code>	Позволяет выводить сообщения на консоль при невозможности записи в журнал.

Наконец, аргумент `facility` позволяет определить источник сообщений:

<code>LOG_KERN</code>	Указывает, что сообщения отправляются ядром.
<code>LOG_USER</code>	Указывает, что сообщения отправлены прикладным процессом (используется по умолчанию).
<code>LOG_MAIL</code>	Указывает, что инициатором сообщений является система электронной почты.
<code>LOG_DAEMON</code>	Указывает, что инициатором сообщений является системный демон.
<code>LOG_NEWS</code>	Указывает, что инициатором сообщений является система телеконференций USENET.
<code>LOG_CRON</code>	Указывает, что инициатором сообщений является система <code>cron(1)</code> .

Закончив работу с журналом, следует аккуратно закрыть его с помощью функции `closelog(3)`:

```
void closelog (void);
```

## Командный интерпретатор

Для примера интерактивного приложения, мы выбрали простейший командный интерпретатор. Данный пример позволяет продемонстрировать использование системных вызовов для порождения процесса, запуска программы и синхронизации выполнения процессов.

Функции приведенного командного интерпретатора сведены к минимуму: он распознает и выполняет несколько встроенных команд, остальной ввод

он расценивает как внешние программы, которые и пытаются запустить с помощью системного вызова *exec(2)*.

```
#include <sys/types.h>
#include<sys/wait.h>
#include <unistd.h>
extern char ** environ;
#define CMDSIZE 80
/*Встроенные команды интерпретатора*/
#define CD 1
#define ECHO 2
#define EXEC 3

define PROGRAM 1000

/*Функция, которая производит анализ строки, введенной
пользователем, выполняет подстановки и определяет, встроенная ли
это команда или программа. В качестве аргумента функция принимает
строку cmdbuf, введенную пользователем, и возвращает имя
команды/программы path и переданные ей параметры arguments.
Возвращаемое значение указывает на внутреннюю команду или внешнюю
программу, которую необходимо запустить.*/
int parse_command(char *cmdbuf, char *path, char **arguments);

main ()
{
char cmd[CMDSIZE];
int command;
int stat_loc;
char **args;
char cmdpath[MAXPATH];
while (1)
{
/*Выведем сообщение интерпретатора*/
    write (1, "$ ", 2);
/*Считаем ввод пользователя и проанализируем строку*/
    cmdsize = read(0, cmd, CMDSIZE);
    cmd[cmdsize-1] = '\0';
    command = parse_command(cmd, cmdpath, args);
    switch(command)
    {
/*Если это внутренняя команда, обработаем ее*/
        case(CD) : chdir(args[0]); break;
        case(ECHO):
            write (1, args[0], strlen(args[0]));
            break;
        case(EXEC):
            execve(path, args, environ);
            write (2, "shell: cannot execute",
                  21);
            break;
    }
}
```

```

/*Если это внешняя программа, создадим дочерний процесс, который
и запустит программу*/
    case (PROGRAM) :
        pid = fork();
        if (pid < 0)
            write(2, "shell: cannot fork",
                  18);
        else if (pid == 0)
        {
/*Дочерний процесс*/
            execve(path, args, environ);
            write(2,
                  "shell: cannot execute",
                  21);
        }
        else
/*Родительский процесс*/
/*Ожидаем завершения выполнения программы*/
            wait(&stat_loc);
        break;
    }
}
}

```

Предложенный командный интерпретатор работает в бесконечном цикле, запрашивая ввод пользователя и анализируя строку с помощью функции `parse_command()`, текст которой здесь не приведен. В случае, если пользователь ввел встроенную команду интерпретатора, он выполняет команду собственными силами. В противном случае shell порождает дочерний процесс, который с помощью вызова `execve(2)` запускает указанную программу. В это время родительский процесс выполняет системный вызов `wait(2)` и приостанавливает свое выполнение до завершения работы программы, после чего на экран вновь выводится приглашение.

## Заключение

Изначально система UNIX создавалась как среда разработки программ. Хотя сегодня UNIX применяется во многих областях, не связанных с разработкой программного обеспечения, эта операционная система по-прежнему пользуется большой популярностью среди программистов. В этой главе рассмотрены уже известные подсистемы операционной системы с точки зрения их программного интерфейса. В первую очередь — это интерфейс системных вызовов, определяющий базовые услуги, предоставляемые ядром системы прикладным процессам. При обсуждении вопросов, связанных с программированием в UNIX были проиллюстрированы отдельные положения фрагментами программ, написанными на языке С — стандартном языке UNIX, на котором написаны ядро и основные утилиты системы.



## Подсистема управления процессами

Сердцем операционной системы UNIX является подсистема управления процессами. Практически все действия ядра имеют отношение к процессам, будь то обслуживание системного вызова, генерация сигнала, размещение памяти, обработка особых ситуаций, вызванных выполнением процесса или обеспечением услуг ввода/вывода по запросу прикладного процесса.

Вся функциональность операционной системы в конечном счете определяется выполнением тех или иных процессов. Даже так называемые *уровни выполнения системы* (run levels) представляют собой ни что иное, как удобную форму определения группы выполняющихся процессов. Возможность терминального или сетевого доступа к системе, различные сервисы, традиционные для UNIX, — система печати, удаленные архивы FTP, электронная почта и система телеконференций (news) — все это результат выполнения определенных процессов.

В этой главе рассматриваются вопросы: что такое процесс в представлении операционной системы, каковы связанные с ним структуры данных, позволяющие UNIX осуществлять управление процессом, а также описывается жизненный цикл процесса — от его создания до прекращения выполнения.

Процессы в UNIX неотъемлемо связаны с двумя важнейшими ресурсами системы — процессором (или процессорами) и оперативной памятью. Как правило, этих ресурсов никогда не бывает "много", и в операционной системе происходит активная конкурентная борьба за право обладания процессором и памятью. Мы рассмотрим принципы организации и управления памятью, т. к. даже при самом умеренном объеме физической памяти адресное пространство процесса составляет несколько гигабайт! Мы также подробно остановимся на том, как операционная система планирует выполнение процессов — ведь в каждый момент времени в однопроцессорной системе UNIX может выполняться не более одного процесса. UNIX является многозадачной системой общего назначения, поэтому задача справедливого распределения этого ресурса между задачами различного класса и с различными требованиями является нетривиальной.

Мы познакомимся с тем, как создаются новые процессы и запускаются новые программы (из предыдущих глав вы помните, что это не одно и то же). По существу процесс является "рамкой", в которую необходимо вставить "картину" или "фотографию" — некоторую прикладную программу. В этой главе рассматриваются важные этапы жизни процесса, такие как сон и пробуждение, переключение контекста, связанного со сменой задачи, и завершение его выполнения.

Последние разделы главы посвящены взаимодействию между процессами. Хотя основной задачей операционной системы является изоляция отдельного процесса от остальных, время от времени процессам все же требуется обмениваться данными. Для этого UNIX предлагает широкий спектр средств — от элементарного механизма сигналов до сложных подсистем межпроцессного взаимодействия — IPC UNIX System V и сокетов BSD.

## Основы управления процессом

Уже говорилось, что процесс UNIX представляет собой исполняемый образ программы, включающий отображение в памяти исполняемого файла, полученного в результате компиляции, стек, код и данные библиотек, а также ряд структур данных ядра, необходимых для управления процессом. На рис. 3.1 схематически представлены компоненты, необходимые для создания и выполнения процесса.

Процесс во время выполнения использует различные системные ресурсы — память, процессор, услуги файловой подсистемы и подсистемы ввода/вывода. Операционная система UNIX обеспечивает иллюзию одновременного выполнения нескольких процессов, эффективно распределяя системные ресурсы между активными процессами и не позволяя в то же время ни одному из них монополизировать использование этих ресурсов.

Новорожденная операционная система UNIX обеспечивала выполнение всего двух процессов, по одному на каждый подключенный к PDP-7 терминал. Спустя год, на той же PDP-7 число процессов заметно увеличилось, появился системный вызов *fork(2)*. В Первой редакции UNIX появился вызов *exec(2)*, но операционная система по-прежнему позволяла размещать в памяти только один процесс в каждый момент времени. После реализации аппаратной подсистемы управления памятью на PDP-11 операционная система была модифицирована, что позволило загружать в память сразу несколько процессов, уменьшая тем самым время на сохранение образа процесса во вторичной памяти (на диске) и считывание его, когда процесс продолжал выполнение. Однако до 1972 года UNIX нельзя было назвать действительно многозадачной системой, т. к. операции ввода/вывода оставались синхронными, и другие процессы не могли выполняться, пока их "коллега" не завершал операцию ввода/вывода (обычно

достаточно продолжительную). Истинная многозадачность появилась только после того, как код UNIX был переписан на языке С в 1973 году. С тех пор основы управления процессами практически не изменились.

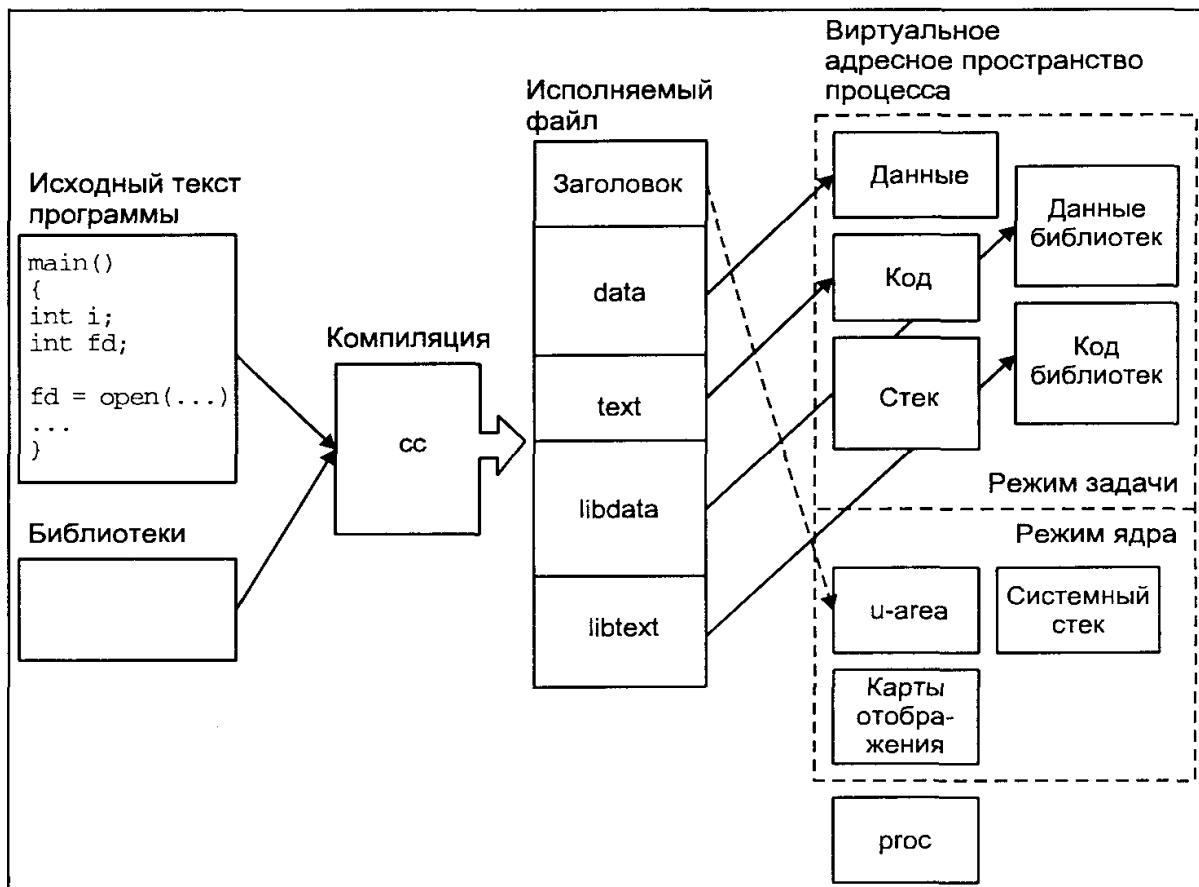


Рис. 3.1. Инфраструктура процесса операционной системы UNIX

Выполнение процесса может происходить в двух режимах — в *режиме ядра* (kernel mode) или в *режиме задачи* (user mode). В режиме задачи процесс выполняет инструкции прикладной программы, допустимые на непrivилегированном уровне защиты процессора. При этом процессу недоступны системные структуры данных. Когда процессу требуется получение каких-либо услуг ядра, он делает системный вызов, который выполняет инструкции ядра, находящиеся на привилегированном уровне. Несмотря на то что выполняются инструкции ядра, это происходит от имени процесса, сделавшего системный вызов. Выполнение процесса при этом переходит в режим ядра. Таким образом ядро системы защищает собственное адресное пространство от доступа прикладного процесса, который может нарушить целостность структур данных ядра и привести к разрушению операционной системы. Более того, часть процессорных инструкций, например, изменение регистров, связанных с управлением памятью, могут быть выполнены только в режиме ядра.

Соответственно и образ процесса состоит из двух частей: данных режима ядра и режима задачи. Образ процесса в режиме задачи состоит из сегмен-

та кода, данных, стека, библиотек и других структур данных, к которым он может получить непосредственный доступ. Образ процесса в режиме ядра состоит из структур данных, недоступных процессу в режиме задачи, которые используются ядром для управления процессом. Сюда относятся данные, диктуемые аппаратным уровнем, например состояния регистров, таблицы для отображения памяти и т. д., а также структуры данных, необходимые ядру для обслуживания процесса. Вообще говоря, в режиме ядра процесс имеет доступ к любой области памяти.

## Структуры данных процесса

Каждый процесс представлен в системе двумя основными структурами данных — `proc` и `user`, описанными, соответственно, в файлах `<sys/proc.h>` и `<sys/user.h>`. Содержимое и формат этих структур различны для разных версий UNIX. В табл. 3.1 приведены некоторые поля структуры `proc` в SCO UNIX, позволяющие проиллюстрировать информацию, необходимую ядру, для управления процессом.

**Таблица 3.1.** Структура `proc`

char	<code>p_stat</code>	Состояние процесса (выполнение, приостановлен, сон и т. д.)
char	<code>p_pri</code>	Текущий приоритет процесса
unsigned int	<code>p_flag</code>	Флаги, определяющие дополнительную информацию о состоянии процесса
unsigned short	<code>p_uid</code>	UID процесса
unsigned short	<code>p_suid</code>	EUID процесса
int	<code>p_sid</code>	Идентификатор сеанса
short	<code>p_pgrp</code>	Идентификатор группы процессов (равен идентификатору лидера группы)
short	<code>p_pid</code>	Идентификатор процесса (PID)
short	<code>p_ppid</code>	Идентификатор родительского процесса (PPID)
<code>sigset t</code>	<code>p_sig</code>	Сигналы, ожидающие доставки
unsigned int	<code>p_size</code>	Размер адресного пространства процесса в страницах
<code>time t</code>	<code>p_utime</code>	Время выполнения в режиме задачи
<code>time t</code>	<code>p_stime</code>	Время выполнения в режиме ядра
<code>caddr t</code>	<code>p_ldt</code>	Указатель на LDT процесса
<code>struct pregion</code>	<code>*p_region</code>	Список областей памяти процесса
short	<code>p_xstat</code>	Код возврата, передаваемый родительскому процессу
unsigned int	<code>p_utbl [ ]</code>	Массив записей таблицы страниц для u-area

В любой момент времени данные структур `proc` для всех процессов должны присутствовать в памяти, хотя остальные структуры данных, включая образ процесса, могут быть перемещены во вторичную память, — область swapинга. Это позволяет ядру иметь под рукой минимальную информацию, необходимую для определения местонахождения остальных данных, относящихся к процессу, даже если они отсутствуют в памяти.

Структура `proc` является записью системной таблицы процессов, которая, как мы только что заметили, всегда находится в оперативной памяти. Запись этой таблицы для выполняющегося в настоящий момент времени процесса адресуется системной переменной `curproc`. Каждый раз при переключении контекста, когда ресурсы процессора передаются другому процессу, соответственно изменяется значение переменной `curproc`, которая теперь указывает на структуру `proc` активного процесса.

Вторая упомянутая структура — `user`, также называемая `u-area` или `u block`, содержит дополнительные данные о процессе, которые требуются ядру только во время выполнения процесса (т. е. когда процессор выполняет инструкции процесса в режиме ядра или задачи). В отличие от структуры `proc`, адресованной указателем `curproc`, данные `user` размещаются (точнее, отображаются) в определенном месте виртуальной памяти ядра и адресуются переменной `u`. На рис. 3.2 показаны две основные структуры данных процесса и способы их адресации ядром UNIX.

В `u-area` хранятся данные, которые используются многими подсистемами ядра и не только для управления процессом. В частности, там содержится информация об открытых файловых дескрипторах, диспозиция сигналов, статистика выполнения процесса, а также сохраненные значения регистров, когда выполнение процесса приостановлено. Очевидно, что процесс не должен иметь возможности модифицировать эти данные произвольным образом, поэтому `u-area` защищена от доступа в режиме задачи.

Как видно из рис. 3.2, `u-area` также содержит стек фиксированного размера, — *системный стек* или *стек ядра* (*kernel stack*). При выполнении процесса в режиме ядра операционная система использует этот стек, а не обычный стек процесса.

## Состояния процесса

Жизненный цикл процесса может быть разбит на несколько состояний. Переход процесса из одного состояния в другое происходит в зависимости от наступления тех или иных событий в системе. На рис. 3.3 показаны состояния, в которых процесс может находиться с момента создания до завершения выполнения.

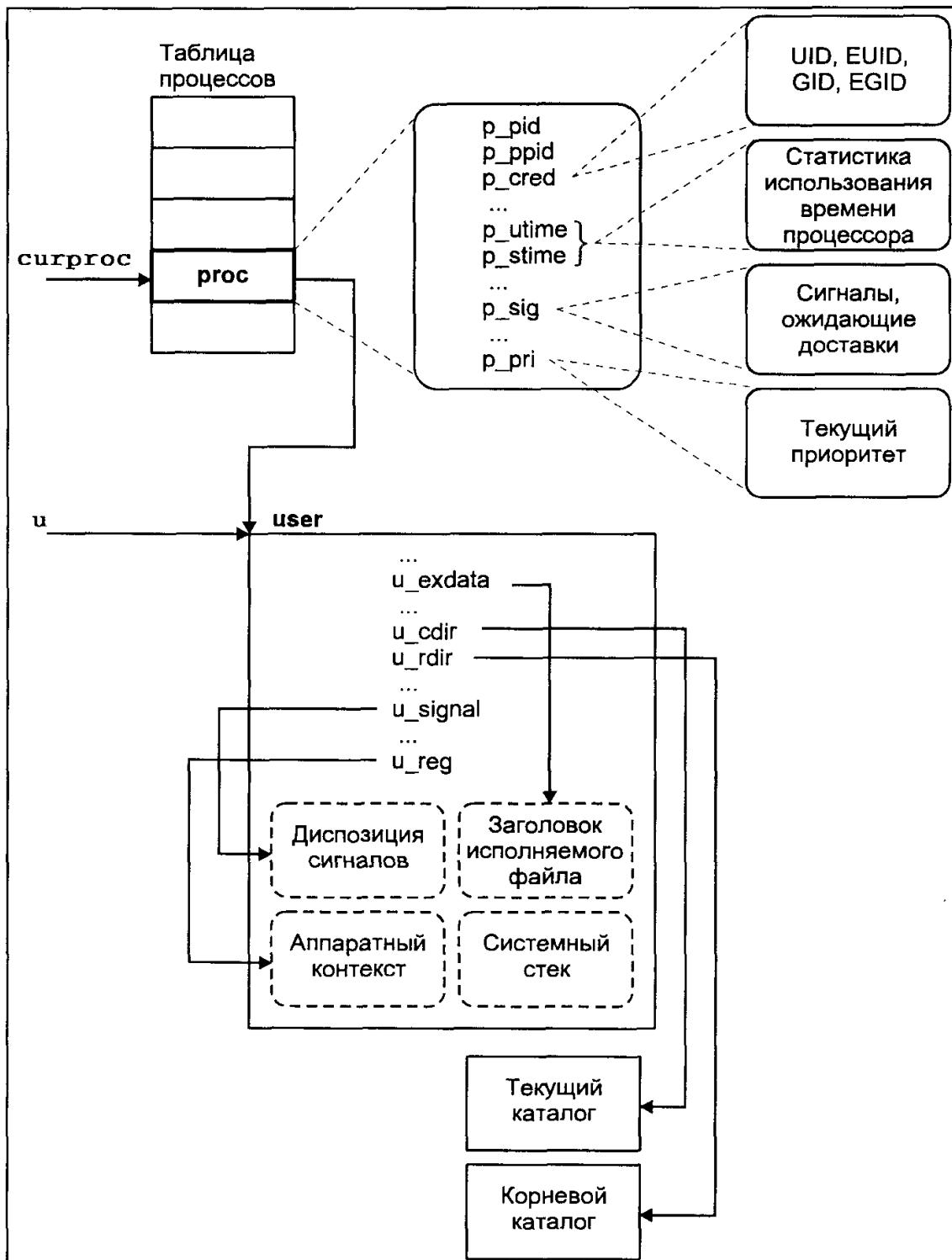


Рис. 3.2. Основные структуры данных процесса

1. Процесс выполняется в режиме задачи. При этом процессором выполняются прикладные инструкции данного процесса.
2. Процесс выполняется в режиме ядра. При этом процессором выполняются системные инструкции ядра операционной системы от имени процесса.

3. Процесс не выполняется, но готов к запуску, как только планировщик выберет его (состояние *runnable*). Процесс находится в очереди на выполнение и обладает всеми необходимыми ему ресурсами, кроме вычислительных.
4. Процесс находится в состоянии сна (*asleep*), ожидая недоступного в данный момент ресурса, например завершения операции ввода/вывода.
5. Процесс возвращается из режима ядра в режим задачи, но ядро прерывает его и производит переключение контекста для запуска более высокоприоритетного процесса.
6. Процесс только что создан вызовом *fork(2)* и находится в переходном состоянии: он существует, но не готов к запуску и не находится в состоянии сна.
7. Процесс выполнил системный вызов *exit(2)* и перешел в состояние зомби (*zombie, defunct*). Как такового процесса не существует, но остаются записи, содержащие код возврата и временную статистику его выполнения, доступную для родительского процесса. Это состояние является конечным в жизненном цикле процесса.

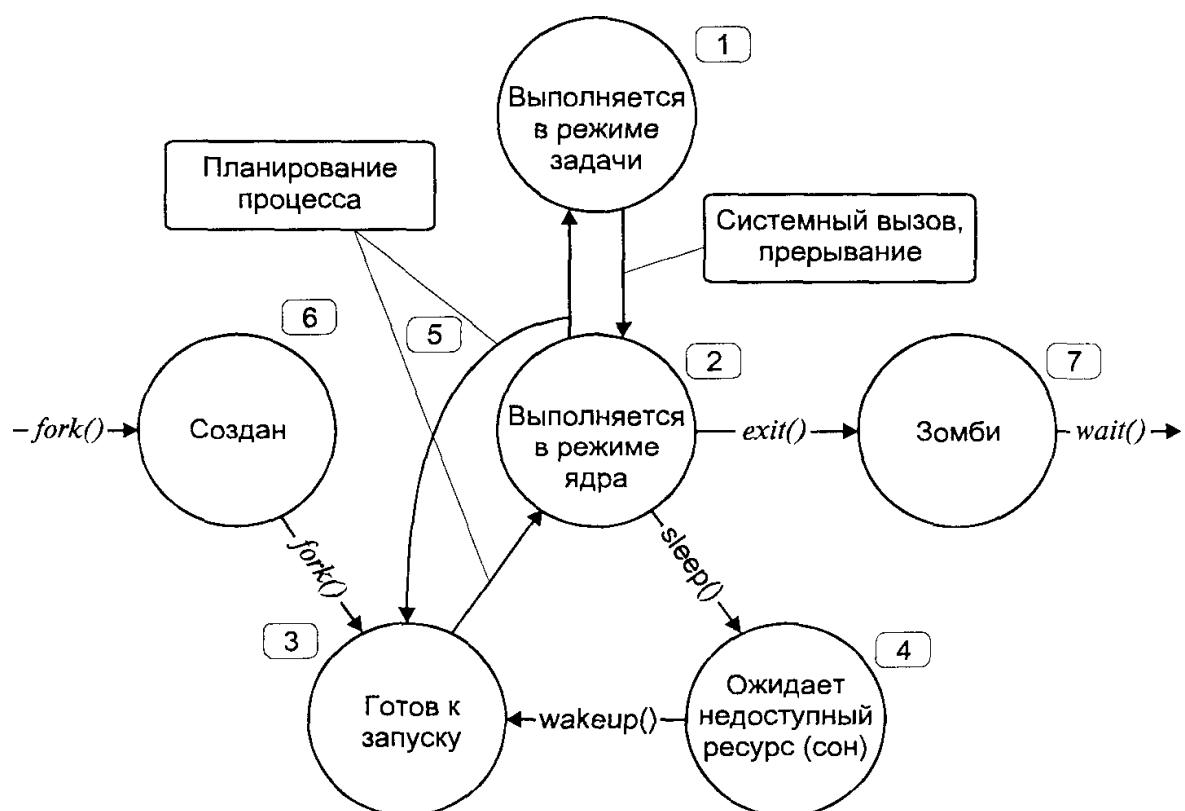


Рис. 3.3. Состояния процесса

Необходимо отметить, что не все процессы проходят через все множество состояний, приведенных выше.

Процесс начинает свой жизненный путь с состояния 6, когда родительский процесс выполняет системный вызов `fork(2)`. После того как создание процесса полностью завершено, процесс завершает "дочернюю часть" вызова `fork(2)` и переходит в состояние 3 готовности к запуску, ожидая своей очереди на выполнение. Когда планировщик выбирает процесс для выполнения, он переходит в состояние 1 и выполняется в режиме задачи.

Выполнение в режиме задачи завершается в результате системного вызова или прерывания, и процесс переходит в режим ядра, в котором выполняется код системного вызова или прерывания. После этого процесс опять может вернуться в режим задачи. Однако во время выполнения системного вызова в режиме ядра процессу может понадобиться недоступный в данный момент ресурс. Для ожидания доступа к такому ресурсу, процесс вызывает функцию ядра `sleep()` и переходит в состояние сна (4). При этом процесс добровольно освобождает вычислительные ресурсы, которые предоставляются следующему наиболее приоритетному процессу. Когда ресурс становится доступным, ядро "пробуждает процесс", используя функцию `wakeup()`, помещает его в очередь на выполнение, и процесс переходит в состояние "готов к запуску"(3).

При предоставлении процессу вычислительных ресурсов происходит *переключение контекста* (context switch), в результате которого сохраняется образ, или контекст, текущего процесса, и управление передается новому. Переключение контекста может произойти, например, если процесс перешел в состояние сна, или если в состоянии готовности к запуску находится процесс с более высоким приоритетом, чем текущий. В последнем случае ядро не может немедленно прервать текущий процесс и произвести переключение контекста. Дело в том, что переключению контекста при выполнении в режиме ядра может привести к нарушению целостности самой системы. Поэтому переключение контекста откладывается до момента перехода процесса из режима ядра в режим задачи, когда все системные операции завершены, и структуры данных ядра находятся в нормальном состоянии.

Таким образом, после того как планировщик выбрал процесс на запуск, последний начинает свое выполнение в режиме ядра, где завершает переключение контекста. Дальнейшее состояние процесса зависит от его предыстории: если процесс был только что создан или был прерван, возвращаясь в режим задачи, он немедленно переходит в этот режим. Если процесс начинает выполнение после состояния сна, он продолжает выполняться в режиме ядра, завершая системный вызов. Заметим, что такой процесс может быть прерван после завершения системного вызова в момент перехода из режима ядра в режим задачи, если в очереди существует более высокоприоритетный процесс.

В UNIX 4.xBSD определены дополнительные состояния процесса, в первую очередь связанные с системой управления заданиями и взаимодействий

вием процесса с терминалом. Процесс может быть переведен в состояние "остановлен" с помощью сигналов останова SIGSTOP, SIGTTIN или SIGTTOU. В отличие от других сигналов, которые обрабатываются только для выполняющегося процесса, отправление этих сигналов приводит к немедленному изменению состояния процесса<sup>1</sup>. В этом случае, если процесс выполняется или находится в очереди на запуск, его состояние изменяется на "остановлен". Если же процесс находился в состоянии сна, его состояние изменится на "остановлен в состоянии сна". Выход из этих состояний осуществляется сигналом продолжения SIGCONT, при этом из состояния "остановлен" процесс переходит в состояние "готов к запуску", а для процесса, остановленного в состоянии сна, следующим пунктом назначения является продолжение "сна". Описанные возможности полностью реализованы и в SVR4.

Наконец, процесс выполняет системный вызов *exit(2)* и заканчивает свое выполнение. Процесс может быть также завершен вследствие получения сигнала. В обоих случаях ядро освобождает ресурсы, принадлежавшие процессу, за исключением кода возврата и статистики его выполнения, и переводит процесс в состояние "зомби". В этом состоянии процесс находится до тех пор, пока родительский процесс не выполнит один из системных вызовов *wait(2)*, после чего вся информация о процессе будет уничтожена, а родитель получит код возврата завершившегося процесса.

## Принципы управления памятью

Одной из основных функций операционной системы является эффективное управление памятью. Оперативная память, или основная память, или память с произвольным доступом (Random Access Memory, RAM) является достаточно дорогостоящим ресурсом. Время доступа к оперативной памяти составляет всего несколько циклов процессора, поэтому работа с данными, находящимися в памяти, обеспечивает максимальную производительность. К сожалению, данный ресурс, как правило, ограничен. В большей степени это справедливо для многозадачной операционной системы общего назначения, каковой является UNIX. Поэтому данные, которые не могут быть размещены в оперативной памяти, располагаются на вторичных устройствах хранения, или во вторичной памяти, роль которой обычно выполняют дисковые накопители. Время доступа ко вторичной памяти на несколько порядков превышает время доступа к оперативной памяти и требует активного содействия операционной системы. Подсистема управления

Существует исключение из этого правила, касающееся процессов, находящихся в состоянии сна для низкоприоритетного события, т. е. события, вероятность наступления которого относительно мала (например, ввода с клавиатуры, который может и не наступить). В этом случае отправление процессу сигнала приведет к его пробуждению. Более подробно этот случай рассмотрен в разделе "Сигналы" этой главы.

памятью UNIX отвечает за справедливое и эффективное распределение разделяемого ресурса оперативной памяти между процессами и за обмен данными между оперативной и вторичной памятью. Часть операций производится аппаратно устройством управления памятью (Memory Management Unit, MMU) процессора под управлением операционной системы, чем достигается требуемое быстродействие.

Примитивное управление памятью значительно уменьшает функциональность операционной системы. Такие системы, как правило, позволяют загрузить в заранее определенное место в оперативной памяти единственную задачу и передать ей управление. При этом задача получает в свое распоряжение все ресурсы компьютера (разделяя их, разумеется, с операционной системой), а адреса, используемые задачей, являются физическими адресами оперативной памяти. Такой способ запуска и выполнения одной программы безусловно является наиболее быстрым и включает минимальные накладные расходы.

Этот подход часто используется в специализированных микропроцессорных системах, однако практически неприменим в операционных системах общего назначения, какой является UNIX. Можно сформулировать ряд возможностей, которые должна обеспечивать подсистема управления памятью современной многозадачной операционной системы:

- Выполнение задач, размер которых превышает размер оперативной памяти.
- П Выполнение частично загруженных в память задач для минимизации времени их запуска.
- Размещение нескольких задач в памяти одновременно для повышения эффективности использования процессора.
- П Размещение задачи в произвольном месте оперативной памяти.
- П Размещение задачи в нескольких различных частях оперативной памяти.
- П Совместное использование несколькими задачами одних и тех же областей памяти. Например, несколько процессов, выполняющих одну и ту же программу, могут совместно использовать сегмент кода.

Все эти возможности реализованы в современных версиях UNIX с помощью т. н. *виртуальной памяти*, о которой пойдет речь в следующем подразделе. Виртуальная память не является "бесплатным приложением", повышая накладные расходы операционной системы: структуры данных управления памятью размещаются в оперативной памяти, уменьшая ее размер; управление виртуальной памятью процесса может требовать ресурсоемких операций ввода/вывода; для системы со средней загрузкой около 7% процессорного времени приходится на подсистему управления памятью. Поэтому от эффективности реализации и работы этой подсистемы во многом зависит производительность операционной системы в целом.

## Виртуальная и физическая память

Оперативная память является, пожалуй, одним из наиболее дорогих компонентов компьютерной системы. Ранние системы UNIX имели в своем распоряжении 64 Кбайт оперативной памяти, и это количество было явно недостаточным, современные компьютеры обладают гигабайтами оперативной памяти, но и этого уже мало.

Оперативная память может быть представлена в виде последовательности байтов, каждый из которых имеет свой уникальный адрес, называемый *физическими адресами*. Именно эти адреса в конечном счете использует процессор, обмениваясь данными с оперативной памятью. Однако адресное пространство процесса существенным образом отличается от адресного пространства физической оперативной памяти. Представим себе, что адресное пространство процесса непосредственно отображалось бы в оперативную память, другими словами, что адреса, используемые процессом, являлись бы физическими адресами. При таком подходе на пути создания многозадачной системы нас ожидал бы ряд непреодолимых препятствий:

- Во-первых, трудно себе представить механизм, защищающий адресное пространство одного процесса, от адресного пространства другого или, что более важно, от адресного пространства самой операционной системы. Поскольку каждый процесс работает с физическими адресами, нет никакой гарантии, что процесс не обратится к ячейкам памяти, принадлежащим другим процессам или ядру системы. Последствия такого обращения скорее всего будут весьма плачевными.
- Во-вторых, уже на этапе компиляции необходимо было бы предусмотреть распределение существующего физического адресного пространства. При запуске каждый процесс должен занимать непрерывную и непересекающуюся область физических адресов.
- П В-третьих, подобное распределение памяти между процессами вряд ли можно назвать оптимальным. Объем физической оперативной памяти будет существенным образом ограничивать число процессов, одновременно выполняющихся в системе. Так восемь процессов, каждый из которых занимает 1 Мбайт памяти, исчерпают 8 Мбайт оперативной памяти, а операционная система при средней загрузке насчитывает более 80 процессов!

Все перечисленные проблемы преодолимы с помощью виртуальной памяти. При этом адреса, используемые приложениями и самим ядром, не обязаны соответствовать физическим адресам. Виртуальные адреса транслируются или отображаются в физические на аппаратном уровне при активном участии ядра операционной системы.

Смысл виртуальной памяти заключается в том, что каждый процесс выполняется в собственном *виртуальном адресном пространстве*. Виртуальное

адресное пространство — настоящий рай для процесса. Во-первых, у процесса создается ощущение исключительности — ведь все адресное пространство принадлежит только ему. Во-вторых, он больше не ограничен объемом физической памяти — виртуальная память может значительно превышать физическую. В результате процессы становятся изолированными друг от друга и не имеют возможности (даже при желании) "хозяйничать" в адресном пространстве соседа. Физическая память распределяется максимально эффективно — она не зависит от распределения виртуальной памяти отдельного процесса.

Очевидно, что для реализации виртуальной памяти необходим управляемый механизм отображения виртуального адреса в физический. В современных компьютерных системах процесс отображения выполняется на аппаратном уровне (с помощью MMU), обеспечивая высокую скорость трансляции. Операционная система осуществляет управление этим процессом.

Современные процессоры, как правило, поддерживают объединение адресного пространства в области переменного размера — *сегменты* и области фиксированного размера — *страницы*. При этом для каждого сегмента или страницы может быть задано собственное отображение виртуальных адресов в физические.

На рис. 3.4 показана взаимосвязь между виртуальным и физическим адресным пространством. Виртуальное адресное пространство процесса, как правило, является последовательным в рамках уже знакомых нам сегментов — кода, данных, стека и библиотек. Расположение соответствующих областей физической памяти может иметь фрагментированный характер, позволяя оптимально распределять память между процессами.

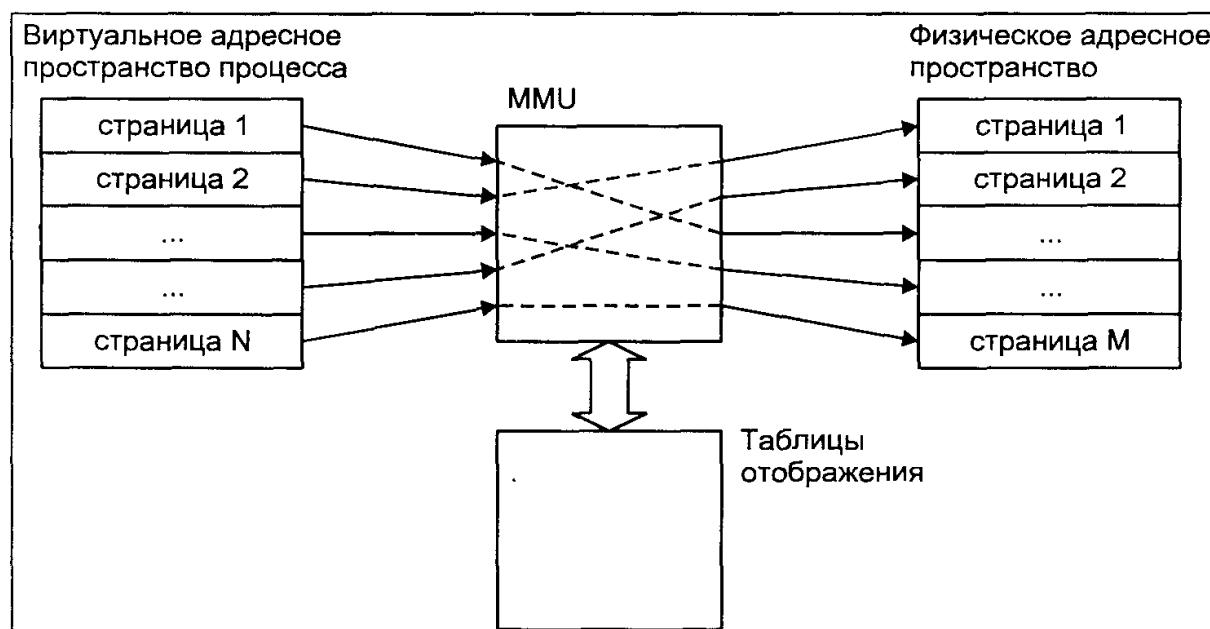


Рис. 3.4. Виртуальная и физическая память

Размер виртуальной памяти может существенно превышать размер физической за счет использования *вторичной памяти* или *области свопинга* — как правило, дискового пространства, где могут сохраняться временно не используемые участки адресного пространства процесса. Например, если при выполнении процесса происходит обращение к виртуальному адресу, для которого присутствует соответствующая страница физической памяти, операция чтения или записи завершится успешно. Если страница в оперативной памяти отсутствует, процессор генерирует аппаратное прерывание, называемое *страничной ошибкой* (page fault), в ответ на которое ядро определяет положение сохраненного содержимого страницы в области свопинга, считывает страницу в память, устанавливает параметры отображения виртуальных адресов в физические и сообщает процессору о необходимости повторить операцию. Все эти действия невидимы для приложения, которое работает с виртуальной памятью.

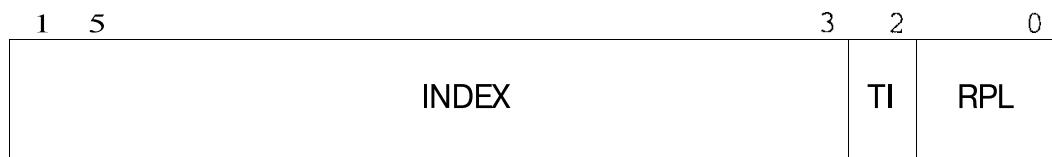
Механизм отображения виртуальных адресов в физические (трансляция адреса) существенным образом зависит от конкретной аппаратной реализации. Чтобы наше обсуждение не носило слишком абстрактного характера, в этом разделе рассмотрим механизм отображения виртуальных адресов в физические в операционной системе SCO UNIX на примере семейства процессоров Intel. Однако, как и для остальных подсистем UNIX, основные принципы отличаются мало, и данное изложение поможет читателю представить механизмы управления памятью и разобраться, при необходимости, в конкретной реализации.

## Сегменты

Семейство процессоров Intel позволяет разделить память на несколько логических частей, называемых *сегментами*. При этом адресное пространство процесса может быть представлено в виде нескольких логических сегментов, каждый из которых состоит из непрерывной последовательности адресов, лежащих в заданном диапазоне. Трансляция адресов, основанная на сегментации, предусматривает однозначное отображение адресов сегмента в непрерывную последовательность физических адресов. Виртуальный адрес при этом состоит из двух частей: *селектора сегмента* и *смещения* относительно начала сегмента. Селектор (точнее, поле селектора INDEX) указывает на так называемый *дескриптор сегмента*, содержащий такие параметры, как его расположение в памяти, размер и права доступа.

Процессор поддерживает косвенную адресацию сегментов через дескрипторы сегментов, которые располагаются в специальных таблицах — областях памяти, на которые указывают предназначенные для этого регистры процессора. Ядро операционной системы отвечает за заполнение этих таблиц и установку значений регистров. Другими словами, ядро задает отображение, а процессор выполняет отображение на аппаратном уровне. Благодаря такой косвенной адресации логические сегменты защищены друг от друга, что обеспечивает целостность адресного пространства процесса и ядра.

Дескрипторы сегментов расположены в двух системных таблицах — *локальной таблице дескрипторов* (Local Descriptor Table — LDT) и *глобальной таблице дескрипторов* (Global Descriptor Table — GDT). Как следует из названия, LDT обеспечивает трансляцию виртуальных адресов сегментов процесса, в то время как GDT обслуживает адресное пространство ядра (например, при обработке системного вызова или прерывания). Для каждого процесса создается собственная LDT, в то время как GDT разделяется всеми процессами. Информация о таблице, на которую указывает селектор, находится в самом селекторе, вид которого представлен на рис. 3.5.



**Рис. 3.5.** Селектор сегмента

Если бит TI равен 0, то селектор указывает на GDT, в противном случае используется LDT. Поле RPL задает уровень привилегий сегмента и является одним из механизмов обеспечения защиты сегментов. Например, если процесс, находясь в режиме задачи, попытается обратиться к сегменту, принадлежащему ядру, процессор сгенерирует особую ситуацию, в ответ на это ядро отправит процессу сигнал SIGSEGV.

Каждая запись LDT или GDT является дескриптором сегмента. Определено несколько типов дескрипторов, используемых для сегментов кода, данных и стека, а также ряд дескрипторов, с помощью которых обеспечивается многозадачность и передача управления от непrivилегированной задачи, например, процесса в режиме задачи, к привилегированной задаче, например, ядру. Дескрипторы, используемые в последнем случае, называются *шлюзами*.

Дескрипторы сегментов (кода, данных, стека) имеют несколько полей:

Базовый адрес	В этом поле хранится 32-битный адрес начала сегмента. Процессор добавляет к нему смещение и получает 32-битный линейный адрес.
Предел	Это поле определяет размер сегмента. Если результирующий линейный адрес выходит за пределы сегмента, процессор генерирует особую ситуацию. Границы сегмента позволяют процессору обнаруживать такие распространенные ошибки, как переполнение стека, неверные указатели, неверные адреса вызовов и переходов. В случае, когда операционная система считает, что обращение за пределы сегмента не является ошибкой (например, при переполнении стека), она может расширить сегмент путем выделения дополнительной памяти и запросить выполнение команды вновь.

Привилегии	Это поле, имеющее название Descriptor Privilege Level (DPL), определяет уровень привилегий сегмента и используется совместно с полем RPL селектора для разрешения или запрещения доступа к сегменту. Для получения доступа к сегменту задача должна иметь по крайней мере такой же уровень привилегий, как и сегмент, т. е. $RPL > DPL$ .
Признак присутствия	Этот бит обеспечивает один из механизмов реализации виртуальной памяти. Если бит не установлен, при попытке обращения к сегменту процессор генерирует особую ситуацию отсутствия сегмента, позволяя ядру подгрузить сегмент из вторичной памяти и вновь повторить инструкцию, не затрагивая при этом выполнение процесса. Однако в большинстве современных версий UNIX виртуальная память основана на страничном механизме, при котором сегмент всегда присутствует в памяти, а обмен между оперативной и вторичной памятью происходит на уровне страниц.
Тип	Это поле определяет тип сегмента. Процессор проверяет тип сегмента на соответствие исполняемой команде. Это, в частности, не позволяет интерпретировать информацию сегмента данных как инструкции процессора.
Права доступа	Это поле определяет права доступа, ограничивающие множество операций, которые можно производить с сегментом. Например, сегмент кода обычно отмечается как исполняемый и читаемый. Сегменты данных могут иметь право доступа только для чтения, или для чтения и записи.

Комбинация селектора и смещения образует логический адрес. Блок управления памятью процессора использует селектор для определения соответствующего ему дескриптора. Складывая базовый адрес сегмента, хранящийся в дескрипторе, со смещением, процессор создает линейный адрес (рис. 3.6).

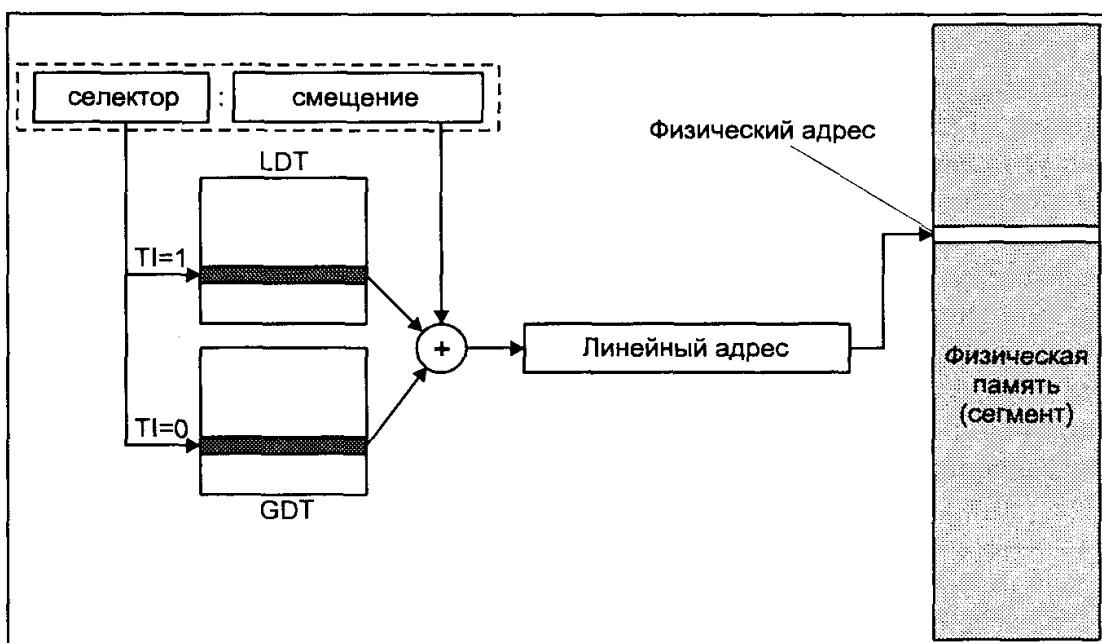


Рис. 3.6. Трансляция адреса с использованием механизма сегментации

Если страничный механизм не используется, полученный линейный адрес является физическим, используемым для непосредственного доступа к оперативной памяти. Однако реализация виртуальной памяти, основанная только на сегментах, не обладает достаточной гибкостью и не используется в современных версиях **UNIX**. Управление памятью в большинстве систем основано на страничном механизме. Сегменты используются ядром для размещения кода, данных и стека процесса, причем каждый из них имеет нулевой базовый адрес и предел — 3 Гбайт, т. е. всю адресуемую виртуальную память за вычетом 1 Гбайт, занимаемых ядром системы. Распределение виртуального адресного пространства между ядром и процессами рассмотрено в разделе "Адресное пространство процесса".

### **Страницный механизм**

При реализации виртуальной памяти, основанной только на сегментации, весь сегмент целиком может либо присутствовать в оперативной памяти, либо отсутствовать (точнее, находится во вторичной памяти или в исполняемом файле процесса). Поскольку размер сегмента может быть достаточно велик, одновременное выполнение нескольких больших процессов вызовет серьезную конкуренцию за ресурсы памяти, что в свою очередь приведет к интенсивному обмену данными между оперативной и вторичной памятью. К тому же обмен областями переменного размера, каковыми являются сегменты, достаточно сложен и, хотя фрагментация памяти при этом будет невелика, приведет к низкой эффективности ее использования, оставляя большое количество неиспользуемого пространства.

Страницный механизм обеспечивает гораздо большую гибкость. В этом случае все виртуальное адресное пространство (4 Гбайт для процессоров Intel) разделено на блоки одинакового размера, называемые *страницами*. Большинство процессоров Intel работает со страницами размером 4 Кбайт. Так же как и в случае сегментации, страница может либо присутствовать в оперативной памяти, либо находиться в области свопинга или исполняемом файле процесса. Основное преимущество такой схемы заключается в том, что система управления памятью оперирует областями достаточно малого размера для обеспечения эффективного распределения ресурсов памяти между процессами. Страницный механизм допускает, чтобы часть сегмента находилась в оперативной памяти, а часть отсутствовала. Это дает ядру возможность разместить в памяти только те страницы, которые в данное время используются процессом, тем самым значительно освобождая оперативную память. Еще одним преимуществом является то, что страницы сегмента могут располагаться в физической памяти в произвольном месте и порядке, что позволяет эффективно использовать свободное пространство<sup>2</sup>.

<sup>2</sup> Данный подход напоминает схему хранения файлов на диске — каждый файл состоит из различного числа блоков хранения данных, которые могут располагаться в любых свободных участках дискового накопителя. Это ведет к значительной фрагментации, но существенно повышает эффективность использования дискового пространства.

При использовании страничного механизма линейный адрес, полученный в результате сложения базового адреса сегмента и смещения также является логическим адресом, который дополнительно обрабатывается блоком страничной трансляции процессора. В этом случае линейный адрес рассматривается процессором как состоящий из трех частей, показанных на рис. 3.7.

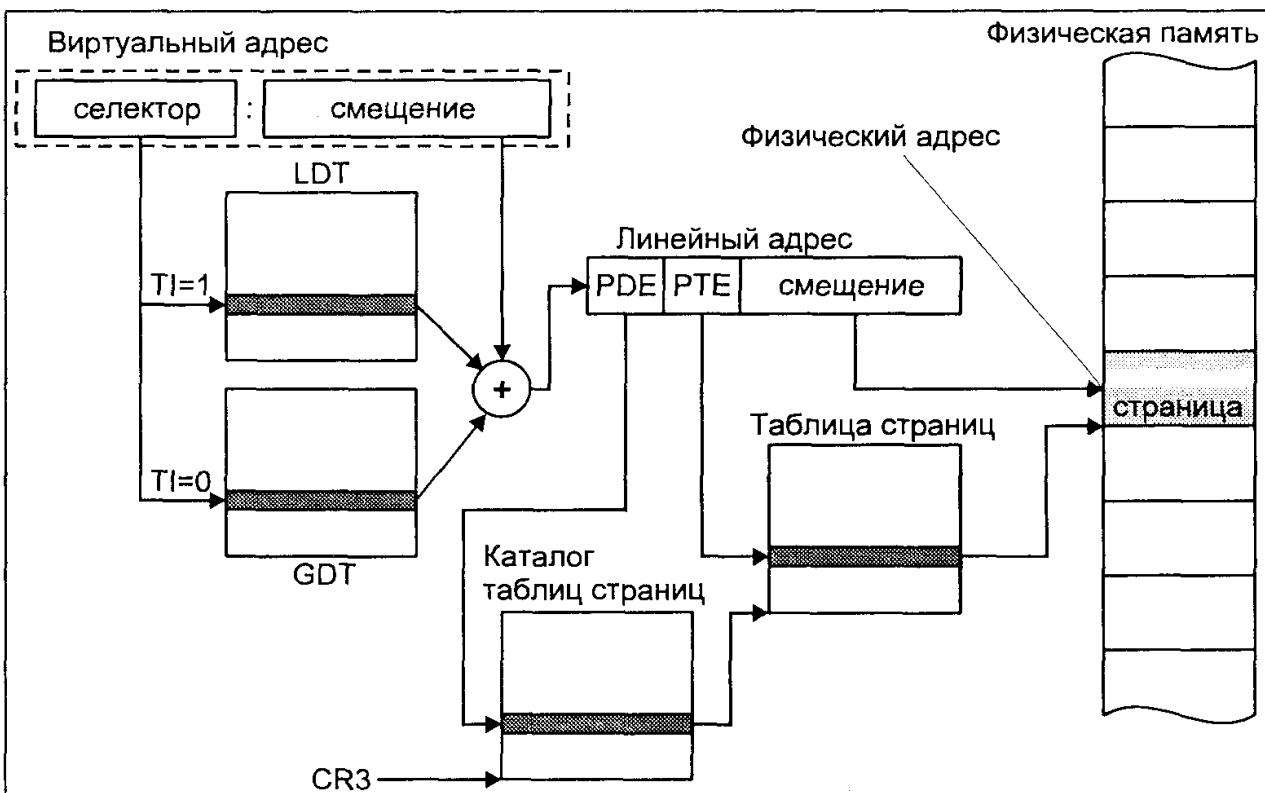


Рис. 3.7. Трансляция адреса с использованием страничного механизма

Первое поле адреса, с 22 по 31 бит, указывает на элемент *кatalogа таблиц страниц* (Page Directory Entry, PDE). Каталог таблиц страниц имеет длину, равную одной странице, и содержит до 1024 указателей на *таблицы страниц* (page table). Таким образом, первое поле адресует определенную таблицу страниц. Второе поле, занимающее с 12 по 21 бит, указывает на элемент *таблицы страниц* (Page Table Entry, PTE). Таблицы страниц также имеют длину 4 Кбайт, а элементы таблицы адресуют в совокупности 1024 страниц. Другими словами, второе поле адресует определенную страницу. Наконец, смещение на странице определяется третьим полем, занимающим младшие 12 бит линейного адреса. Таким образом, с помощью одного каталога таблиц процессор может адресовать  $1024 \times 1024 \times 4096 = 4$  Гбайт физической памяти.

На рис. 3.7 показано, как блок страничной адресации процессора транслирует линейный адрес в физический. Процессор использует поле PDE адреса (старшие 10 бит) в качестве индекса в каталоге таблиц. Найденный элемент содержит адрес таблицы страниц. Второе поле линейного адреса,

РТЕ, позволяет процессору выбрать нужный элемент таблицы, адресующий физическую страницу. Складывая адрес начала страницы со смещением, хранящимся в третьем поле, процессор получает 32-битный физический адрес<sup>3</sup>.

Каждый элемент таблицы страниц содержит несколько полей (табл. 3.2), описывающих различные характеристики страницы.

**Таблица 3.2.** Поля РТЕ

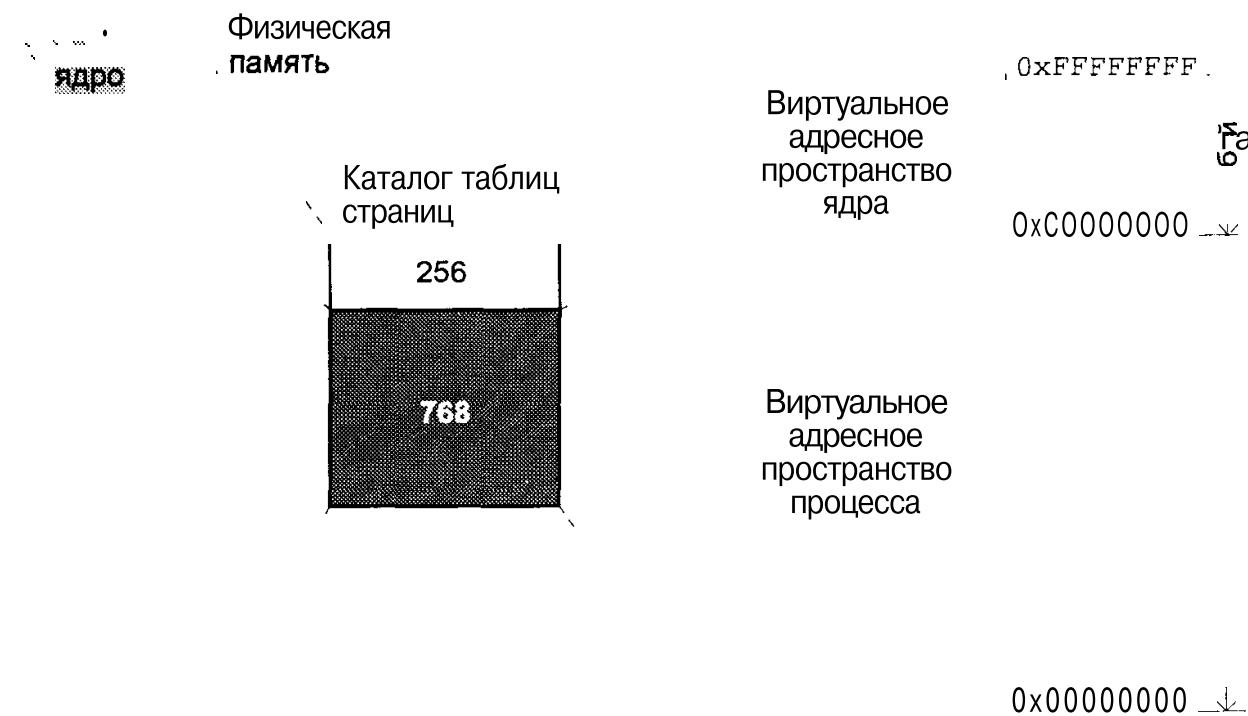
P	Признак присутствия в оперативной памяти. Доступ к странице, отсутствующей в памяти ( $P=0$ ) вызывает страниценную ошибку, особую ситуацию, о чём процессор информирует ядро, которое обрабатывает ее соответствующим образом.
R/W	Права только на чтение страницы ( $R/W=0$ ) или на чтение и запись ( $R/W=1$ ).
U/S	Привилегии доступа. Если $U/S = 0$ , только привилегированные задачи (ядро) имеют доступ к адресам страницы. В противном случае, доступ к странице имеют все задачи.
Адрес	Физический адрес начала страницы (адрес базы).

### Адресное пространство процесса

Адресное пространство ядра обычно совпадает с адресным пространством выполняющегося в данный момент процесса. В этом случае говорят, что ядро расположено в том же *контексте*, что и процесс. Каждый раз, когда процессу передаются вычислительные ресурсы, система восстанавливает контекст задачи этого процесса, включающий значения регистров общего назначения, сегментных регистров, а также указатели на таблицы страниц, отображающие виртуальную память процесса в режиме задачи. При этом системный контекст остается неизменным для всех процессов. Вид адресного пространства процесса представлен на рис. 3.8.

Специальный регистр (CR3 для Intel) указывает на расположение каталога таблиц страниц в памяти. В SCO UNIX используется только один каталог, независимо от выполняющегося процесса, таким образом значение регистра CR3 не меняется на протяжении жизни системы. Поскольку ядро (код и данные) является частью выполняющегося процесса, таблицы страниц, отображающие старший 1 Гбайт виртуальной памяти, принадлежащей ядру системы, не изменяются при переключении между процессами. Для отображения ядра используются старшие 256 элементов каталога.

Следует отметить, что большинство современных процессоров и, в частности, процессоры семейства Intel, помещают данные о нескольких последних использовавшихся ими страницах в сверхоперативный кэш. Только когда процессор не находит требуемой страницы в этом кэше, он обращается к каталогу и таблицам страниц. Как правило, 98–99% адресных ссылок попадают в кэш, не требуя для трансляции адреса обращения к оперативной памяти, где расположены каталог и таблицы.



**Рис. 3.8.** Адресное пространство в режимах ядра и задачи

При переключении между процессами, однако, изменяется адресное пространство режима задачи, что вызывает необходимость изменения оставшихся 768 элементов каталога. В совокупности они отображают 3 Гбайт виртуального адресного пространства процесса в режиме задачи. Таким образом, при смене процесса адресное пространство нового процесса становится видимым (отображаемым), в то время как адресное пространство предыдущего процесса является недоступным<sup>4</sup>.

Формат виртуальной памяти процесса в режиме задачи зависит, в первую очередь, от типа исполняемого файла, образом которого является процесс. На рис. 3.9 изображено расположение различных сегментов процесса в виртуальной памяти для двух уже рассмотренных нами форматов исполняемых файлов — COFF и ELF. Заметим, что независимо от формата исполняемого файла виртуальные адреса процесса не могут выходить за пределы 3 Гбайт.

Для защиты виртуальной памяти процесса от модификации другими процессами прикладные задачи не могут менять заданное отображение. По-

<sup>4</sup> При этом физические страницы, принадлежащие предыдущему процессу, могут по-прежнему оставаться в памяти, однако доступ к ним невозможен ввиду отсутствия установленного отображения. Любой допустимый виртуальный адрес будет отображаться либо в страницы ядра, либо в страницы нового процесса.

скольку ядро системы выполняется на привилегированном уровне, оно может управлять отображением как собственного адресного пространства, так и адресного пространства процесса.

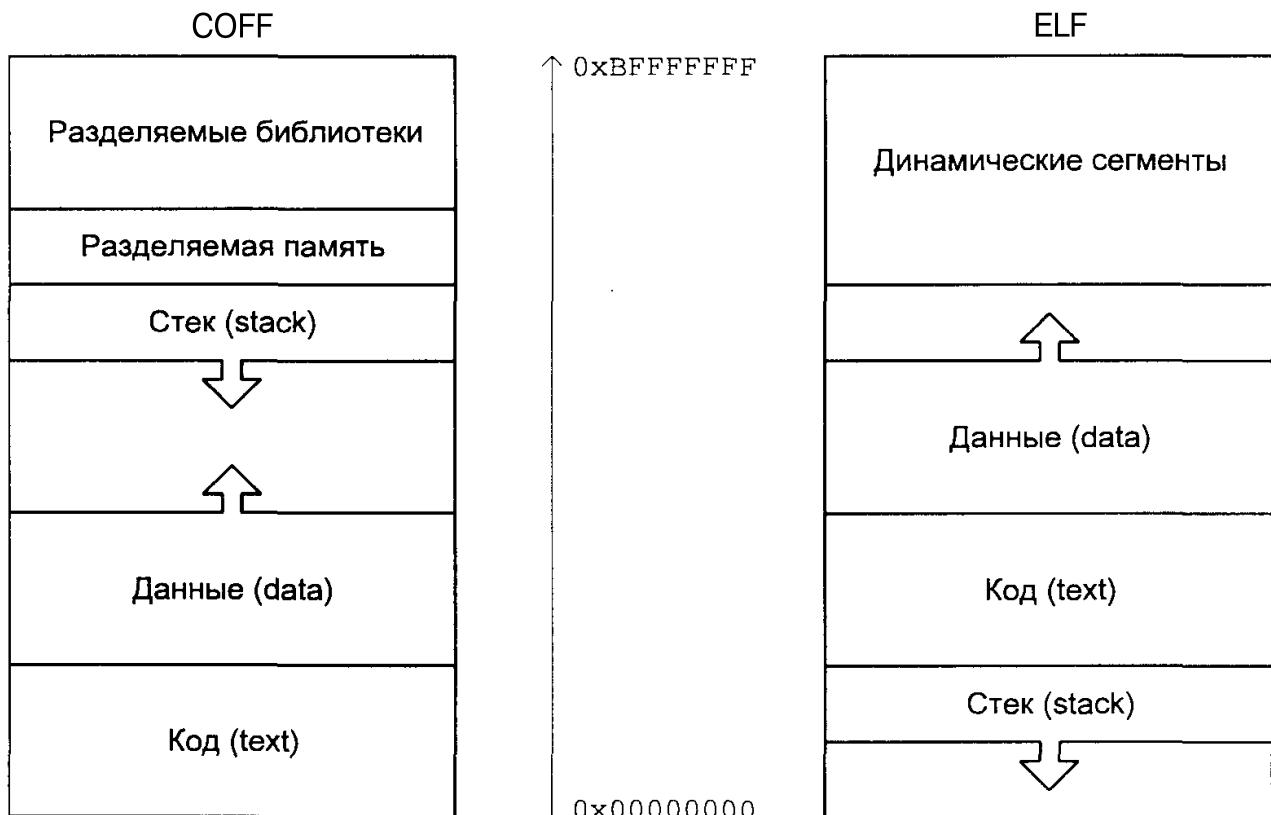


Рис. 3.9. Виртуальная память процесса в режиме задачи

## Управление памятью процесса

Можно сказать, что каждый процесс в операционной системе UNIX выполняется на собственной виртуальной вычислительной машине, где все ресурсы принадлежат исключительно данному процессу. Подсистема управления памятью обеспечивает такую иллюзию в отношении физической памяти.

Как уже говорилось, аппаратная поддержка страничного механизма имеет существенное значение для реализации виртуальной памяти. Однако при этом также требуется участие операционной системы. Можно перечислить ряд операций, за выполнение которых отвечает сама операционная система:

- Размещение в памяти каталога страниц и таблиц страниц; инициализация регистра — указателя на каталог таблиц страниц (для Intel — CR3) (в системах, использующих несколько каталогов страниц, каждый процесс хранит в и-агеа значение этого регистра; в этом случае инициализацию указателя необходимо проводить при каждом переключении контекста); инициализация каталога страниц.

- Установка отображения путем записи соответствующих значений в таблицы страниц.
- Обработка страничных ошибок.
- Управление сверхоперативным кэшем.
- Обеспечение обмена страницами между оперативной и вторичной памятью.

В реализации перечисленных функций существенную роль играют структуры данных, обеспечивающие удобное представление адресного пространства процесса для операционной системы. Фактический формат этих структур существенным образом зависит от аппаратной архитектуры и версии UNIX, поэтому в следующих разделах для иллюстрации тех или иных положений также использована операционная система SCO UNIX.

## Области

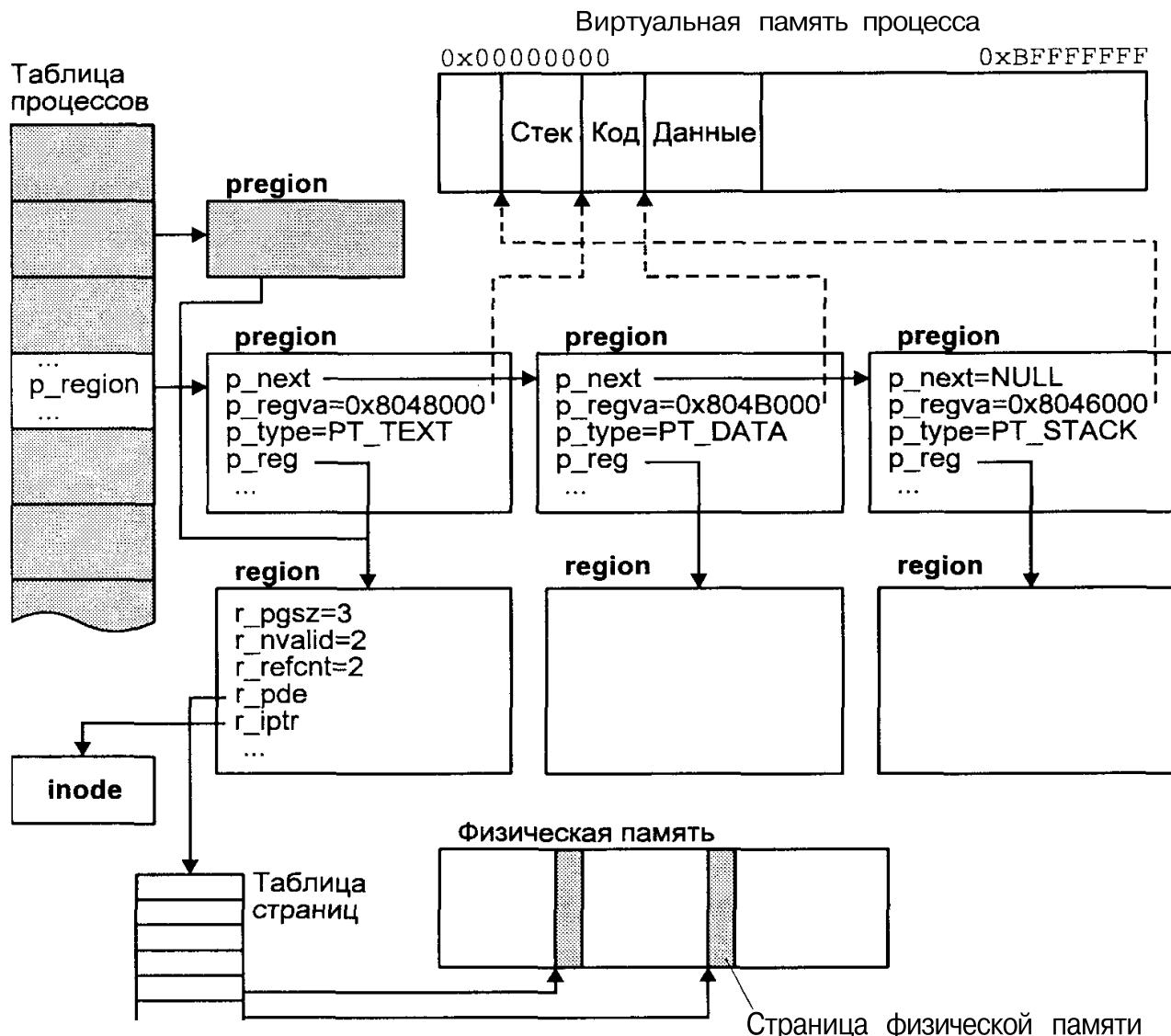
В SCO UNIX адресное пространство процесса разделено на несколько участков, называемых областями (region). *Область* представляет собой непрерывный участок виртуального адресного пространства процесса, который рассматривается ядром системы как отдельный объект, разделяемый или защищенный от постороннего доступа. Область может использоваться для хранения данных различных типов, включая код, данные, разделяемую память, сегменты библиотек и отображаемые в память файлы. Каждая активная область представлена соответствующей структурой данных ядра и служит основой для управления памятью процесса.

Каждая область представлена собственным сегментом памяти. В совокупности со страничным механизмом организации виртуальной памяти такой подход предоставляет ядру системы большие возможности по эффективному управлению виртуальной памятью процесса.

Области могут совместно использоваться несколькими процессами, при этом ядру нет необходимости создавать дополнительные копии, нужно лишь задать требуемое отображение (виртуальные адреса области у различных процессов могут не совпадать). В качестве примеров разделяемых областей можно привести разделяемую память, разделяемые библиотеки или отображаемые в память файлы. Часто код программы совместно используется несколькими родственными процессами. Информация о каждой активной области хранится ядром в структуре данных *region*.

Поскольку одна и та же область может использоваться несколькими процессами, для каждого процесса ядро создает связанный список структур *pregion* (per process region), которые в свою очередь адресуют области, используемые процессом. Указатель на список структур *pregion* для каждого процесса находится в записи таблицы процессов — структуре *proc*.

Основные поля структур *region* и *pregion* приведены на рис. 3.10.



**Рис. 3.10.** Управление адресным пространством процесса в SCO UNIX

Помимо указателей `p_next`, организующих структуры `pregion` в виде связанного списка, и `p_reg`, обеспечивающих адресацию соответствующей структуры `region`, в каждой структуре `pregion` определен набор флагов `p_flags`, определяющий права доступа к области, режим блокирования в памяти и т. д. Поле `p_type` указывает на тип области. Оно может содержать одно из следующих значений:

Значение	Описание
PT UNUSED	Область не используется
PT TEXT	Область содержит сегмент кода
PT DATA	Область содержит сегмент данных
PT STACK	Область используется в качестве стека процесса
PT SHMEM	Область используется в качестве разделяемой памяти

(продолжение)

Значение	Описание
PT_LIBTXT	Область содержит код библиотек
PT_LIBDAT	Область содержит данные библиотек
PT_SHFIL	Область используется для хранения файла, отображенного в память

Наконец, поле `p_regva` задает виртуальный адрес области в адресном пространстве процесса.

Поля структуры `region`, приведенные на рис. 3ЛО, имеют следующие значения. Поле `r_pgsize` определяет размер области в страницах, из которых `r_nvalid` страниц присутствуют в оперативной памяти (см. далее раздел "Страницное замещение"). Несколько процессов могут ссылаться на одну и ту же область, поле `r_refcnt` хранит число таких ссылок. Поле `r_pde` адресует таблицу страниц области<sup>5</sup>. Поле `r_iptr` адресует `inode` файла, где располагаются данные области (например, для области кода, `r_iptr` будет указывать на `inode` исполняемого файла).

Фактическую информацию о структурах управления адресным пространством процесса можно получить с помощью команды `crash(1M)`. В следующем примере таким образом определяется содержимое структур `pregion` процесса и характеристики соответствующих областей.

```
t crash
dumpfile = /dev/mem, namelist = /unix, outfile = stdout
> pregion 101
  SLOT  PREG#  REG#        REGVA      TYPE      FLAGS
    101      0    12      0x700000  text      rdonly
              1    22      0x701000  data
              2    23      0x7fffffc  stack
              3   145      0x80001000  lbtxt    rdonly
              4   187      0x80031000  lbdat    pr
```

Как можно увидеть из вывода команды `crash(1M)`, с рассматриваемым процессом связаны пять областей: сегмент кода, данных и стека, а также сегменты кода и данных подключенной библиотеки. Столбец `REG#` определяет запись таблицы областей, где расположена адресуемая каждой `pregion` область `region`. Заметим, что значение в столбце `REG#` лишь отчасти соответствует полю `p_reg` структуры `pregion`, поскольку последнее является указателем, а не индексом таблицы. Столбец `REGVA` содержит значения виртуальных адресов областей.

<sup>5</sup> Для областей, размер которых превышает 4 Мбайт, одной таблицы страниц недостаточно, и `region` хранит элементы каталога таблиц страниц в виде связанного списка.

С помощью полученной информации мы можем более детально рассмотреть любую из областей процесса. Выведем данные о сегментах кода, данных и стека:

```
>region 12 22 23
SLOT  PGSZ  VALID  SMEM  NONE  SOFF  KEF  SWP  NSW  FORW  BACK  INOX  TYPE  FLAGS
  12    1      1      1      0      0     11     0      0     15      5    154  stxt  done
  22    3      1      0      0      0     0     1     0     238     23    154  priv  done
  23    2      1      1      0      0     0     1     0     135     24          priv  stack
```

Столбец PGSZ определяет размер области в страницах, а столбец VALID — число страниц этой области, находящихся в оперативной памяти. Как можно заметить, для сегментов данных и стека страниц недостаточно, поэтому может возникнуть ситуация, когда процессу потребуется обращение к адресу, в настоящее время отсутствующему в памяти. Заметим также, что столбец INOX содержит индексы таблиц inode, указывающие на метаданные файлов, откуда было загружено содержимое соответствующих сегментов.

Мы можем взглянуть на дополнительные сведения об этом файле:

```
>inode 154
INODE TABLE SIZE = 472
SLOT  MAJ/MIN  FS  INUMB  RCNT  LINK  UID  GID  SIZE      MODE  MNT  M/ST  FLAGS
  154    1,42    2  1562     3      1   123     56  8972  f---755      0  R130  tx
```

Из этой таблицы мы можем определить файловую систему, в которой расположен файл (MAJ/MIN), а также номер его дискового inode — INUMB. В данном случае он равен 1562. Выполнив команду *ncheck(1)*, мы узнаем имя исполняемого файла, соответствующего исследуемому процессу:

```
$ ncheck -i 1562
/de/root:
1562      /home/andrei/CH3/test
```

## Замещение страниц

Ранние версии UNIX работали на компьютерах PDP-11 с 16-разрядной архитектурой и адресным пространством 64 Кбайт. Некоторые модификации позволяли использовать отдельные адресные пространства для кода и данных, накладывая тем не менее существенные ограничения на размер адресного пространства процесса. Это привело к разработке различных схем *программных оверлеев* (overlay), использовавшихся как для прикладных задач, так и для ядра операционной системы. Суть этих методов заключается в том, что в неиспользуемые участки адресного пространства процесса записываются другие части программы. Например, после запуска системы необходимость в функциях начальной инициализации отпадает и часть памяти, содержащая этот код, может быть использована для хранения других данных или инструкций операционной системы. Не говоря о значительной сложности такого подхода для разработчиков программного обеспечения, использование этих методов приводило к низкой переноси-

ности программ, поскольку они в значительной степени зависели от конкретной организации памяти. Порой даже расширение оперативной памяти требовало внесения модификаций в программное обеспечение.

Механизмы управления памятью сводились к использованию свопинга. Процессы загружались в непрерывные области оперативной памяти целиком, выгружался процесс также целиком. Только небольшое число процессов могло быть одновременно размещено в памяти, и при запуске процесса на выполнение, несколько других процессов необходимо было переместить во вторичную память. Схема управления памятью, основанная на механизме свопинга, показана на рис. 3.11.

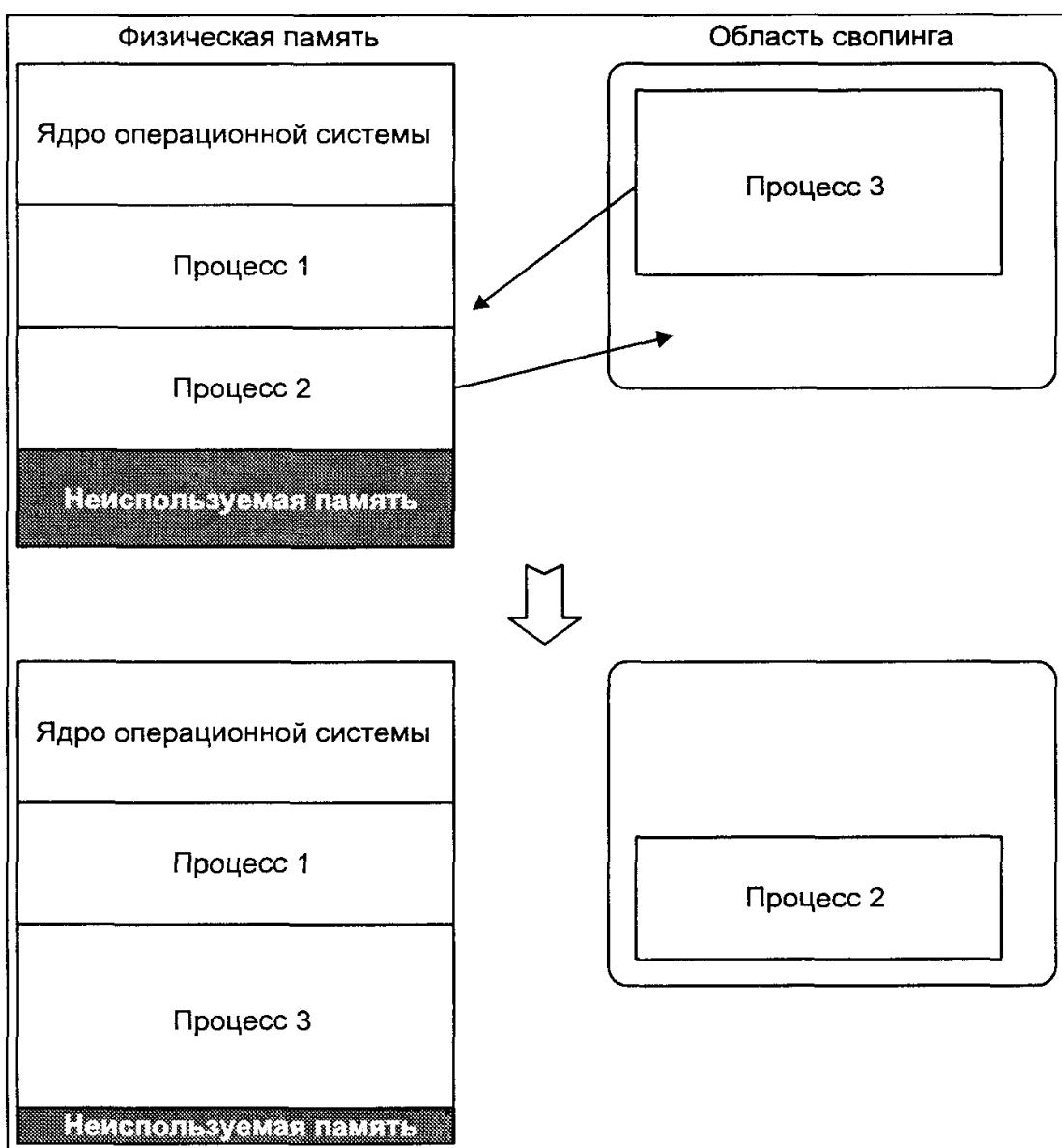


Рис. 3.11. Управление памятью, основанное на свопинге

Механизм страничного замещения по требованию был реализован в UNIX в 1978 году на новом компьютере VAX-11/780, имевшем 32-разрядную ар-

хитектуру, 4 Гбайт адресуемого пространства и аппаратную поддержку страничного механизма. Первой системой UNIX, в которой управление памятью основывалось на страничном замещении по требованию, явилась версия 3.xBSD. Уже в середине 80-х годов все основные версии UNIX обеспечивали страничное замещение в качестве основного механизма, оставляя swapингу вторую роль.

Как уже говорилось в системах с виртуальной памятью, основанной на страничном механизме, адресное пространство процесса разделено на последовательные участки равной длины, называемыми страницами. Такая же организация присуща и физической памяти, и в конечном итоге любое место физической памяти адресуется номером страницы и смещением в ней. Деление адресного пространства процесса является логическим, причем логическим последовательным страницам виртуальной памяти при поддержке операционной системы и аппаратуры (MMU процессора) ставятся в соответствие определенные физические страницы оперативной памяти. Эта операция получила название *трансляции адреса*.

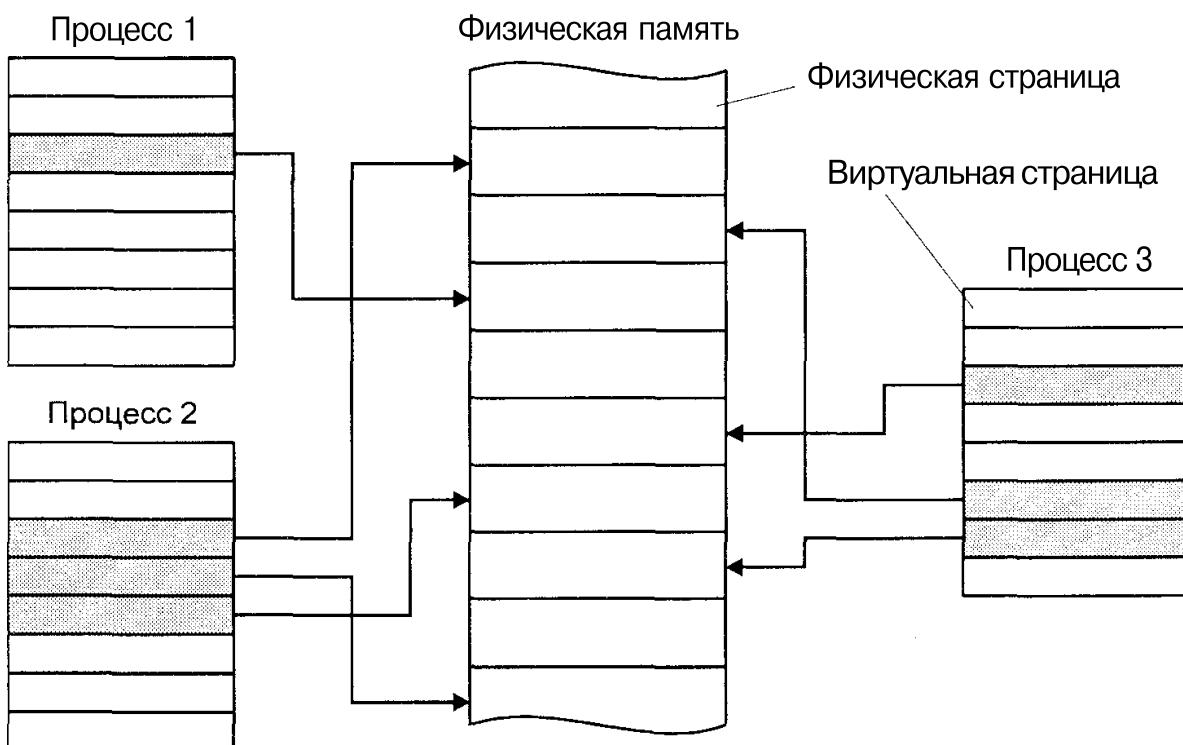
Однако механизм трансляции адреса является первым условием реализации виртуальной памяти, позволяя отделить виртуальное адресное пространство процесса от физического адресного пространства процессора. Вторым условием является возможность выполнения процесса, чье адресное пространство не имеет полного отображения на физическую память. Чтобы удовлетворить второму условию, каждая страница виртуальной памяти имеет флаг присутствия в оперативной памяти. Если адресуемая страница отсутствует в памяти, аппаратура генерирует страничную ошибку, которая обрабатывается операционной системой, в конечном итоге приводя к размещению этой страницы в памяти. Таким образом, для выполнения процесса является необходимым присутствие в памяти лишь нескольких страниц процесса, к которым в данный момент происходит обращение (рис. 3.12).

Вообще говоря, конкретный механизм страничного замещения зависит от того, как реализованы три основных принципа:

1. При каких условиях система загружает страницы в память, т. н. *принцип загрузки* (fetch policy).
2. В каких участках памяти система размещает страницы, т. н. *принцип размещения* (placement policy).
3. Каким образом система выбирает страницы, которые требуется освободить из памяти, когда отсутствуют свободные страницы для размещения (или их число меньше некоторого порогового значения), т. н. *принцип замещения* (replacement policy).

Обычно все физические страницы одинаково подходят для размещения, и принцип размещения не оказывает существенного влияния на работу механизма в целом. Таким образом эффективность управления памятью пол-

ностью зависит от двух остальных принципов: загрузки и замещения. В системах с чистым страничным замещением по требованию в память помещаются только требуемые страницы, а замещение производится, когда полностью отсутствует свободная оперативная память. Соответственно, производительность таких систем полностью зависит от реализации принципа замещения. Однако большинство современных версий UNIX не используют чистого страничного замещения по требованию. Вместо этого принцип загрузки предполагает размещение сразу нескольких страниц, обращение к которым наиболее вероятно в ближайшее время, а замещение производится до того, как память будет полностью занята.

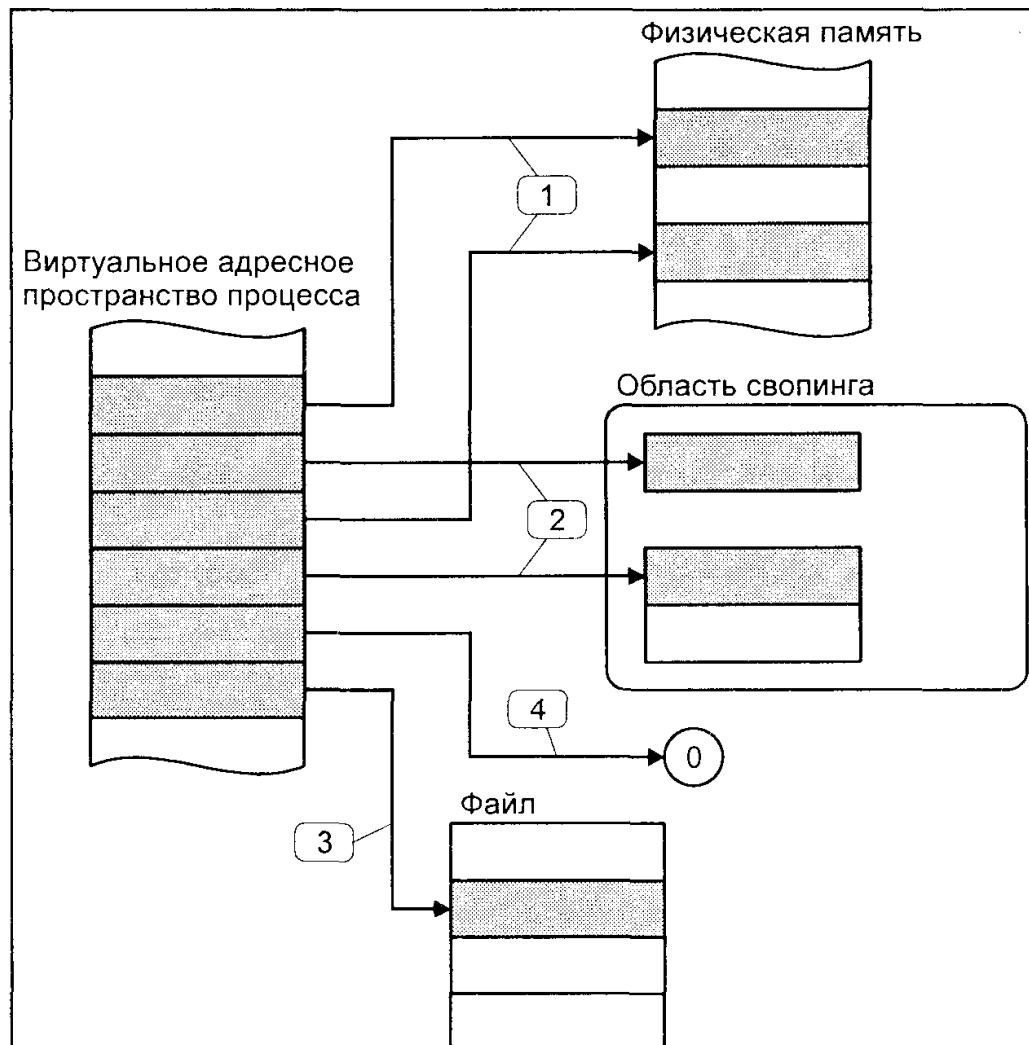


**Рис. 3.12.** Управление памятью, основанное на страничном замещении по требованию

Описанный механизм управления памятью допускает ситуацию, когда суммарный размер всех выполняющихся в данный момент процессов превышает размер физической памяти, в которой располагается только часть страниц процессов. Содержимое остальных страниц хранится вне физической памяти и должно быть загружено ядром, если процессу требуется доступ к этой части адресного пространства. Однако виртуальное адресное пространство процесса не зависит от фактического расположения физических страниц, и его размещение производится ядром при создании процесса или запуске новой программы. Виртуальное адресное пространство может изменяться в результате динамического размещения памяти (хипа) или увеличения стека процесса.

Таким образом, сам процесс "видит" только собственное виртуальное адресное пространство. Однако физические страницы, соответствующие

этому адресному пространству могут в действительности располагаться в различных местах, как это показано на рис. 3.13.



**Рис. 3.13.** Возможное местонахождение физических страниц процесса

1. Виртуальный адрес может быть ассоциирован со страницей физической памяти. Обращение к виртуальным адресам из диапазона, соответствующего этой странице, приведет к обращению к соответствующим адресам физической памяти. От операционной системы не требуется дополнительных действий при обращении к такой странице.
2. Страница может быть перемещена в область свопинга, если требуется освободить память для другого процесса. Обращение к виртуальному адресу, соответствующему этой странице, приведет к страницной ошибке, что, в свою очередь, потребует от ядра размещения новой страницы в памяти, записи ее содержимого из области свопинга и соответствующего изменения карты отображения (записи таблицы страниц) таким образом, чтобы виртуальный адрес указывал на новую страницу. Если потребуется опять переместить такую страницу в об-

ласть свопинга, ядро сделает это только в том случае, если с момента последней загрузки произошла модификация страницы.

3. Адресуемая страница отсутствует в памяти, но ее содержимое находится в файле на диске. Типичными примерами такой ситуации могут служить страницы сегмента кода или области файлов, отображенных в памяти. Обращение к виртуальному адресу, соответствующему этой странице, приведет к страницной ошибке, что, в свою очередь, потребует от ядра размещения новой страницы в памяти, записи ее содержимого из файла и соответствующего изменения карты отображения (записи таблицы страниц) таким образом, чтобы виртуальный адрес указывал на новую страницу.
4. Адресуемая страница отсутствует в памяти и она не ассоциирована ни с областью свопинга, ни с файлом. Типичным примером такой ситуации является страница сегмента неинициализированных данных. Обращение к такой странице потребует размещения новой страницы, заполненной нулями.

Ядро должно иметь достаточную информацию обо всех страницах, отсутствующих в памяти для того, чтобы при необходимости загрузить их в память. Для страниц, перемещенных во вторичную память, необходимо знать их расположение в области свопинга. Ядро должно иметь возможность распознать, что страницу необходимо заполнить нулями или загрузить ее содержимое из файла. В последнем случае ядро должно хранить местонахождение файла в файловой системе. Таким образом, наряду с картами отображения, необходимыми для трансляции адреса, ядро хранит ряд структур данных для поиска и загрузки отсутствующих в памяти страниц.

Различные версии UNIX используют разные подходы. Например, в SCO UNIX для описания страниц используются структуры pfdat и связанные с ними дескрипторы дисковых блоков. В UNIX 4.3BSD для этого используются поля записи таблицы страниц.

Страницное замещение имеет ряд важных преимуществ по сравнению со свопингом:

- Размер программы ограничивается лишь размером виртуальной памяти, который для компьютеров с 32-разрядной архитектурой составляет 4 Гбайт.
- СИ** Запуск программы происходит очень быстро, т. к. не требуется загружать в память всю программу целиком.
- Значительно большее число программ может быть загружено и выполняться одновременно, т. к. для выполнения каждой из них в каждый момент времени достаточно всего нескольких страниц.
- Перемещение отдельных страниц между оперативной и вторичной памятью требует значительно меньших затрат, чем перемещение процесса целиком.

## Планирование выполнения процессов

Как и оперативная память, процессор является разделяемым ресурсом, который должен быть справедливо распределен между конкурирующими процессами. Планировщик процессов как раз и является той подсистемой ядра, которая обеспечивает предоставление процессорных ресурсов процессам, выполняющимся в операционной системе. UNIX является системой разделения времени, это означает, что каждому процессу вычислительные ресурсы выделяются на ограниченный промежуток времени, после чего они предоставляются другому процессу и т. д. Максимальный временной интервал, на который процесс может захватить процессор, называется *временным квантом* (time quantum или time slice). Таким образом создается иллюзия, что процессы выполняются одновременно, хотя в действительности в каждый момент времени выполняется только один (на однопроцессорной системе) процесс.

UNIX является многозадачной системой, а это значит, что одновременно выполняются несколько приложений. Очевидно, что приложения предъявляют различные требования к системе с точки зрения их планирования и общей производительности. Можно выделить три основных класса приложений:

- *Интерактивные приложения.* К этому классу относятся командные интерпретаторы, текстовые редакторы и другие программы, непосредственно взаимодействующие с пользователем. Такие приложения большую часть времени обычно проводят в ожидании пользовательского ввода, например, нажатия клавиш клавиатуры или действия мышью. Однако они должны достаточно быстро обрабатывать такие действия, обеспечивая комфортное для пользователя время реакции. Допустимая задержка для таких приложений составляет от 100 до 200 миллисекунд.
- *Фоновые приложения.* К этому классу можно отнести приложения, не требующие вмешательства пользователя. Примерами таких задач могут служить компиляция программного обеспечения и сложные вычислительные программы. Для этих приложений важно минимизировать суммарное время выполнения в системе, загруженной другими процессами, порожденными, в частности, интерактивными задачами. Более того, предпочтительной является ситуация, когда интерактивные приложения не оказывают существенного влияния на среднюю производительность задач данного класса.
- *Приложения реального времени.* Хотя система UNIX изначально разрабатывалась как операционная система разделения времени, ряд приложений требуют дополнительных системных возможностей, в частности, гарантированного времени совершения той или иной операции, времени отклика и т. п. Примером могут служить измерительные комплексы или системы управления. Видеоприложения

также могут обладать определенными ограничениями на время обработки кадра изображения.

Планирование процессов построено на определенном наборе правил, исходя из которых планировщик выбирает, когда и какому процессу предоставить вычислительные ресурсы системы. При этом желательным является удовлетворение нескольких требований, например, минимальное время отклика для интерактивных приложений, высокая производительность для фоновых задач и т. п. Большинство из этих требований не могут быть полностью удовлетворены одновременно, поэтому в задачу планировщика процессов входит нахождение "золотой середины", обеспечивающей максимальную эффективность и производительность системы в целом.

В этом разделе мы рассмотрим основные принципы и механизмы планирования в традиционных UNIX-системах. Начнем с обработки прерываний таймера, поскольку именно здесь инициируются функции планирования и ряд других действий, например, отложенные вызовы (callout) и алармы (alarm).

## Обработка прерываний таймера

Каждый компьютер имеет аппаратный таймер или системные часы, которые генерируют аппаратное прерывание через фиксированные интервалы времени. Временной интервал между соседними прерываниями называется *тиком процессора* или просто *тиком* (CPU tick, clock tick). Как правило, системный таймер поддерживает несколько значений тиков, но в UNIX это значение обычно устанавливается равным 10 миллисекундам, хотя это значение может отличаться для различных версий операционной системы. Большинство систем хранят это значение в константе *HZ*, которая определена в файле заголовков `<param.h>`. Например, для тика в 10 миллисекунд значение *HZ* устанавливается равным 100.

Обработка прерываний таймера зависит от конкретной аппаратной архитектуры и версии операционной системы. Мы остановимся на принципах обработки прерываний, общих для большинства систем. Обработчик прерываний ядра вызывается аппаратным прерыванием таймера, приоритет которого обычно самый высокий. Таким образом, обработка прерывания должна занимать минимальное количество времени. В общем случае, обработчик решает следующие задачи:

- Обновление статистики использования процессора для текущего процесса
- Выполнение ряда функций, связанных с планированием процессов, например пересчет приоритетов и проверку истечения временного кванта для процесса
- Проверка превышения процессорной квоты для данного процесса и отправка этому процессу сигнала `SIGXCPU` в случае превышения

- П Обновление системного времени (времени дня) и других связанных с ним таймеров
- П Обработка отложенных вызовов (callout)
- П Обработка алармов (alarm)
- П Пробуждение в случае необходимости системных процессов, например диспетчера страниц и свопера

Часть перечисленных задач не требует выполнения на каждом тике. Большинство систем вводят нотацию *главного тика* (major tick), который проходит каждые  $n$  тиков, где  $n$  зависит от конкретной версии системы. Определенный набор функций выполняется только на главных тиках. Например, 4.3BSD производит пересчет приоритетов каждые 4 тика, а SVR4 обрабатывает алармы и производит пробуждение системных процессов раз в секунду.

## Отложенные вызовы

*Отложенный вызов* определяет функцию, вызов которой будет произведен ядром системы через некоторое время. Например, в SVR4 любая подсистема ядра может зарегистрировать отложенный вызов следующим образом:

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

где `fn()` определяет адрес функции, которую необходимо вызвать, при этом ей будет передан аргумент `arg`, а сам вызов будет произведен через `delta` ТИКОВ.

Ядро производит вызов `fn()` в системном контексте, таким образом функция отложенного вызова не должна обращаться к адресному пространству текущего процесса (поскольку не имеет к нему отношения), а также не должна переходить в состояние сна.

Отложенные вызовы применяются для выполнения многих функций, например:

- Выполнение ряда функций планировщика и подсистемы управления памятью
- П Выполнение ряда функций драйверов устройств для событий, вероятность ненаступления которых относительно велика. Примером может служить модуль протокола TCP, реализующий таким образом повторную передачу сетевых пакетов по тайм-ауту
- Опрос устройств, не поддерживающих прерывания

Заметим, что функции отложенных вызовов выполняются в системном контексте, а не в контексте прерывания. Вызов этих функций выполняется не обработчиком прерывания таймера, а отдельным обработчиком отложенных вызовов, который запускается после завершения обработки прес-

рывания таймера. При обработке прерывания таймера система проверяет необходимость запуска тех или иных функций отложенного вызова и устанавливает соответствующий флаг для них. В свою очередь обработчик отложенных вызовов проверяет флаги и запускает необходимые функции в системном контексте.

Эти функции хранятся в системной таблице отложенных вызовов, организация которой отличается для различных версий UNIX. Поскольку просмотр этой таблицы осуществляется каждый тик при обработке высокоприоритетного прерывания, для минимизации влияния этой операции на функционирование системы в целом, организация этой таблицы должна обеспечивать быстрый поиск нужных функций. Например, в 4.3BSD и SCO UNIX таблица отложенных вызовов организована в виде списка, отсортированного по времени запуска. Каждый элемент хранит разницу между временем вызова функции и временем вызова функции предыдущего элемента таблицы. На каждом тике значение этой величины уменьшается на единицу для первого элемента таблицы. Когда это значение становится равным 0, производится вызов соответствующей функции и запись удаляется. На рис. 3.14 приведена схема организации этой таблицы.

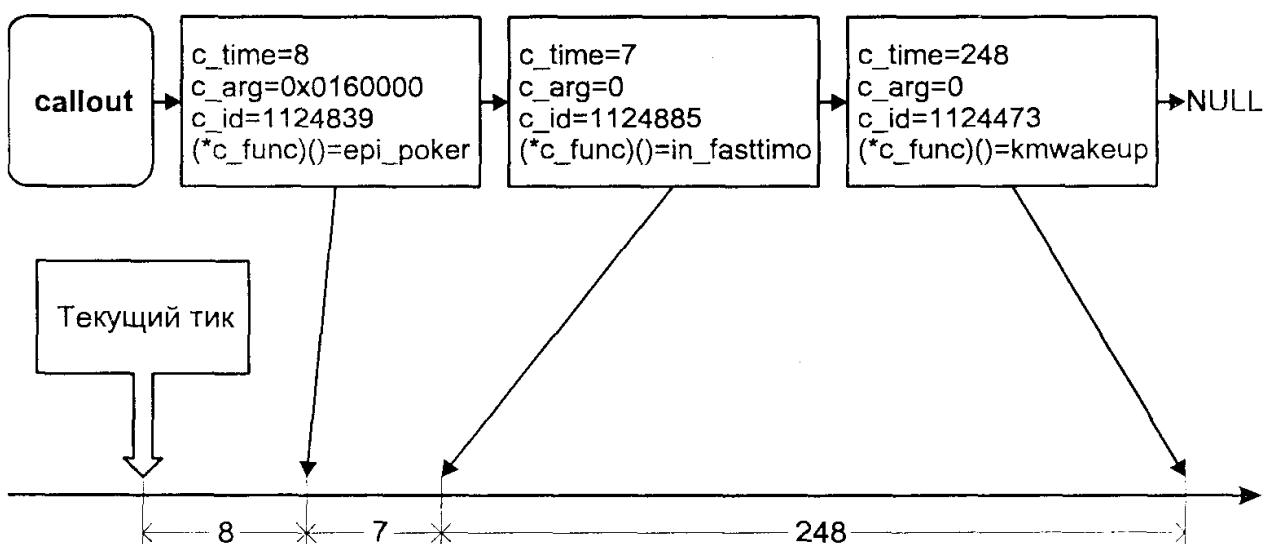


Рис. 3.14. Организация таблицы отложенных вызовов

## Алармы

Процесс может запросить ядро отправить сигнал по прошествии определенного интервала времени. Существуют три типа алармов — *реального времени* (real-time), *профилирования* (profiling) и *виртуального времени* (virtual time). С каждым из этих типов связан *таймер интервала* (interval timer, или itimer). Значение itimer уменьшается на единицу при каждом тике. Когда значение itimer достигает нуля, процессу отправляется соответствующий сигнал.

Указанные таймеры обладают следующими характеристиками:

ITIMER_REAL	Этот таймер используется для отсчета реального времени. Когда значение таймера становится равным нулю, процессу отправляется сигнал SIGALRM.
ITIMER_PROF	Этот таймер уменьшается только когда процесс выполняется в режиме ядра или задачи. Когда значение таймера становится равным нулю, процессу отправляется сигнал SIGPROF.
ITIMER_VIRT	Этот таймер уменьшается только когда процесс выполняется в режиме задачи. Когда значение таймера становится равным нулю, процессу отправляется сигнал SIGVTALRM.

В версиях BSD UNIX для установки таймеров всех трех типов используется системный вызов *settimer(2)*, для которого значение таймера устанавливается в микросекундах<sup>6</sup>. Ядро системы преобразует это значение в тики, на основании которых и производится уменьшение таймера. Напомним, что тик является максимальным времененным разрешением, которое может обеспечить система. В версиях System V для установки таймера реального времени используется вызов *alarm(2)*, позволяющий указать интервал в секундах. UNIX SVR4 позволяет установить таймеры высокого разрешения с помощью системного вызова *hrtsys(2)*, для которого время указывается в микросекундах. С помощью этого вызова также достигается совместимость с BSD, которая обеспечивается библиотечной функцией *settimer(3)*. Аналогично, в BSD UNIX вызов *alarm(3)* реализован в виде библиотечной функции.

Не следует, однако, заблуждаться насчет высокого разрешения таймеров реального времени. На самом деле их точность может быть довольно низкой. Допустим, что значение таймера реального времени, установленного каким-либо процессом, достигло нуля. При этом ядро отправит этому процессу сигнал SIGALRM. Однако процесс сможет получить и обработать этот сигнал, только когда он будет выбран планировщиком и поставлен на выполнение. В зависимости от приоритета процесса и текущей загрузки системы это может привести к существенным задержкам и, как следствие, к неточностям определения временного интервала. Таймеры реального времени высокого разрешения обладают достаточной точностью лишь для больших интервалов времени или для высокоприоритетных процессов. Тем не менее и для таких процессов получение сигнала может быть задержано, если в текущий момент процесс выполняется в режиме ядра и не может быть приостановлен.

Два других типа таймера обладают более высокой точностью, поскольку не имеют отношения к реальному течению времени. Однако их точность для малых временных интервалов может определяться следующим фактором.

<sup>6</sup> Некоторые системы System V, например SCO UNIX, также имеют в своем распоряжении этот системный вызов.

При обработке таймера процессу засчитывается тик целиком, даже если, предположим, процесс выполнялся лишь часть тика. Для временных интервалов порядка тика это может внести значительную погрешность.

## Контекст процесса

Каждый процесс UNIX имеет *контекст*, под которым понимается вся информация, требуемая для описания процесса. Эта информация сохраняется, когда выполнение процесса простанавливается, и восстанавливается, когда планировщик предоставляет процессу вычислительные ресурсы. Контекст процесса состоит из нескольких частей:

- **Адресное пространство процесса в режиме задачи.** Сюда входят код, данные и стек процесса, а также другие области, например, разделяемая память или код и данные динамических библиотек.
- **Управляющая информация.** Ядро использует две основные структуры данных для управления процессом — proc и user. Сюда же входят данные, необходимые для отображения виртуального адресного пространства процесса в физическое.
- **Окружение процесса.** Переменные окружения процесса представляют собой строки пар вида:

*переменная=значение*

которые наследуются дочерним процессом от родительского и обычно хранятся в нижней части стека. Окружение процесса упоминалось в предыдущих главах, там же были показаны функции, позволяющие получить или изменить переменные окружения.

- **Аппаратный контекст.** Сюда входят значения общих и ряда системных регистров процессора. К системным регистрам, в частности, относятся:
  - указатель инструкций, содержащий адрес следующей инструкции, которую необходимо выполнить;
  - указатель стека, содержащий адрес последнего элемента стека;
  - регистры плавающей точки;
  - регистры управления памятью, отвечающие за трансляцию виртуального адреса процесса в физический.

Переключение между процессами, необходимое для справедливого распределения вычислительного ресурса, по существу выражается в *переключении контекста*, когда контекст выполнявшегося процесса запоминается, и восстанавливается контекст процесса, выбранного планировщиком. Переключение контекста является достаточно ресурсоемкой операцией. Помимо сохранения состояния регистров процесса, ядро вынуждено выполнить множество других действий. Например, для некоторых систем ядру

необходимо очистить кэш данных, инструкций или адресных трансляций, чтобы предотвратить некорректные обращения нового процесса. Поэтому запущенный процесс сначала вынужден работать по существу без кэша, что также сказывается на производительности.

Существуют четыре ситуации, при которых производится переключение контекста:

1. Текущий процесс переходит в состояние сна, ожидая недоступного ресурса.
2. Текущий процесс завершает свое выполнение.
3. После пересчета приоритетов в очереди на выполнение находится более высокоприоритетный процесс.
4. Происходит пробуждение более высокоприоритетного процесса.

Первые два случая соответствуют добровольному переключению контекста и действия ядра в этом случае достаточно просты. Ядро вызывает процедуру переключения контекста из функций `sleep()` или `exit()`. Третий и четвертый случаи переключения контекста происходят не по воле процесса, который в это время выполняется в режиме ядра и поэтому не может быть немедленно приостановлен. В этой ситуации ядро устанавливает специальный флаг `runrun`, который указывает, что в очереди находится более высокоприоритетный процесс, требующий предоставления вычислительных ресурсов. Перед переходом процесса из режима ядра в режим задачи ядро проверяет этот флаг и, если он установлен, вызывает функцию переключения контекста.

## Принципы планирования процессов

Традиционные алгоритмы планирования UNIX обеспечивают возможность одновременного выполнения интерактивных и фоновых приложений. Таким образом, они хорошо подходят для систем общего назначения с несколькими подключенными пользователями, работающими с текстовыми и графическими редакторами, компилирующими программы и выполняющими вычислительные задачи. Эти алгоритмы обеспечивают малое время реакции для интерактивных приложений, следя в то же время, чтобы фоновым громоздким задачам справедливо предоставлялись ресурсы системы. Современные системы поддерживают выполнение задач реального времени, однако в данном разделе мы остановимся на планировании системы разделения времени.

Планирование процессов в UNIX основано на *приоритете* процесса. Планировщик всегда выбирает процесс с наивысшим приоритетом. Приоритет процесса не является фиксированным и динамически изменяется системой в зависимости от использования вычислительных ресурсов, времени ожидания запуска и текущего состояния процесса. Если процесс готов к

запуску и имеет наивысший приоритет, планировщик приостановит выполнение текущего процесса (с более низким приоритетом), даже если последний не "выработал" свой временной квант.

Традиционно ядро UNIX является "непрерываемым" (nonpreemptive). Это означает, что процесс, находящийся в режиме ядра (в результате системного вызова или прерывания) и выполняющий системные инструкции, не может быть прерван системой, а вычислительные ресурсы переданы другому, более высокоприоритетному процессу. В этом состоянии выполняющийся процесс может освободить процессор "по собственному желанию", в результате недоступности какого-либо ресурса перейдя в состояние сна. В противном случае система может прервать выполнение процесса только при переходе из режима ядра в режим задачи. Такой подход значительно упрощает решение задач синхронизации и поддержания целостности структур данных ядра.

Каждый процесс имеет два атрибута приоритета: *текущий приоритет*, на основании которого происходит планирование, и заказанный *относительный приоритет*, называемый nice number (или просто nice), который задается при рождении процесса и влияет на текущий приоритет.

Текущий приоритет варьируется в диапазоне от 0 (низкий приоритет) до 127 (наивысший приоритет). Процессы, выполняющиеся в режиме задачи, имеют более низкий приоритет, чем в режиме ядра. Для режима задачи приоритет меняется в диапазоне 0—65, для режима ядра — 66—95 (системный диапазон).

Процессы, приоритеты которых лежат в диапазоне 96—127, являются процессами с фиксированным приоритетом, не изменяемым операционной системой, и предназначены для поддержки приложений реального времени<sup>7</sup>.

Процессу, ожидающему недоступного в данный момент ресурса, система определяет значение *приоритета сна*, выбираемое ядром из диапазона системных приоритетов и связанное с событием, вызвавшее это состояние. В табл. 3.3 приведены значения приоритетов сна для систем 4.3BSD UNIX и SCO UNIX (OpenServer 5.0). Заметим, что направление роста значений приоритета для этих систем различно — в BSD UNIX большему значению соответствует более низкий приоритет.

<sup>7</sup> Схема нумерации текущих приоритетов различна для различных версий UNIX. Например, более высокому значению текущего приоритета может соответствовать более низкий фактический приоритет планирования. Разделение между приоритетами режима ядра и задачи также зависит от версии. Здесь мы привели схему, используемую в SCO UNIX, при которой большему значению соответствует более высокий приоритет.

**Таблица 3.3.** Системные приоритеты сна

Событие	Приоритет 4.3BSD UNIX	Приоритет SCO UNIX
Ожидание загрузки в память сегмента/страницы (свопинг/страничное замещение)	0	95
Ожидание индексного дескриптора	10	88
Ожидание ввода/вывода	20	81
Ожидание буфера	30	80
Ожидание терминального ввода		75
Ожидание терминального вывода		74
Ожидание завершения выполнения		73
Ожидание события — низкоприоритетное состояние сна	40	66

Когда процесс пробуждается, ядро устанавливает значение текущего приоритета процесса равным приоритету сна. Поскольку приоритет такого процесса находится в системном диапазоне и выше, чем приоритет режима задачи, вероятность предоставления процессу вычислительных ресурсов весьма велика. Такой подход позволяет, в частности, быстро завершить системный вызов, выполнение которого, в свою очередь, может блокировать некоторые системные ресурсы.

После завершения системного вызова перед возвращением в режим задачи ядро восстанавливает приоритет режима задачи, сохраненный перед выполнением системного вызова. Это может привести к понижению приоритета, что, в свою очередь, вызовет переключение контекста.

Текущий приоритет процесса в режиме задачи `p_priuser` зависит от двух факторов: значения `nice number` и степени использования вычислительных ресурсов `p_cpu`:

$$p_{\text{priuser}} = a * p_{\text{nice}} - b * p_{\text{cpu}},$$

где `p_nice` — постоянная составляющая, зависящая от параметра `nice`<sup>8</sup>.

Задача планировщика разделения времени — справедливо распределить вычислительный ресурс между конкурирующими процессами. Для принятия решения о выборе следующего запускаемого процесса планировщику необходима информация об использовании процессора. Эта составляющая

Мы специально не выделили явно параметр `nice` по следующей причине. Традиционно, большему значению параметра `nice` соответствует меньший приоритет, это уже обсуждалось в главе 1. В данном обсуждении выбрана схема, при которой большему значению `p_cri` соответствует больший приоритет. Поэтому в простейшем случае коэффициент `a` является отрицательным, а `p_nice` равно значению параметра `nice` (`nice number`).

приоритета уменьшается обработчиком прерываний таймера каждый тик. Таким образом, пока процесс выполняется в режиме задачи, его текущий приоритет линейно уменьшается.

Каждую секунду ядро пересчитывает текущие приоритеты процессов, готовых к запуску (приоритеты которых меньше 65), последовательно увеличивая их<sup>9</sup>. Это перемещает процессы в более приоритетные очереди и повышает вероятность их последующего запуска.

Например, UNIX версии SVR3, использует следующую формулу:

$$p_{\text{cpu}} = p_{\text{cpu}}/2$$

Эта простая схема проявляет недостаток нивелирования приоритетов при повышении загрузки системы. Это происходит потому, что в этом случае каждый процесс получает незначительный объем вычислительных ресурсов и следовательно имеет малую составляющую  $p_{\text{cpu}}$ , которая еще более уменьшается благодаря формуле пересчета  $p_{\text{cpu}}$ . В результате степень использования процессора перестает оказывать заметное влияние на приоритет, и низкоприоритетные процессы (т. е. процессы с высоким nice number) практически "отлучаются" от вычислительных ресурсов системы.

В 4.3BSD UNIX для пересчета  $p_{\text{cpu}}$  используется другая формула:

$$p_{\text{cpu}} = p_{\text{cpu}} * (2 * \text{load}) / (2 * \text{load} + 1)$$

Здесь параметр  $\text{load}$  равен среднему числу процессов, находившихся в очереди на выполнение за последнюю секунду, и характеризует среднюю загрузку системы за этот период времени. Этот алгоритм позволяет частично избавиться от недостатка планирования SVR3, поскольку при значительной загрузке системы уменьшение  $p_{\text{cpu}}$  при пересчете будет происходить медленнее.

Описанные алгоритмы планирования позволяют учесть интересы низкоприоритетных процессов, т. к. в результате длительного ожидания очереди на запуск приоритет таких процессов увеличивается, соответственно увеличивается и вероятность запуска. Представленные алгоритмы также обеспечивают более вероятный выбор планировщиком интерактивных процессов по отношению к вычислительным (фоновым). Такие задачи, как командный интерпретатор или редактор, большую часть времени проводят в ожидании ввода, имея, таким образом, высокий приоритет (приоритет сна). При наступлении ожидаемого события (например, пользователь осуществил ввод данных) им сразу же предоставляются вычислительные ресурсы. Фоновые процессы, потребляющие значительные ресурсы процессора, имеют высокую составляющую  $p_{\text{cpu}}$  и, как следствие, более низкий приоритет.

<sup>9</sup> Ядро последовательно уменьшает отрицательную компоненту времени использования процессора.

Как правило, очередь на выполнение не одна. Например, SCO UNIX имеет 127 очередей — по одной на каждый приоритет. BSD UNIX использует 32 очереди, каждая из которых обслуживает диапазон приоритетов, например 0—3, 4—7 и т. д. При выборе следующего процесса на выполнение из одной очереди, т. е. из нескольких процессов с одинаковым текущим приоритетом, используется механизм *кругового чередования* (round robin)<sup>10</sup>. Этот механизм запускается ядром через каждый временной квант для наиболее приоритетной очереди. Однако если в системе появляется готовый к запуску процесс с более высоким приоритетом, чем текущий, он будет запущен, не дожидаясь прошествия временного кванта. С другой стороны, если все процессы, готовые к запуску, находятся в низкоприоритетных по отношению к текущему процессу очередях, последний будет продолжать выполняться и в течение следующего временного кванта.

## Создание процесса

Как уже обсуждалось, в UNIX проведена четкая грань между *программой* и *процессом*. Каждый процесс в конкретный момент времени выполняет инструкции некоторой программы, которая может быть одной и той же для нескольких процессов<sup>11</sup>. Примером может служить командный интерпретатор, с которым одновременно работают несколько пользователей, таким образом инструкции программы shell выполняют несколько различных процессов. Такие процессы могут совместно использовать один сегмент кода в памяти, но в остальном они являются изолированными друг от друга и имеют собственные сегменты данных и стека.

В любой момент процесс может запустить другую программу и начать выполнять ее инструкции; такую операцию он может сделать несколько раз.

В операционной системе UNIX имеются отдельные системные вызовы для создания (порождения) процесса, и для запуска новой программы. Системный вызов *fork(2)* создает новый процесс, который является точной копией родителя. После возвращения из системного вызова оба процесса выполняют инструкции одной и той же программы и имеют одинаковые сегменты данных и стека.

Тем не менее между родительским и дочерним процессом имеется ряд различий:

- Дочернему процессу присваивается уникальный идентификатор PID, отличный от родительского.

<sup>10</sup> Round robin (англ.) означает петицию, подписи под которой располагаются по кругу — чтобы нельзя было определить, кто подписался первым. Отсюда и название схемы выбора процессов.

<sup>11</sup> Естественно, речь здесь идет о выполнении в режиме задачи, в режиме ядра процесс выполняет инструкции ядра операционной системы.

## Создание процесса

- Соответственно и идентификатор родительского процесса PPID для родителя и потомка различны.
- Дочерний процесс получает собственную копию и-агса и, в частности, собственные файловые дескрипторы, хотя он разделяет те же записи файловой таблицы.
- Для дочернего процесса очищаются все ожидающие доставки сигналы.
- Временная статистика выполнения процесса в режиме ядра и задачи для дочернего процесса обнуляется.
- Блокировки памяти и записей, установленные родительским процессом, потомком не наследуются.

Более подробно наследуемые характеристики представлены в табл. 3.4.

**Таблица 3.4.** Наследование установок при создании процесса и запуске программы

Атрибут	Наследование по- <b>томком (fork(2))</b>	Сохранение при запус- <b>ке программы (exec(2))</b>
Сегмент кода (text)	Да, разделяемый	Нет
Сегмент данных (data)	Да, копируется при записи (copy-on-write)	Нет
Окружение	Да	Возможно
Аргументы	Да	Возможно
Идентификатор пользователя UID	Да	Да
Идентификатор группы GID	Да	Да
Эффективный идентификатор пользователя EUID	Да	Да (Нет, при вызове <i>setuid(2)</i> )
Эффективный идентификатор группы EGID	Да	Да (Нет, при вызове <i>setgid(2)</i> )
ID процесса (PID)	Нет	Да
ID группы процессов	Да	Да
ID родительского процесса (PPID)	Нет	Да
Приоритет nice number	Да	Да
Права доступа к создаваемому файлу	Да	Да
Ограничение на размер файла	Да	Да
Сигналы, обрабатываемые по умолчанию	Да	Да
Игнорируемые сигналы	Да	Да

Таблица 3.4 (продолжение)

Атрибут	Наследование потомком ( <b>fork(2)</b> )	Сохранение при запуске программы ( <b>exec(2)</b> )
Перехватываемые сигналы	Да	Нет
Файловые дескрипторы	Да	Да, если для файлового дескриптора не установлен флаг FD_CLOEXEC (например, с помощью <i>fcntl(2)</i> )
Файловые указатели	Да, разделяемые	Да, если для файлового дескриптора не установлен флаг FD_CLOEXEC (например, с помощью <i>fcntl(2)</i> )

В общем случае вызов *fork(2)* выполняет следующие действия:

- П Резервирует место в области свопинга для сегмента данных и стека процесса.
- П Размещает новую запись *proc* в таблице процессов и присваивает процессу уникальный идентификатор PID.
- П Инициализирует структуру *proc* (поля структуры *proc* подробно рассматривались в разделе "Структуры данных процесса").
- П Размещает карты отображения, необходимые для трансляции адреса.
- Размещает *u-area* процесса и копирует ее содержимое с родительского.
- П Создает соответствующие области процесса, часть из которых совпадает с родительскими.
- Инициализирует аппаратный контекст процесса, копируя его с родительского.
- Устанавливает в ноль возвращаемое дочернему процессу вызовом *fork(2)* значение.
- П Устанавливает возвращаемое родительскому процессу вызовом *fork(2)* значение равным PID потомка.
- П Помечает процесс готовым к запуску и помещает его в очередь на выполнение.

Системный вызов *fork(2)* в итоге создает для дочернего процесса отдельную копию адресного пространства родителя. Во многих случаях, вскоре после этого, дочерний процесс делает системный вызов *exec(2)* для запуска новой программы, при этом существующее адресное пространство уничтожается и создается новое. Таким образом создание фактической копии адресного пространства процесса, т. е. выделение оперативной памяти и создание соответствующих карт отображения, является неоправданным.

Для решения данной проблемы используются два подхода. Первый из них, предложенный в UNIX System V, называется "копирование при записи" (copy-on-write или COW). Суть этого подхода заключается в том, что сегменты данных и стека родительского процесса помечаются доступными только для чтения, а дочерний процесс, хотя и получает собственные карты отображения, разделяет эти сегменты с родительским. Другими словами, сразу после создания процесса и родитель и потомок адресуют одни и те же страницы физической памяти. Если какой-либо из двух процессов попытается модифицировать данные или стек, возникнет страницчная ошибка, поскольку страница открыта только для чтения, а не для записи. При этом будет запущен обработчик ошибки ядра, который создаст для процесса копию этой страницы, доступную для записи. Таким образом, фактическому копированию подлежат только модифицируемые страницы, а не все адресное пространство процесса. Если дочерний процесс делает системный вызов *exec(2)* или вообще завершает свое выполнение, права доступа к страницам родителя, имеющим флаг COW, возвращаются к их прежним значениям (т. е. до создания дочернего процесса), а флаг COW очищается.

Другой подход используется в BSD UNIX. В этой версии системы был предложен новый системный вызов — *vfork(2)*. Использование этого вызова имеет смысл, когда дочерний процесс сразу же выполняет вызов *exec(2)* и запускает новую программу. При вызове *vfork(2)* родительский процесс предоставляет свое адресное пространство дочернему и переходит в состояние сна, пока последний не вернет его обратно. Далее дочерний процесс выполняется в адресном пространстве родителя, пока не делает вызов *exec(2)* или *exit(2)*, после чего ядро возвращает адресное пространство родителю и пробуждает его. С помощью *vfork(2)* можно добиться максимального быстродействия, т. к. в этом случае мы полностью избегаем копирования, даже для карт отображения. Вместо этого адресное пространство родительского процесса предоставляется потомку передачей нескольких аппаратных регистров, отвечающих за трансляцию адресов. Однако *vfork(2)* таит в себе потенциальную опасность, поскольку позволяет одному процессу использовать и даже модифицировать адресное пространство другого.

Для управления памятью процесса ядру необходимо соответствующим образом задать области. При этом структуры *preion* дочернего процесса, соответствующие разделяемым областям, указывают на те же структуры *region*, что и для родителя. Для областей, совместное использование которых недопустимо, ядро размещает отдельные структуры *region* для дочернего процесса (изначально копируя их содержимое с родительского) и устанавливает соответствующие указатели. На рис. 3.15 представлена схема этих операций. Заметим, что совместная работа и дублирование областей являются отдельным механизмом, не связанным с рассмотренными выше подходами, для совместного использования адресного пространства, например COW. Так, после создания отдельной копии неразделяемой облас-

ти она по-прежнему будет адресовать те же страницы памяти, что и соответствующая область родителя.

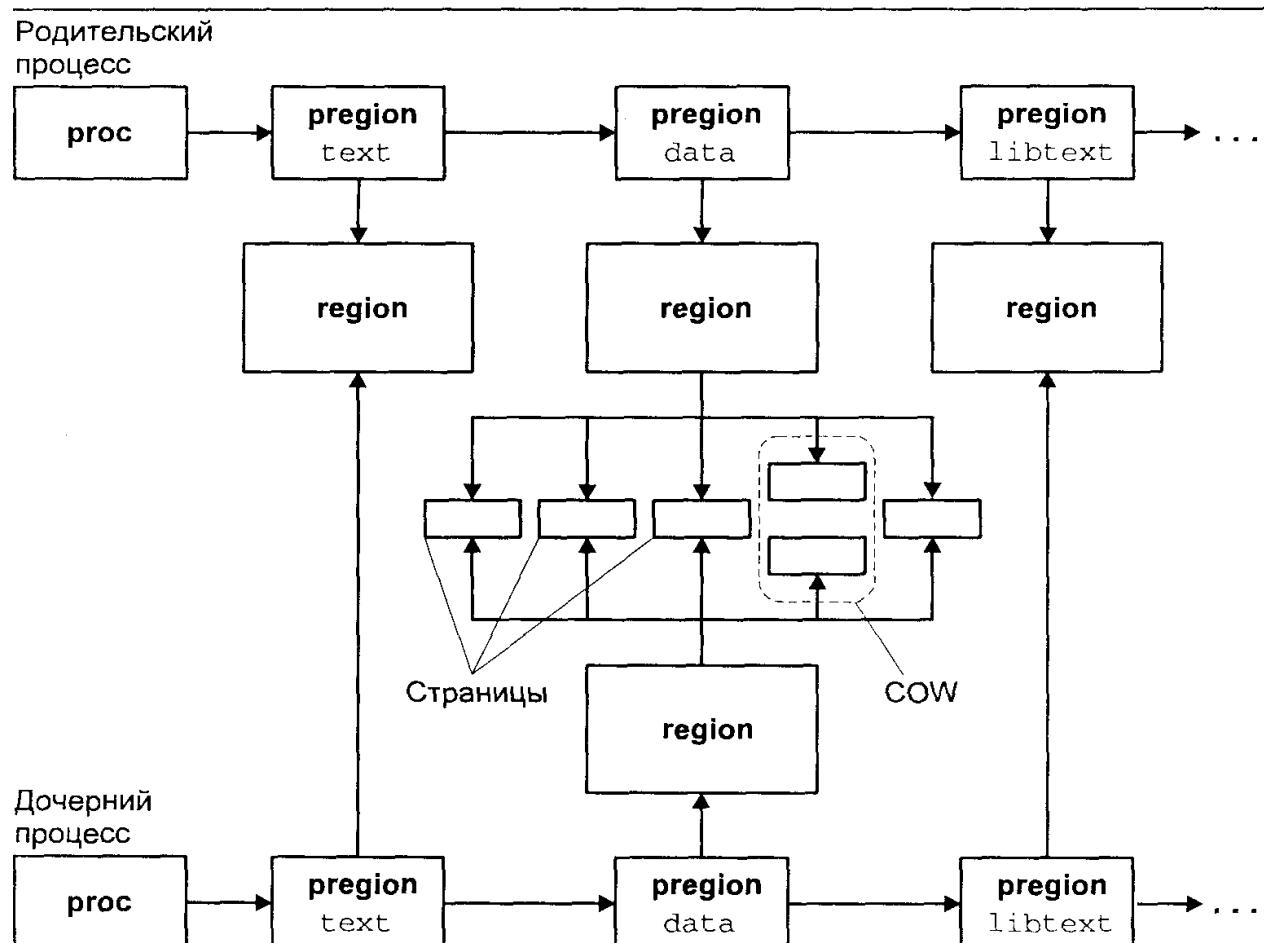


Рис. 3.15. Создание областей нового процесса

### Запуск новой программы

Запуск новой программы осуществляется с помощью системного вызова `exec(2)`. Напомним, что при этом создается не новый процесс, а новое адресное пространство процесса, которое загружается содержимым новой программы. Если процесс был создан вызовом `vfork(2)`, старое адресное пространство возвращается родителю, в противном случае оно просто уничтожается. После возврата из вызова `exec(2)` процесс продолжает выполнение кода новой программы.

Операционная система UNIX обычно поддерживает несколько форматов исполняемых файлов. Старейший из них — a.out, в разделе "Форматы исполняемых файлов" главы 2 также были рассмотрены форматы COFF и ELF. В любом случае исполняемый файл содержит заголовок, позволяющий ядру правильно разместить адресное пространство процесса и загрузить в него соответствующие фрагменты исполняемого файла.

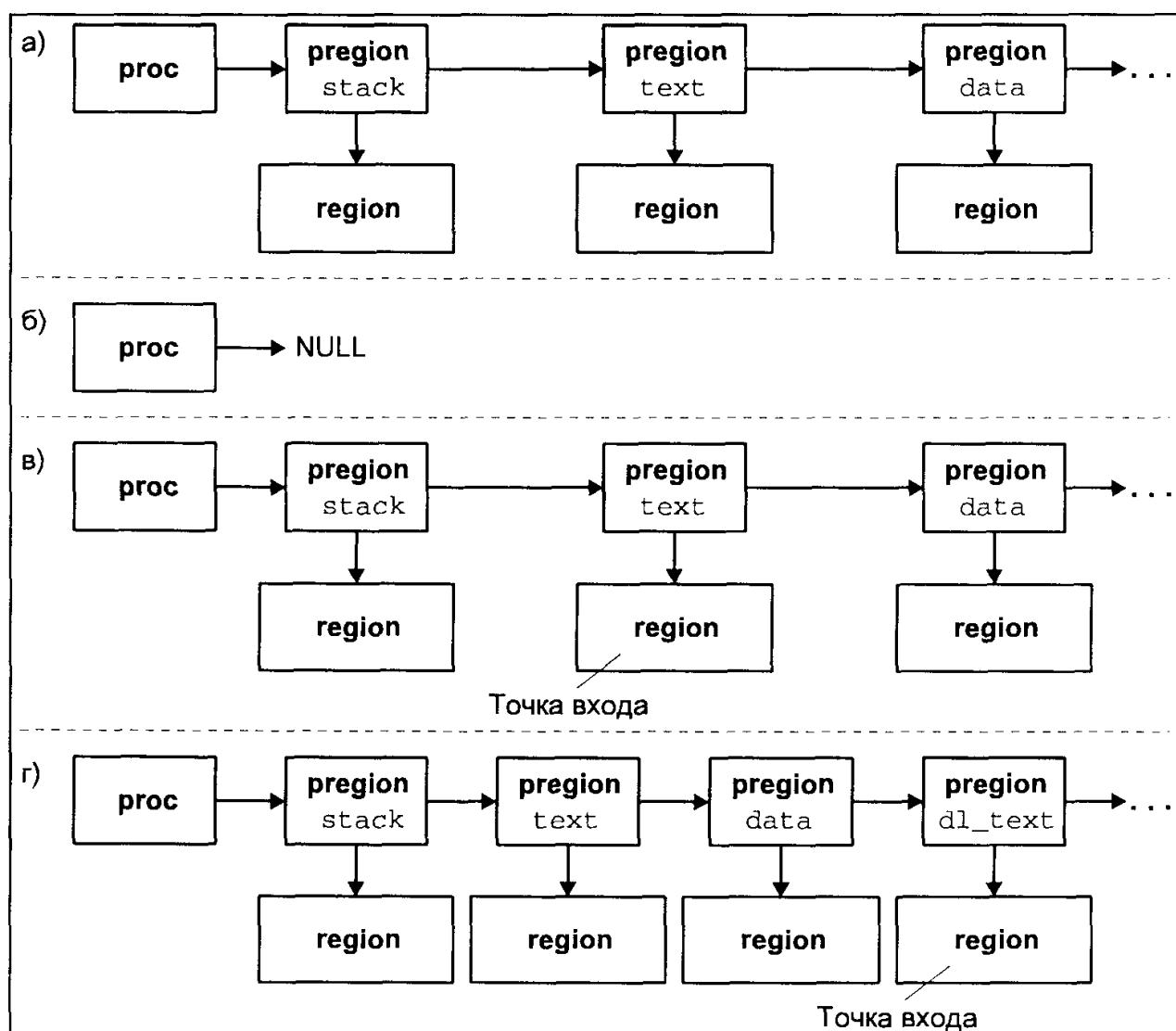
Перечислим ряд действий, которые выполняет *exec(2)* для запуска новой программы:

- Производит трансляцию имени файла. В результате возвращается индексный дескриптор, с помощью которого осуществляется доступ к файлу. При этом проверяются права доступа.
- Считывает заголовок файла и проверяет, является ли файл исполняемым. Вызов *exec(2)* также распознает скрипты, о которых говорилось в главе 1. При этом он анализирует первую строку скрипта, которая обычно имеет вид `#!/shellname`. В этом случае *exec(2)* запускает программу, указанную `shellname`, передавая ей в качестве аргумента имя скрипта. Если исполняемый файл (т. е. файл с установленным атрибутом x) не является бинарным и не содержит в первой строке названия интерпретатора, *exec(2)* запускает интерпретатор по умолчанию (`/bin/sh`, `/usr/bin/sh`, или `/usr/bin/ksh`, как предписывает стандарт XPG4), передавая ему содержимое файла в качестве ввода.
- Если исполняемый файл имеет атрибуты SUID или SGID, *exec(2)* соответствующим образом изменяет эффективные идентификаторы UID и GID для этого процесса<sup>12</sup>.
- Сохраняет аргументы вызова *exec(2)* и переменные окружения в адресном пространстве ядра, поскольку адресное пространство процесса будет уничтожено.
- Резервирует место в области свопинга для сегмента данных и стека.
- Освобождает старые области процесса и соответствующие области свопинга. Если процесс был создан вызовом *ufork(2)*, старое адресное пространство возвращается родителю.
- Размещает и инициализирует карты отображения для новых сегментов кода, данных и стека. Если сегмент кода является активным, например, какой-либо процесс уже выполняет эту программу, данная область используется совместно. В противном случае область заполняется содержимым соответствующего раздела исполняемого файла или инициализируется нулями для неинициализированных данных. Поскольку управление памятью процесса построено на механизме страничного замещения по требованию, копирование происходит постранично и только тогда, когда процесс обращается к страницам, отсутствующим в памяти.
- Копирует сохраненные аргументы и переменные окружения в новый стек процесса.
- Устанавливает обработку всех сигналов на умалчивающие значения, поскольку процесс теперь не имеет требуемых обработчиков. Установки для игнорируемых и заблокированных сигналов не изменяются.

<sup>12</sup> Напомним, что в этом случае EUID и EGID не наследуются от родительского процесса, а присваиваются равными идентификаторам UID и GID исполняемого файла.

- Инициализирует аппаратный контекст процесса. В частности, после этого указатель инструкций адресует точку входа новой программы.

В случае, когда программа использует динамические библиотеки, соответствующий раздел исполняемого файла (для файла формата ELF данный раздел имеет тип INTERP) содержит имя редактора связей динамической библиотеки. В этом случае редактор связей должен быть запущен до начала выполнения основной программы для связывания с программами требуемых динамических библиотек. Таким образом точка входа в программу устанавливается на точку входа в редактор связей. После завершения своей работы редактор связей, в свою очередь, запускает программу самостоятельно, анализируя заголовок исполняемого файла. Стадии запуска новой программы проиллюстрированы на рис. 3.16.



**Рис. 3.16.** Запуск новой программы: а) Адресное пространство процесса до вызова `exec(2)`; б) Уничтожение старого адресного пространства; в) Новое адресное пространство процесса; г) Новое адресное пространство процесса при использовании динамических библиотек

## Выполнение в режиме ядра

Существуют всего три события, при которых выполнение процесса переходит в режим ядра — аппаратные прерывания, особые ситуации и системные вызовы. Во всех случаях ядро UNIX получает управление и вызывает соответствующую системную процедуру для обработки события. Перед вызовом ядро сохраняет состояние прерванного процесса в системном стеке. После завершения обработки, состояние процесса восстанавливается и процесс возвращается в исходный режим выполнения. Чаще всего это режим задачи, но если, например, прерывание возникло, когда процесс уже находился в режиме ядра, после обработки события он останется в этом режиме.

Отметим существенную разницу между прерываниями и особыми ситуациями. Аппаратные прерывания генерируются периферийными устройствами при наступлении определенных событий (например, завершение дисковой операции ввода/вывода или поступление данных на последовательный порт) и имеют асинхронный характер, поскольку невозможно точно сказать, в какой момент наступит то или иное прерывание. Более того, эти прерывания, как правило, не связаны с текущим процессом, а вызваны внешними событиями. Именно поэтому, обработка прерываний происходит в системном контексте, при этом недопустим доступ к адресному пространству процесса, например, к его u-area. По этой же причине, обработка прерываний не должна блокироваться, поскольку это вызовет блокирование выполнения независимого процесса.

Напротив, особые ситуации вызваны самим процессом, и связаны с выполнением тех или иных инструкций, например, деление на ноль или обращение к несуществующей странице памяти. Таким образом, обработка особых ситуаций производится в контексте процесса, при этом может использоваться его адресное пространство, а сам процесс — при необходимости блокироваться (перемещаться в состояние сна).

Системные вызовы позволяют процессам воспользоваться базовыми услугами ядра. Интерфейс системных вызовов определяет ограниченный набор точек входа в ядро системы, обращение к которым изменяет режим выполнения процесса и позволяет выполнять привилегированные инструкции ядра. Стандартная библиотека C, позволяющая использовать системные функции как обычные процедуры, на самом деле содержит заглушки, обеспечивающие фактическую реализацию вызова соответствующей точки входа ядра. Эта реализация существенным образом зависит от аппаратной архитектуры системы. Например, для систем на базе процессоров Intel используются *шлюзы* (gate). Имеются два типа шлюзов: *шлюзы ловушек* (trap gate) и *шлюзы вызовов* (call gate). Для осуществления вызова через шлюз ловушки процесс выполняет команду прерывания, а при работе через шлюз вызова — команду межсегментного вызова.

Выполнение системного вызова происходит в режиме ядра, но в контексте процесса, сделавшего системный вызов. Таким образом, открыт доступ к адресному пространству процесса и используется стек ядра процесса.

## **Сон и пробуждение**

Процесс обычно переводится в состояние сна при обработке системной функции. Если для завершения обработки запроса требуется недоступный ресурс, процесс снимается с процессора и переводится в состояние сна. Недоступность ресурса может быть связана с запуском операции ввода/вывода с диска, ожиданием выделения (освобождения) буфера, ожиданием ввода или вывода на терминал или ожиданием завершения дочернего процесса. К недоступным ресурсам можно также отнести отсутствующую в памяти страницу, к виртуальному адресу которой обратился процесс. В любом случае процесс переходит в состояние сна до наступления события, делающего ресурс доступным. Во время сна процесс не потребляет вычислительные ресурсы системы. При этом выполняется переключение контекста на другой, высокоприоритетный процесс для выполнения. Таким образом, процесс, ожидающий ввода с клавиатуры, не занимает процессор, циклически опрашивая терминальную линию, а процесс,читывающий данные с диска, не блокирует выполнение других задач.

Состояние сна — это логическое состояние процесса, при этом он не перемещается физически в памяти. Переход в состояние сна в первую очередь определяется занесением в системную таблицу процессов соответствующего флага состояния и события, пробуждающего процесс.

События возвещают о доступности того или иного ресурса. Как правило события связаны с работой периферийных устройств, таких как диск, терминал и принтер, поэтому об их наступлении сигнализируют соответствующие аппаратные прерывания. Наступления одного и того же события может ожидать несколько процессов. Поскольку переход из состояния в состояние акт скорее логический, то и пробуждаются все эти процессы одновременно. Однако это не означает, что какой-либо один из них сразу начнет выполняться. Это лишь приводит к тому, что их состояние меняется от "сна" к "готов к выполнению", и они помещаются в очередь на запуск. Задачу выбора процесса для запуска затем решает планировщик процессов.

События, в ожидании которых "засыпают" процессы, не являются равнозначными.

Во-первых, они различаются по вероятности наступления. Например событие, связанное с завершением операции ввода с диска или освобождением буфера, имеет высокую вероятность. Как правило, подобные операции имеют конечное время выполнения, в противном случае система ока-

заявилась бы заблокированной. С другой стороны, вероятность наступления события, связанного с вводом с терминала, может быть весьма низкой. Пользователь может надолго оставить терминал, не завершив сеанса работы с системой. В длительном ожидании события нет ничего опасного — процесс не занимает ресурсы процессора, однако без специальных мер выключение терминала приведет к блокировке этого устройства. Для того чтобы избежать подобной ситуации, должна существовать возможность вывести процесс из состояния сна, несмотря на отсутствие ожидаемого события. В этом случае используется стандартное решение — отправление процессу сигнала. В противоположность этому, отправление сигнала процессу, ожидающему операции ввода с диска, может привести к ухудшению производительности системы. Поэтому все события и связанные с ними ресурсы разделяются на две категории по вероятности их наступления: на *допускающие* прерывание сигналом и на *не допускающие* таковых.

Во-вторых, процессы, разбуженные событием, должны иметь различную вероятность запуска. Это, в первую очередь, связано с тем, что несколько ресурсов могут отображаться на одно событие. Например, процесс А, ожидающий завершения операции ввода с диска, и процесс В, ожидающий освобождения буфера ввода, будут связаны с одним и тем же событием. Они оба окажутся "разбуженными" и затем "готовыми к запуску" после завершения этой операции. Если процесс В будет запущен первым, он все равно не сможет выполняться, так как буфер не освобожден процессом А. Даже в случае, когда спящие процессы связаны с различными событиями, необходимо отдавать предпочтение процессу с более ценным ресурсом. Например, освобождение буфера ввода безусловно предпочтительнее завершения ввода с терминала.

Поскольку планировщик принимает решение о запуске процесса, основываясь на приоритетах, единственным способом установить "справедливый" порядок запуска процессов является присвоение определенного приоритета каждому событию. Приоритет процесса и его влияние на планирование достаточно подробно обсуждались в разделе "Контекст процесса".

## **Завершение выполнения процесса**

Процесс завершает свое выполнение с помощью функции `exit()`. Эта функция может быть вызвана системным вызовом `exit(2)`, а если завершение процесса вызвано получением сигнала, функцию `exit()` вызывает само ядро. Функция `exit()` выполняет следующие действия:

- Отключает все сигналы.
- Закрывает все открытые файлы.
- Сохраняет статистику использования вычислительных ресурсов и код возврата в записи `proc` таблицы процессов.

- Изменяет состояние процесса на "зомби".
- Делает процесс *init(1M)* родительским для всех потомков данного процесса.
- П Освобождает адресное пространство процесса, u-area, карты отображения и области свопинга, связанные с процессом.
- П Отправляет сигнал SIGCHLD родительскому процессу, уведомляя его о "смерти" потомка.
- П Пробуждает родительский процесс, если тот ожидает завершения потомка.
- П Запускает функцию переключения контекста, в результате чего высокоприоритетный процесс получает доступ к вычислительным ресурсам.

После завершения выполнения функции `exit()` процесс находится в состоянии "зомби". При этом от процесса остается запись `proc` в таблице процессов, содержащая статистику использования вычислительных ресурсов и код возврата. Эта информация может потребоваться родительскому процессу, поэтому освобождение структуры `proc` производит родитель с помощью системного вызова `wait(2)`, возвращающего статистику и код возврата потомка. Если родительский процесс заканчивает свое выполнение раньше потомка, "родительские права" переходят к процессу *init(1M)*. В этом случае после смерти потомка *init(1M)* делает системный вызов `wait(2)` и освобождает структуру `proc`.

Другая ситуация возникает, если потомок заканчивает свое выполнение раньше родителя, а родительский процесс не производит вызова `wait(2)`. В этом случае структура `proc` потомка не освобождается и процесс продолжает находиться в состоянии "зомби" до перезапуска операционной системы. Хотя такой процесс (которого, вообще говоря, не существует) не потребляет ресурсов системы, он занимает место в таблице процессов, тем самым уменьшая максимальное число активных задач.

## Сигналы

В некотором смысле сигналы обеспечивают простейшую форму межпроцессного взаимодействия, позволяя уведомлять процесс или группу процессов о наступлении некоторого события. Мы уже рассмотрели в предыдущих главах сигналы с точки зрения пользователя и программиста. Теперь мы остановимся на обслуживании сигналов операционной системой.

## Группы и сеансы

Группы процессов и сеансы уже обсуждались в главе 2. Такое представление набора процессов используется в UNIX для управления доступом к

терминалу и поддержки пользовательских сеансов работы в системе. Перечислим еще раз наиболее важные понятия, связанные с группами и сессиями.

- *Группа процессов.* Каждый процесс принадлежит определенной группе процессов. Каждая группа имеет уникальный идентификатор. Группа может иметь в своем составе *лидера группы* — процесс, чей идентификатор PID равен идентификатору группы. Обычно процесс наследует группу от родителя, но может покинуть ее и организовать собственную группу.
- *Управляющий терминал.* Процесс может быть связан с терминалом, который называется управляющим. Все процессы группы имеют один и тот же управляющий терминал.
- *Специальный файл устройства /dev/tty.* Этот файл связан с управляющим терминалом процесса. Драйвер для этого псевдоустройства по существу перенаправляет запросы на фактический терминальный драйвер, который может быть различным для различных процессов. Например, два процесса, принадлежащие различным сессиям, открывая файл /dev/tty, получат доступ к различным терминалам.

## Управление сигналами

Сигналы обеспечивают механизм вызова определенной процедуры при наступлении некоторого события. Каждое событие имеет свой идентификатор и символьную константу. Некоторые из этих событий имеют асинхронный характер, например, когда пользователь нажимает клавишу *<Del>* или *<Ctrl>+<C>* для завершения выполнения процесса, другие являются уведомлением об ошибках и особых ситуациях, например, при попытке доступа к недопустимому адресу или вызовы недопустимой инструкции. Различные события, соответствующие тем или иным сигналам, подробно рассматривались в главе 2.

Говоря о сигналах необходимо различать две фазы этого механизма — генерация или отправление сигнала и его доставка и обработка. Сигнал отправляется, когда происходит определенное событие, о наступлении которого должен быть уведомлен процесс. Сигнал считается доставленным, когда процесс, которому был отправлен сигнал, получает его и выполняет его обработку. В промежутке между этими двумя моментами сигнал ожидает доставки.

## Отправление сигнала

Ядро генерирует и отправляет процессу сигнал в ответ на ряд событий, которые могут быть вызваны самим процессом, другим процессом, прерыванием или какими-либо внешними событиями. Можно выделить основные причины отправки сигнала:

**Особые ситуации**

Когда выполнение процесса вызывает особую ситуацию, например, деление на ноль, процесс получает соответствующий сигнал.

**Терминальные прерывания**

Нажатие некоторых клавиш терминала, например,  $<Del>$ ,  $<Ctrl>+<C>$  или  $<Ctrl>+<|>$ , вызывает отправление сигнала текущему процессу, связанному с терминалом.

**Другие процессы**

Процесс может отправить сигнал другому процессу или группе процессов с помощью системного вызова *kill(2)*. В этом случае сигналы являются элементарной формой межпроцессного взаимодействия.

**Управление заданиями**

Командные интерпретаторы, поддерживающие систему управления заданиями, используют сигналы для манипулирования фоновым и текущими задачами. Когда процесс, выполняющийся в фоновом режиме делает попытку чтения или записи на терминал, ему отправляется сигнал останова. Когда дочерний процесс завершает свою работу, родитель уведомляется об этом также с помощью сигнала.

**Квоты**

Когда процесс превышает выделенную ему квоту вычислительных ресурсов или ресурсов файловой системы, ему отправляется соответствующий сигнал.

**Уведомления**

Процесс может запросить уведомление о наступлении тех или иных событий, например, готовности устройства и т. д. Такое уведомление отправляется процессу в виде сигнала.

**Алармы**

Если процесс установил таймер, ему будет отправлен сигнал, когда значение таймера станет равным нулю.

**Доставка и обработка сигнала**

Для каждого сигнала в системе определена обработка по умолчанию, которую выполняет ядро, если процесс не указал другого действия. В общем случае существуют пять возможных действий: завершить выполнение процесса (с созданием образа *core* и без), игнорировать сигнал, остановить процесс и продолжить процесс (справедливо для остановленного процесса, для остальных сигнал игнорируется), наиболее употребительным из которых является первое.

Как уже обсуждалось в главе 2, процесс может изменить действие по умолчанию, либо зарегистрировав собственный обработчик сигнала, либо указав, что сигнал следует игнорировать. Процесс также может заблокировать сигнал, отложив на некоторое время его обработку. Это возможно не для всех сигналов. Например, для сигналов SIGKILL и SIGSTOP единственным действием является действие по умолчанию, эти сигналы нельзя ни перехватить, ни заблокировать, ни игнорировать. Для ряда сигналов, преимущественно связанных с аппаратными ошибками и особыми ситуациями, обработка, отличная от умалчивающей, не рекомендуется, так как может привести к непредсказуемым (для процесса) результатам.

Следует заметить, что любая обработка сигнала, в том числе обработка по умолчанию, подразумевает, что процесс выполняется. На системах с высокой загрузкой это может привести к существенным задержкам между отправлением и доставкой сигнала, т. к. процесс не получит сигнал, пока не будет выбран планировщиком, и ему не будут предоставлены вычислительные ресурсы. Этот вопрос был затронут при разговоре о точности таймеров, которые может использовать процесс.

Доставка сигнала происходит после того, как ядро от имени процесса вызывает системную процедуру `issig()`, которая проверяет, существуют ли ожидающие доставки сигналы, адресованные данному процессу. Функция `issig()` вызывается ядром в трех случаях:

1. Непосредственно перед возвращением из режима ядра в режим задачи после обработки системного вызова или прерывания.
2. Непосредственно перед переходом процесса в состояние сна с приоритетом, допускающим прерывание сигналом.
3. Сразу же после пробуждения после сна с приоритетом, допускающим прерывание сигналом.

Если процедура `issig()` обнаруживает ожидающие доставки сигналы, ядро вызывает функцию доставки сигнала, которая выполняет действия по умолчанию или вызывает специальную функцию `sendsig()`, запускающую обработчик сигнала, зарегистрированный процессом. Функция `sendsig()` возвращает процесс в режим задачи, передает управление обработчику сигнала, а затем восстанавливает контекст процесса для продолжения прерванного сигналом выполнения.

Рассмотрим типичные ситуации, связанные с отправлением и доставкой сигналов. Допустим, пользователь, работая за терминалом, нажимает клавишу прерывания (`<Del>` или `<Ctrl>+<C>` для большинства систем). Нажатие любой клавиши вызывает аппаратное прерывание (например, прерывание от последовательного порта), а драйвер терминала при обработке этого прерывания определяет, что была нажата специальная клавиша, генерирующая сигнал, и отправляет текущему процессу, связанному с терминалом, сигнал SIGINT. Когда процесс будет выбран планировщиком и

запущен на выполнение, при переходе в режим задачи он обнаружит поступление сигнала и обработает его. Если же в момент генерации сигнала терминальным драйвером процесс, которому был адресован сигнал, уже выполнялся (т. е. был прерван обработчиком терминального прерывания), он также обработает сигнал при возврате в режим задачи после обработки прерывания.

Работа с сигналами, связанными с особыми ситуациями, незначительно отличается от вышеописанной. Особая ситуация возникает при выполнении процессом определенной инструкции, вызывающей в системе ошибку (например, деление на ноль, обращение к недопустимой области памяти, недопустимая инструкция или вызов и т. д.). Если такое происходит, вызывается системный обработчик особой ситуации, и процесс переходит в режим ядра, почти так же, как и при обработке любого другого прерывания. Обработчик отправляет процессу соответствующий сигнал, который доставляется, когда выполнение возвращается в режим задачи.

При обсуждении состояния сна процесса мы выделили две категории событий, вызывающих состояние сна процесса: допускающие прерывание сигналом и не допускающие такого прерывания. В последнем случае сигнал будет терпеливо ожидать нормального пробуждения процесса, например, после завершения операции дискового ввода/вывода.

В первом случае, доставка сигнала будет проверена ядром непосредственно перед переходом процесса в состояние сна. Если такой сигнал поступил, будет вызван обработчик сигнала, а системный вызов, который выполнялся процессом, будет аварийно завершен с ошибкой EINTR. Если генерация сигнала произошла в течение сна процесса, ядро будет вынуждено разбудить его и снять прерванный системный вызов (ошибка EINTR). После пробуждения процесса либо вследствие получения сигнала, либо из-за наступления ожидаемого события, ядром будет вызвана функция `issig()`, которая обнаружит поступление сигнала и вызовет соответствующую обработку<sup>13</sup>.

## **Взаимодействие между процессами**

Как уже обсуждалось, в UNIX процессы выполняются в собственном адресном пространстве и по существу изолированы друг от друга. Тем самым сведены к минимуму возможности влияния процессов друг на друга, что является необходимым в многозадачных операционных системах. Однако

<sup>13</sup> В BSD UNIX были введено понятие *перезапускаемых системных вызовов*. Суть этого механизма заключается в том, что прерванный сигналом системный вызов автоматически повторяется после обработки сигнала, **вместо** аварийного завершения с ошибкой EINTR. Допускается отключение этой возможности для конкретных сигналов.

от одиночного изолированного процесса мало пользы. Сама концепция UNIX заключается в модульности, т. е. основана на взаимодействии между отдельными процессами.

Для реализации взаимодействия требуется:

- обеспечить средства взаимодействия между процессами и одновременно
- П исключить нежелательное влияние одного процесса на другой.

Взаимодействие между процессами необходимо для решения следующих задач:

- Передача данных.* Один процесс передает данные другому процессу, при этом их объем может варьироваться от десятков байтов до нескольких мегабайтов.
- П *Совместное использование данных.* Вместо копирования информации от одного процесса к другому, процессы могут совместно использовать одну копию данных, причем изменения, сделанные одним процессом, будут сразу же заметны для другого. Количество взаимодействующих процессов может быть больше двух. При совместном использовании ресурсов процессам может понадобиться некоторый протокол взаимодействия для сохранения целостности данных и исключения конфликтов при доступе к ним.
- О *Извещения.* Процесс может известить другой процесс или группу процессов о наступлении некоторого события. Это может понадобиться, например, для синхронизации выполнения нескольких процессов.

Очевидно, что решать данную задачу средствами самих процессов неэффективно, а в рамках многозадачной системы — опасно и потому невозможно. Таким образом, сама операционная система должна обеспечить механизмы межпроцессного взаимодействия (Inter-Process Communication, IPC).

К средствам межпроцессного взаимодействия, присутствующим во всех версиях UNIX, можно отнести:

- П сигналы
- П каналы
- П FIFO (именованные каналы)
- П сообщения (очереди сообщений)
- П семафоры
- П разделяемую память

Последние три типа IPC обычно обобщенно называют *System V IPC*.

Во многих версиях UNIX есть еще одно средство IPC — сокеты, впервые предложенные в BSD UNIX (им посвящен отдельный раздел главы).

Сигналы изначально были предложены как средство уведомления об ошибках, но могут использоваться и для элементарного IPC, например, для синхронизации процессов или для передачи простейших команд от одного процесса к другому<sup>14</sup>. Однако использование сигналов в качестве средства IPC ограничено из-за того, что сигналы очень ресурсоемки. Отправка сигнала требует выполнения системного вызова, а его доставка — прерывания процесса-получателя и интенсивных операций со стеком процесса для вызова функции обработки и продолжения его нормального выполнения. При этом сигналы слабо информативны и их число весьма ограничено. Поэтому сразу переходим к следующему механизму — каналам.

## Каналы

Вспомните синтаксис организации программных каналов при работе в командной строке shell:

```
cat myfile | wc
```

При этом (стандартный) вывод программы *cat(1)*, которая выводит содержимое файла **myfile**, передается на (стандартный) ввод программы *wc(1)*, которая, в свою очередь подсчитывает количество строк, слов и символов. В результате мы получим что-то вроде:

```
12 45 260
```

что будет означать количество строк, слов и символов в файле **myfile**.

Таким образом, два процесса обмениались данными. При этом использовался программный канал, обеспечивающий *однонаправленную передачу данных* между двумя задачами.

Для создания канала используется системный вызов *pipe(2)*:

```
int pipe(int *filedes);
```

который возвращает два файловых дескриптора — *filedes[0]* для записи в канал и *filedes[1]* для чтения из канала. Теперь, если один процесс записывает данные в *filedes[0]*, другой сможет получить эти данные из *filedes[1]*. Вопрос только в том, как другой процесс сможет получить сам файловый дескриптор *filedes[1]*?

Вспомним наследуемые атрибуты при создании процесса. Дочерний процесс наследует и разделяет все назначенные файловые дескрипторы родительского. То есть доступ к дескрипторам *filedes* канала может получить сам процесс, вызвавший *pipe(2)*, и его дочерние процессы. В этом заклю-

<sup>14</sup> Например, для сервера системы имен (DNS) *named (1M)* таким образом используется сигнал *SIGHUP*, по существу являющийся командой обновления базы данных.

чается серьезный недостаток каналов, поскольку они могут быть использованы для передачи данных только между родственными процессами. Каналы не могут использоваться в качестве средства межпроцессного взаимодействия между независимыми процессами.

Хотя в приведенном примере может показаться, что процессы *cat(1)* и *wc(1)* независимы, на самом деле оба этих процесса создаются процессом *shell* и являются родственными.

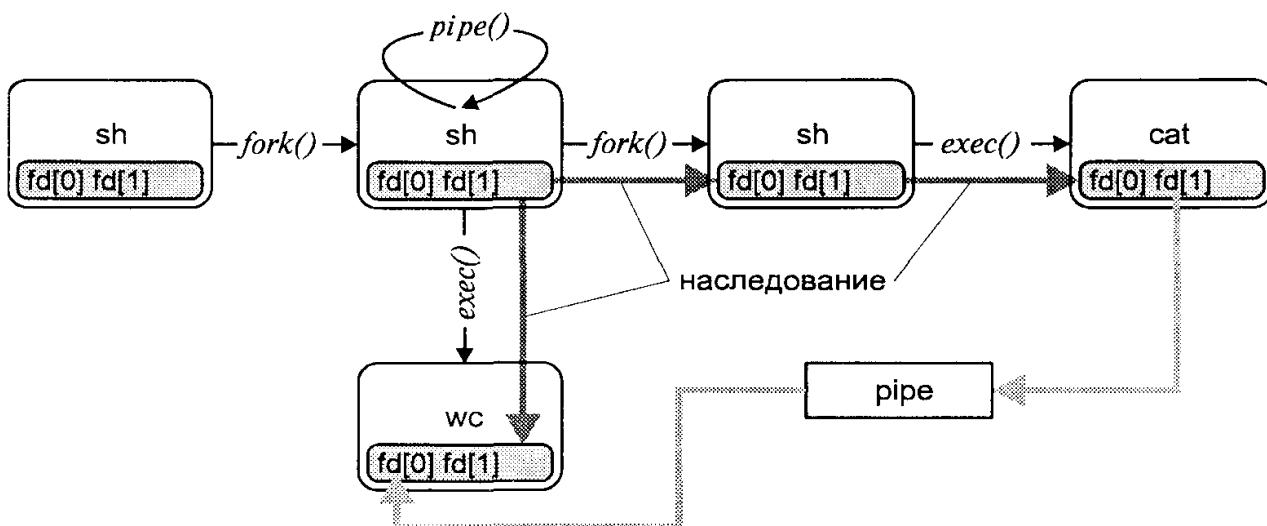


Рис. 3.17. Создание канала между задачами *cat(1)* и *wc(1)*

## FIFO

Название каналов FIFO происходит от выражения First In First Out (первый вошел — первый вышел). FIFO очень похожи на каналы, поскольку являются односторонним средством передачи данных, причем чтение данных происходит в порядке их записи. Однако в отличие от программных каналов, FIFO имеют имена, которые позволяют независимым процессам получить к этим объектам доступ. Поэтому иногда FIFO также называют *именованными каналами*. FIFO являются средством UNIX System V и не используются в BSD. Впервые FIFO были представлены в System III, однако они до сих пор не документированы и поэтому мало используются.

FIFO является отдельным типом файла в файловой системе UNIX (*ls -l* покажет символ *p* в первой позиции, см. раздел "Файлы и файловая система UNIX" главы 1). Для создания FIFO используется системный вызов *mknod(2)*:

```
int mknod(char *pathname, int mode, int dev);
```

где *pathname* — имя файла в файловой системе (имя FIFO),  
*mode* — флаги владения, прав доступа и т. д. (см. поле *mode* файла),  
*dev* — при создании FIFO игнорируется.

FIFO может быть создан и из командной строки shell:

```
$ mknod name p
```

После создания FIFO может быть открыт на запись и чтение, причем запись и чтение могут происходить в разных независимых процессах.

Каналы FIFO и обычные каналы работают по следующим правилам:

1. При чтении меньшего числа байтов, чем находится в канале или FIFO, возвращается требуемое число байтов, остаток сохраняется для последующих чтений.
2. При чтении большего числа байтов, чем находится в канале или FIFO, возвращается доступное число байтов. Процесс, читающий из канала, должен соответствующим образом обработать ситуацию, когда прочитано меньше, чем заказано.
3. Если канал пуст и ни один процесс не открыл его на запись, при чтении из канала будет получено 0 байтов. Если один или более процессов открыли канал для записи, вызов *read(2)* будет заблокирован до появления данных (если для канала или FIFO не установлен флаг отсутствия блокирования *O\_NDELAY*).
4. Запись числа байтов, меньшего емкости канала или FIFO, гарантированно атомарно. Это означает, что в случае, когда несколько процессов одновременно записывают в канал, порции данных от этих процессов не перемешиваются.
5. При записи большего числа байтов, чем это позволяет канал или FIFO, вызов *write(2)* блокируется до освобождения требуемого места. При этом атомарность операции не гарантируется. Если процесс пытается записать данные в канал, не открытый ни одним процессом на чтение, процессу генерируется сигнал *SIGPIPE*, а вызов *write(2)* возвращает 0 с установкой ошибки (*errno=EPPIPE*) (если процесс не установил обработки сигнала *SIGPIPE*, производится обработка по умолчанию — процесс завершается).

В качестве примера приведем простейший пример приложения клиент-сервер, использующего FIFO для обмена данными. Следуя традиции, клиент посыпает серверу сообщение "Здравствуй, Мир!", а сервер выводит это сообщение на терминал.

#### **Сервер:**

```
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO      "fifo.1"
#define MAXBUFF  80
main()
{
    int readfd, n;
```

```

    char buff[MAXBUFF]; /*буфер для чтения данных из FIFO*/
/*Создадим специальный файл FIFO с открытыми для всех правами
доступа на чтение и запись*/
    if ( mknod(FIFO, S_IFIFO | 0666, 0)<0){
        printf("Невозможно создать FIFO\n"); exit(1); }
/*Получим доступ к FIFO*/
    if ( ( readfd = open(FIFO, O_RDONLY)) < 0){
        printf("Невозможно открыть FIFO\n"); exit(1); }
/*Прочитаем сообщение ("Здравствуй, Мир!") и выведем его на
экран*/
    while ( (n = read(readfd, buff, MAXBUFF)) > 0)
        if (write(1, buff, n) != n) {
            printf ("Ошибка вывода\n"); exit(1); }
/*Закроем FIFO, удаление FIFO – дело клиента*/
    close (readfd);

    exit (0);

```

### Клиент:

```

#include <sys/types.h>
#include <sys/stat.h>
/*Соглашение об имени FIFO*/
#define FIFO      "fifo.1"
main()
{
    int writefd, n;
/*Получим доступ к FIFO*/
    if ( (writefd = open(FIFO, O_WRONLY)) < 0){
        printf("Невозможно открыть FIFO\n"); exit(1); }
/*Передадим сообщение серверу FIFO*/
    if (write(writefd, "Здравствуй, Мир!\n", 18) != 18)
        printf("Ошибка записи\n"); exit(1);

/*Закроем FIFO*/
    close(writefd);

/*Удалим FIFO*/
    if (unlink(FIFO) < 0){
        printf("Невозможно удалить FIFO\n"); exit(1); }

    exit (0);

```

## Идентификаторы и имена в IPC

Как было показано, отсутствие имен у каналов делает их недоступными для независимых процессов. Этот недостаток устранен у FIFO, которые имеют имена. Другие средства межпроцессного взаимодействия, являющиеся более сложными, требуют дополнительных соглашений по именам

и идентификаторам. Множество возможных имен объектов конкретного типа межпроцессного взаимодействия называется *пространством имен* (name space). Имена являются важным компонентом системы межпроцессного взаимодействия для всех объектов, кроме каналов, поскольку позволяют различным процессам получить доступ к общему объекту. Так, именем FIFO является имя файла именованного канала. Используя условленное имя созданного FIFO два процесса могут обращаться к этому объекту для обмена данными.

Для таких объектов IPC, как очереди сообщений, семафоры и разделяемая память, процесс назначения имени является более сложным, чем просто указание имени файла. Имя для этих объектов называется *ключом* (key) и генерируется функцией *ftok(3C)* из двух компонентов — имени файла и идентификатора проекта:

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok (char *filename, char proj) ;
```

В качестве filename можно использовать имя некоторого файла, известное взаимодействующим процессам. Например, это может быть имя программы-сервера. Важно, чтобы этот файл существовал на момент создания ключа. Также нежелательно использовать имя файла, который создается и удаляется в процессе работы распределенного приложения, поскольку при генерации ключа используется номер inode файла. Вновь созданный файл может иметь другой inode и впоследствии процесс, желающий иметь доступ к объекту, получит неверный ключ.

Пространство имен позволяет создавать и совместно использовать IPC неродственным процессам. Однако для ссылок на уже созданные объекты используются идентификаторы, точно так же, как файловый дескриптор используется для работы с файлом, открытым по имени.

Каждое из перечисленных IPC имеет свой уникальный дескриптор (идентификатор), используемый ОС (ядром) для работы с объектом. Уникальность дескриптора обеспечивается уникальностью дескриптора для каждого из типов объектов (очереди сообщений, семафоры и разделяемая память), т. е. какая-либо очередь сообщений может иметь тот же числовой идентификатор, что и разделяемая область памяти (хотя любые две очереди сообщений должны иметь различные идентификаторы).

**Таблица 3.5.** Идентификация объектов IPC

Объект IPC	Пространство имен	Дескриптор
Канал	—	Файловый дескриптор
FIFO	Имя файла	Файловый дескриптор
Очередь сообщений	Ключ	Идентификатор

Таблица 3.5 (продолжение)

Объект IPC	Пространство имен	Дескриптор
Семафор	Ключ	Идентификатор
Разделяемая память	Ключ	Идентификатор

Работа с объектами IPC System V во многом сходна. Для создания или получения доступа к объекту используются соответствующие системные вызовы *get*: *msgget(2)* для очереди сообщений, *semget(2)* для семафора и *shmat(2)* для разделяемой памяти. Все эти вызовы возвращают дескриптор объекта в случае успеха и -1 в случае неудачи. Отметим, что функции *get* позволяют процессу получить ссылку на объект, которой по существу является возвращаемый дескриптор, но не позволяют производить конкретные операции над объектом (помещать или получать сообщения из очереди сообщений, устанавливать семафор или записывать данные в разделяемую память. Все функции *get* в качестве аргументов используют ключ key и флагги создания объекта *ipcflag*. Остальные аргументы зависят от конкретного типа объекта. Переменная *ipcflag* определяет права доступа к объекту *PERM*, а также указывает, создается ли новый объект или требуется доступ к существующему. Последнее определяется комбинацией (или отсутствием) флагков *IPC\_CREAT* и *IPC\_EXCL*.

Права доступа к объекту указываются набором флагков доступа, подобно тому, как это делается для файлов:

Значение <i>PERM</i> (в восьмеричном виде)	Аналог прав доступа для файлов	Разрешено
0400	r -----	Чтение для владельца-пользователя
0200	-w-----	Запись для владельца-пользователя
0040	---r----	Чтение для владельца-группы
0020	----w---	Запись для владельца-группы
0004	-----r--	Чтение для всех остальных
0002	-----w-	Запись для всех остальных

Комбинацией флагков можно добиться различных результатов:

Значение аргумента <i>ipcflag</i>	Результат действия функции	
	Объект существует	Объект не существует
PERM IPC_CREAT	Возвращает дескриптор	Ошибка: отсутствие объекта (ENOENT)
PERM IPC_CREAT !IPC_EXCL	Возвращает дескриптор	Создает объект с соответствующими PERM правами доступа
	Ошибка: объект уже существует (EEXIST)	Создает объект с соответствующими PERM правами доступа

Работа с объектами IPC System V во многом похожа на работу с файлами в UNIX. Одним из различий является то, что файловые дескрипторы имеют значимость в контексте процесса, в то время как значимость дескрипторов объектов IPC распространяется на всю систему. Так файловый дескриптор 3 одного процесса в общем случае никак не связан с дескриптором 3 другого неродственного процесса (т. е. эти дескрипторы ссылаются на различные файлы). Иначе обстоит дело с дескрипторами объектов IPC. Все процессы, использующие, скажем, одну очередь сообщений, получат одинаковые дескрипторы этого объекта.

Для каждого из объектов IPC ядро поддерживает соответствующую структуру данных, отличную для каждого типа объекта IPC (очереди сообщений, семафора или разделяемой памяти). Общей у этих данных является структура `ipc_perm`, описывающая права доступа к объекту, подобно тому, как это делается для файлов. Основными полями этой структуры являются:

<code>uid</code>	Идентификатор владельца-пользователя объекта
<code>gid</code>	Идентификатор владельца-группы объекта
<code>cuid</code>	UID создателя объекта
<code>cgid</code>	GID создателя объекта
<code>mode</code>	Права доступа на чтение и запись для всех классов доступа (9 битов)
<code>key</code>	Ключ объекта

Права доступа (как и для файлов) определяют возможные операции, которые может выполнять над объектом конкретный процесс (получение доступа к существующему объекту, чтение, запись и удаление).

Заметим, что система не удаляет созданные объекты IPC даже тогда, когда ни один процесс не пользуется ими. Удаление созданных объектов является обязанностью процессов, которым для этого предоставляются соответствующие функции управления `msgctl(2)`, `semctl(2)`, `shmtctl(2)`. С помощью этих функций процесс может получить и установить ряд полей внутренних структур, поддерживаемых системой для объектов IPC, а также удалить созданные объекты. Безусловно, как и во многих других случаях использования объектов IPC процессы предварительно должны "договориться", какой процесс и когда удалит объект. Чаще всего, таким процессом является сервер.

## Сообщения

Как уже обсуждалось, очереди сообщений являются составной частью UNIX System V, они обслуживаются операционной системой, размещаются в адресном пространстве ядра и являются разделяемым системным ресурсом. Каждая очередь сообщений имеет свой уникальный идентификатор. Процессы могут записывать и считывать сообщения из различных очередей. Процесс, пославший сообщение в очередь, может не ожидать

чтения этого сообщения каким-либо другим процессом. Он может закончить свое выполнение, оставив в очереди сообщение, которое будет прочитано другим процессом позже.

Данная возможность позволяет процессам обмениваться структуризованными данными, имеющими следующие атрибуты:

- Тип сообщения (позволяет мультиплексировать сообщения в одной очереди)
- Длина данных сообщения в байтах (может быть нулевой)
- Собственно данные (если длина ненулевая, могут быть структуризованными)

Очередь сообщений хранится в виде внутреннего одностороннего связанных списка в адресном пространстве ядра. Для каждой очереди ядро создает заголовок очереди (`msqid_ds`), где содержится информация о правах доступа к очереди (`msg_perm`), ее текущем состоянии (`msg_cbytes` — число байтов и `msg_qnum` — число сообщений в очереди), а также указатели на первое (`msg_first`) и последнее (`msg_last`) сообщения, хранящиеся в виде связанного списка (рис. 3.18). Каждый элемент этого списка является отдельным сообщением.

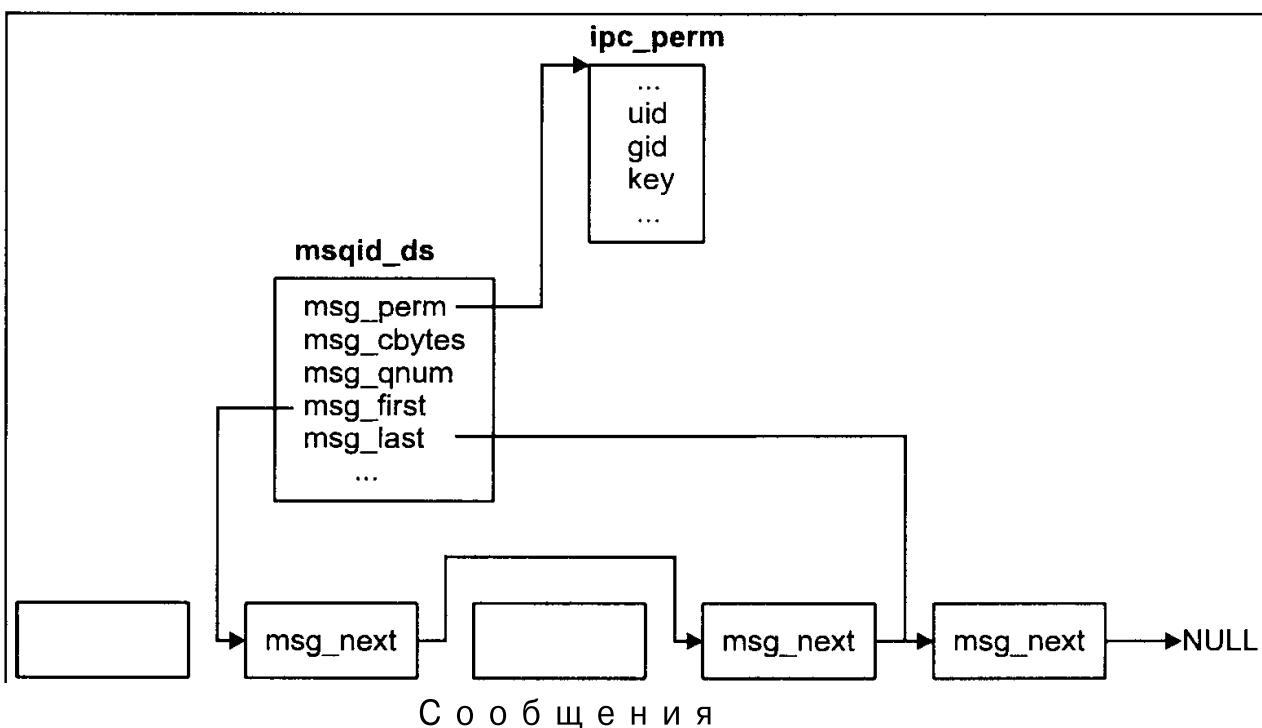


Рис. 3.18. Структура очереди сообщений

Для создания новой очереди сообщений или для доступа к существующей используется системный вызов `msgget(2)`:

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/msg.h>
int msgget( key_t key, int msgflg);
```

Функция возвращает дескриптор объекта-очереди, либо -1 в случае ошибки. Подобно файловому дескриптору, этот идентификатор используется процессом для работы с очередью сообщений. В частности, процесс может:

- Помещать в очередь сообщения с помощью функции *msgsnd(2)*;
- Получать сообщения определенного типа из очереди с помощью функции *msgrecv(2)*;
- Управлять сообщениями с помощью функции *msgctl(2)*.

Перечисленные системные вызовы манипулирования сообщениями имеют следующий вид:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp,
           size_t msgsz, int msgflg);

int msgrecv(int msqid, void *msgp,
            size_t msgsz, long msgtyp, int msgflg);
```

Здесь *msqid* является дескриптором объекта, полученного в результате вызова *msgget(2)*. Параметр *msgp* указывает на буфер, содержащий тип сообщения и его данные, размер которого равен *msgsz* байт. Буфер имеет следующие поля:

long msgtype	тип сообщения
char msgtext[]	данные сообщения

Аргумент *msgtyp* указывает на тип сообщения и используется для их выборочного получения. Если *msgtyp* равен 0, функция *msgrecv(2)* получит первое сообщение из очереди. Если величина *msgtyp* выше 0, будет получено первое сообщение указанного типа. Если *msgtyp* меньше 0, функция *msgrecv(2)* получит сообщение с минимальным значением типа, меньше или равного абсолютному значению *msgtyp*.

Очереди сообщений обладают весьма полезным свойством — в одной очереди можно мультиплексировать сообщения от различных процессов. Для демультиплексирования используется атрибут *msgtype*, на основании которого любой процесс может фильтровать сообщения с помощью функции *msgrecv(2)*, как это было показано выше.

Рассмотрим типичную ситуацию взаимодействия процессов, когда серверный процесс обменивается данными с несколькими клиентами. Свойство мультиплексирования позволяет использовать для такого обмена одну очередь сообщений. Для этого сообщениям, направляемым от любого из клиентов серверу, будем присваивать значение типа, скажем, равным 1. Если

в теле сообщения клиент каким-либо образом идентифицирует себя (например, передает свой PID), то сервер сможет передать сообщение конкретному клиенту, присваивая тип сообщения равным этому идентификатору.

Поскольку функция `msgrecv(2)` позволяет принимать сообщения определенного типа (типов), сервер будет принимать сообщения с типом 1, а клиенты — сообщения с типами, равными идентификаторам их процессов. Схема такого взаимодействия представлена на рис. 3.19.

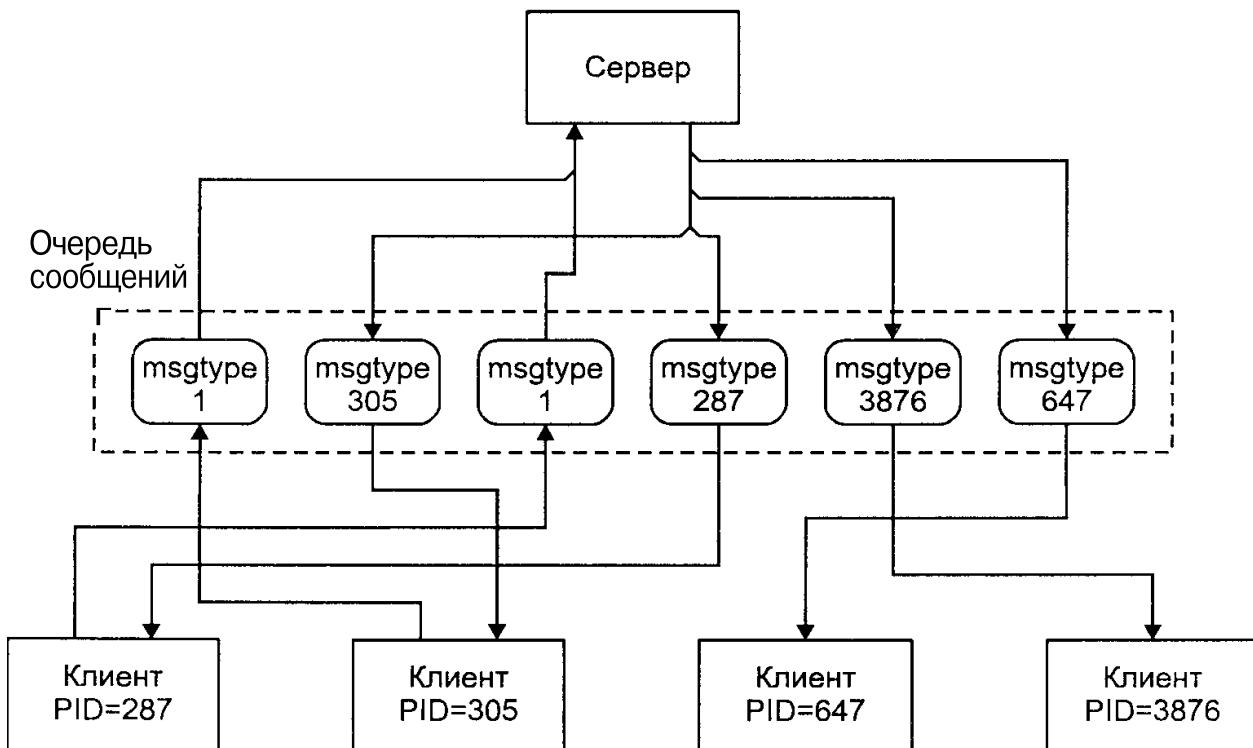


Рис. 3.19. Мультиплексирование сообщений в одной очереди

Атрибут `msgtype` также можно использовать для изменения порядка извлечения сообщений из очереди. Стандартный порядок получения сообщений аналогичен принципу FIFO — сообщения получаются в порядке их записи. Однако используя тип, например, для назначения приоритета сообщений, этот порядок легко изменить.

Пример приложения "Здравствуй, Мир!", использующего сообщения:

#### Файл описания `mesg.h`

```
#define MAXBUFF 80
#define PERM 0666
/* Определим структуру нашего сообщения. Она может отличаться от
структуры msgbuf, но должна содержать поле mtype. В данном случае
структура сообщения состоит из буфера обмена */
typedef struct our msgbuf {
    long mtype;
```

```
    char    buff[MAXBUFF];
} Message;
```

**Сервер:**

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include "msg.h"
main()
{
    /*Структура нашего сообщения (может отличаться от структуры msgbuf)*/
    Message    message;
    key_t      key;
    int msgid, length, n;

    /*Получим ключ*/
    if ( (key = ftok("server", 'A')) < 0) {
        printf("Невозможно получить ключ\n"); exit(1); }

    /*Тип принимаемых сообщений*/
    message.mtype=1L;
    /*Создадим очередь сообщений*/
    if ( (msgid = msgget(key, PERM | IPC_CREAT)) < 0) {
        printf("Невозможно создать очередь\n"); exit(1); }

    /*Прочитаем сообщение*/
    n = msgrcv(msgid, &message, sizeof(message), message.mtype, 0);
    /*Если сообщение поступило, выведем его содержимое на терминал*/
    if (n > 0) {
        if (write(1, message buff, n) != n) {
            printf("Ошибка вывода\n"); exit(1); }
    }
    else { printf("Ошибка чтения сообщения\n"); exit(1); }

    /*Удалить очередь поручим клиенту*/
    exit (0);
}
```

**Клиент:**

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include "msg.h"
main()
{
    /*Структура нашего сообщения (может отличаться от структуры msgbuf)*/
    Message    message;
    key_t      key;
    int msgid, length;
    /*Тип посылаемого сообщения, может использоваться для
    мультиплексирования*/
    message.mtype = 1L;
```

```
/*Получим ключ*/
if ( (key = ftok("server", 'A')) < 0) {
    printf("Невозможно получить ключ\n"); exit(1); }
/*Получим доступ к очереди сообщений, очередь уже должна быть создана
сервером*/
if ( (msgid= msgget(key, 0)) < 0) {
    printf("Невозможно получить доступ к очереди\n"); exit(1);
}
/*Поместим строку в сообщение*/
if ( (length = sprintf(message.buf,
    "Здравствуй, Мир!\n")) < 0) {
    printf("Ошибка копирования в буфер\n"); exit(1); }
/*Передадим сообщение*/
if (msgsnd(msgid, (void *) &message, length, 0) !=0) {
    printf("Ошибка записи сообщения в очередь\n");
    exit(1); }
/*Удалим очередь сообщений*/
if (msgctl(msgid, IPC_RMID, 0) < 0) {
    printf("Ошибка удаления очереди\n"); exit(1); }
exit(0);
}
```

## Семафоры

Для синхронизации процессов, а точнее, для синхронизации доступа нескольких процессов к разделяемым ресурсам, используются *семафоры*. Являясь одной из форм IPC, семафоры не предназначены для обмена большими объемами данных, как в случае FIFO или очередей сообщений. Вместо этого, они выполняют функцию, полностью соответствующую своему названию — разрешать или запрещать процессу использование того или иного разделяемого ресурса.

Применение семафоров поясним на простом примере. Допустим, имеется некий разделяемый ресурс (например, файл). Необходимо блокировать доступ к ресурсу для других процессов, когда некий процесс производит операцию над ресурсом (например, записывает в файл). Для этого свяжем с данным ресурсом некую целочисленную величину — счетчик, доступный для всех процессов. Примем, что значение 1 счетчика означает доступность ресурса, 0 — его недоступность. Тогда перед началом работы с ресурсом процесс должен проверить значение счетчика. Если оно равно 0 — ресурс занят и операция недопустима — процессу остается ждать. Если значение счетчика равно 1 — можно работать с ресурсом. Для этого, прежде всего, необходимо заблокировать ресурс, т. е. изменить значение счетчика на 0. После выполнения операции для освобождения ресурса значение счетчика необходимо изменить на 1. В приведенном примере счетчик играет роль семафора.

Для нормальной работы необходимо обеспечить выполнение следующих условий:

1. Значение семафора должно быть доступно различным процессам. Поэтому семафор находится не в адресном пространстве процесса, а в адресном пространстве ядра.
2. Операция проверки и изменения значения семафора должна быть реализована в виде одной атомарной по отношению к другим процессам (т. е. непрерываемой другими процессами) операции. В противном случае возможна ситуация, когда после проверки значения семафора выполнение процесса будет прервано другим процессом, который в свою очередь проверит семафор и изменит его значение. Единственным способом гарантировать атомарность критических участков операций является выполнение этих операций в режиме ядра (см. режимы выполнения процесса).

Таким образом семафоры являются системным ресурсом, действия над которым производятся через интерфейс системных вызовов.

Семафоры в System V обладают следующими характеристиками:

- Семафор представляет собой не один счетчик, а группу, состоящую из нескольких счетчиков, объединенных общими признаками (например, дескриптором объекта, правами доступа и т. д.).
- Каждое из этих чисел может принимать любое неотрицательное значение в пределах, определенных системой (а не только значения 0 и 1).

Для каждой группы семафоров (в дальнейшем мы будем называть группу просто семафором) ядро поддерживает структуру данных `semid_ds`, включающую следующие поля:

<code>struct ipc_perm sem_perm</code>	Описание прав доступа
<code>struct sem *sem_base</code>	Указатель на первый элемент массива семафоров
<code>ushort sem_nsems</code>	Число семафоров в группе
<code>time_t sem_otime</code>	Время последней операции
<code>time_t sem_ctime</code>	Время последнего изменения

Значение конкретного семафора из набора хранится во внутренней структуре `sem`:

<code>ushort semval</code>	Значение семафора
<code>pid_t sempid</code>	Идентификатор процесса, выполнившего последнюю операцию над семафором
<code>ushort semncnt</code>	Число процессов, ожидающих увеличения значения семафора
<code>ushort semzcnt</code>	Число процессов, ожидающих обнуления семафора

Помимо собственно значения семафора, в структуре `sem` хранится идентификатор процесса, вызвавшего последнюю операцию над семафором, число процессов, ожидающих увеличения значения семафора, и число

процессов, ожидающих, когда значение семафора станет равным нулю. Эта информация позволяет ядру производить операции над семафорами, которые мы обсудим несколько позже.

Для получения доступа к семафору (и для его создания, если он не существует) используется системный вызов *semget(2)*:

```
ttinclude <sys/types.h>
#include <sys/ipc.h>
ttinclude <sys/sem.h>
int semget (key_t key, int nsems, int semflag);
```

В случае успешного завершения операции функция возвращает дескриптор объекта, в случае неудачи——1. Аргумент *nsems* задает число семафоров в группе. В случае, когда мы не создаем, а лишь получаем доступ к существующему семафору, этот аргумент игнорируется. Аргумент *semflag* определяет права доступа к семафору и флагги для его создания (IPC\_CREAT, IPC\_EXCL).

После получения дескриптора объекта процесс может производить операции над семафором, подобно тому, как после получения файлового дескриптора процесс может читать и записывать данные в файл. Для этого используется системный вызов *semop(2)*:

```
ttinclude <sys/types.h>
ttinclude <sys/ipc.h>
ttinclude <sys/sem.h>
int semop(int semid, struct sembuf *semop, size_t nops);
```

В качестве второго аргумента функции передается указатель на структуру данных, определяющую операции, которые требуется произвести над семафором с дескриптором *semid*. Операций может быть несколько, и их число указывается в последнем аргументе *nops*. Важно, что ядро обеспечивает атомарность выполнения критических участков операций (например, проверка значения — изменение значения) по отношению к другим процессам.

Каждый элемент набора операций *semop* имеет вид:

```
struct sembuf {
    short sem_num;      /*номер семафора в группе*/
    short sem_op;        /*операция*/
    short sem_flg;       /*флаги операции*/
};
```

UNIX допускает три возможные операции над семафором, определяемые полем *semop*:

1. Если величина *semop* положительна, то текущее значение семафора увеличивается на эту величину.

2. Если значение `semop` равно нулю, процесс ожидает, пока семафор не обнулится.
3. Если величина `semop` отрицательна, процесс ожидает, пока значение семафора не станет большим или равным абсолютной величине `semop`. Затем абсолютная величина `semop` вычитается из значения семафора.

Можно заметить, что первая операция изменяет значение семафора (безусловное выполнение), вторая операция только проверяет его значение (условное выполнение), а третья — проверяет, а затем изменяет значение семафора (условное выполнение).

При работе с семафорами взаимодействующие процессы должны договориться об их использовании и кооперативно проводить операции над семафорами. Операционная система не накладывает ограничений на использование семафоров. В частности, процессы вольны решать, какое значение семафора является разрешающим, на какую величину изменяется значение семафора и т. п.

Таким образом, при работе с семафорами процессы используют различные комбинации из трех операций, определенных системой, по-своему трактуя значения семафоров.

В качестве примера рассмотрим два случая использования бинарного семафора (т. е. значения которого могут принимать только 0 и 1). В первом примере значение 0 является разрешающим, а 1 запирает некоторый разделяемый ресурс (файл, разделяемая память и т. п.), ассоциированный с семафором. Определим операции, запирающие ресурс и освобождающие его:

```
static struct sembuf sop_lock[2] = {
    0, 0, 0,      /*ожидать обнуления семафора*/
    0, 1, 0      /*затем увеличить значение семафора на 1*/
};

static struct sembuf sop_unlock [1] = {
    0, -1, 0      /*обнулить значение семафора*/
};
```

Итак, для запирания ресурса процесс производит вызов:

```
semop(semid, &sop_lock[0], 2);
```

обеспечивающий атомарное выполнение двух операций<sup>15</sup>:

<sup>15</sup> Ядро обеспечивает атомарное выполнение не всего набора операций в целом, а лишь критических участков. Так, например, в процессе ожидания освобождения ресурса (ожидание нулевого значения семафора) выполнение процесса будет (и должно быть) прервано процессом, который освободит ресурс (т. е. установит значение семафора равным 1). Ожидание семафора соответствует состоянию "сна" процесса, допускающим выполнение других процессов в системе. В противном случае, процесс, ожидающий ресурс, остался бы заблокированным навсегда.

1. Ожидание доступности ресурса. В случае, если ресурс уже занят (значение семафора равно 1), выполнение процесса будет приостановлено до освобождения ресурса (значение семафора равно 0).
2. Запирание ресурса. Значение семафора устанавливается равным 1.

Для освобождения ресурса процесс должен произвести вызов:

```
semop(semid, &sop_unlock[0], 1);
```

который уменьшит текущее значение семафора (равное 1) на 1, и оно станет равным 0, что соответствует освобождению ресурса. Если какой-либо из процессов ожидает ресурса (т. е. произвел вызов операции `sop_lock`), он будет "разбужен" системой, и сможет в свою очередь запереть ресурс и работать с ним.

Во втором примере изменим трактовку значений семафора: значению 1 семафора соответствует доступность некоторого ассоциированного с семафором ресурса, а нулевому значению — его недоступность. В этом случае содержание операций несколько изменится.

```
static struct sembuf sop_lock[2] = {
    0, -1, 0,      /*ожидать разрешающего сигнала (1),
                     затем обнулить семафор*/
};

static struct sembuf sop_unlock [1] = {
    0, 1, 0        /*увеличить значение семафора на 1*/
};
```

Процесс запирает ресурс вызовом:

```
semop(semid, &sop_lock[0], 1);
```

**а освобождает:**

```
semop(semid, &sop_unlock[0], 1);
```

Во втором случае операции получились проще (по крайней мере их код стал компактнее), однако этот подход имеет потенциальную опасность: при создании семафора, его значения устанавливаются равными 0, и во втором случае он сразу же запирает ресурс. Для преодоления данной ситуации процесс, первым создавший семафор, должен вызвать операцию `sop_unlock`, однако в этом случае процесс инициализации семафора перестанет быть атомарным и может быть прерван другим процессом, который, в свою очередь, изменит значение семафора. В итоге, значение семафора станет равным 2, что повредит нормальной работе с разделяемым ресурсом.

Можно предложить следующее решение данной проблемы:

```
/*Создаем семафор, если он уже существует semget возвращает
ошибку, поскольку указан флаг IPC_EXCL*/
if ( (semid = semget( key, nsems, perms  IPC_CREAT | IPC_EXCL ) )
    < 0)
```

```

if (errno == EEXIST)
{
    /*Действительно, ошибка вызвана существованием объекта*/
    if ((semid = semget(key, nsems, perms)) < 0)
        return(-1); /*Возможно, не хватает системных ресурсов*/
    }
    else return(-1); /*Возможно, не хватает системных ресурсов*/
}
/*Если семафор создан нами, проинициализируем его*/
else semop(semid, &sop_unlock[0], 1);

```

## Разделяемая память

Интенсивный обмен данными между процессами с использованием рассмотренных механизмов межпроцессного взаимодействия (каналы, FIFO, очереди сообщений) может вызвать падение производительности системы. Это, в первую очередь, связано с тем, что данные, передаваемые с помощью этих объектов, копируются из буфера передающего процесса в буфер ядра и затем в буфер принимающего процесса. Механизм разделяемой памяти позволяет избавиться от накладных расходов передачи данных через ядро, предоставляя двум или более процессам возможность непосредственного получения доступа к одной области памяти для обмена данными.

Безусловно, процессы должны предварительно "договориться" о правилах использования разделяемой памяти. Например, пока один из процессов производит запись данных в разделяемую память, другие процессы должны воздержаться от работы с ней. К счастью, задача кооперативного использования разделяемой памяти, заключающаяся в синхронизации выполнения процессов, легко решается с помощью семафоров.

Примерный сценарий работы с разделяемой памятью выглядит следующим образом:

1. Сервер получает доступ к разделяемой памяти, используя семафор.
2. Сервер производит запись данных в разделяемую память.
3. После завершения записи сервер освобождает разделяемую память с помощью семафора.
4. Клиент получает доступ к разделяемой памяти, запирая ресурс с помощью семафора.
5. Клиент производит чтение данных из разделяемой памяти и освобождает ее, используя семафор.

Для каждой области разделяемой памяти, ядро поддерживает структуру данных `shmid_ds`, основными полями которой являются:

<code>struct ipc_perm shm_perm</code>	Права доступа, владельца и создателя области (см. описание <code>ipc_perm</code> выше)
<code>int shm_segsz</code>	Размер выделяемой памяти

ushort	shm_nattch	Число процессов, использующих разделяемую память
time_t	shm_atime	Время последнего присоединения к разделяемой памяти
time_t	shm_dtime	Время последнего отключения от разделяемой памяти
time_t	shm_ctime	Время последнего изменения

Для создания или для доступа к уже существующей разделяемой памяти используется системный вызов *shmget(2)*:

```
#include <sys/types.h>
#include <sys/IPC.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflag);
```

Функция возвращает дескриптор разделяемой памяти в случае успеха, и -1 в случае неудачи. Аргумент *size* определяет размер создаваемой области памяти в байтах. Значения аргумента *shmflag* задают права доступа к объекту и специальные флаги *IPC\_CREAT* и *IPC\_EXCL*. Заметим, что вызов *shmget(2)* лишь создает или обеспечивает доступ к разделяемой памяти, но не позволяет работать с ней. Для работы с разделяемой памятью (чтение и запись) необходимо сначала присоединить (attach) область вызовом *shmat(2)*:

```
#include <sys/types.h>
#include <sys/IPC.h>
#include <sys/shm.h>

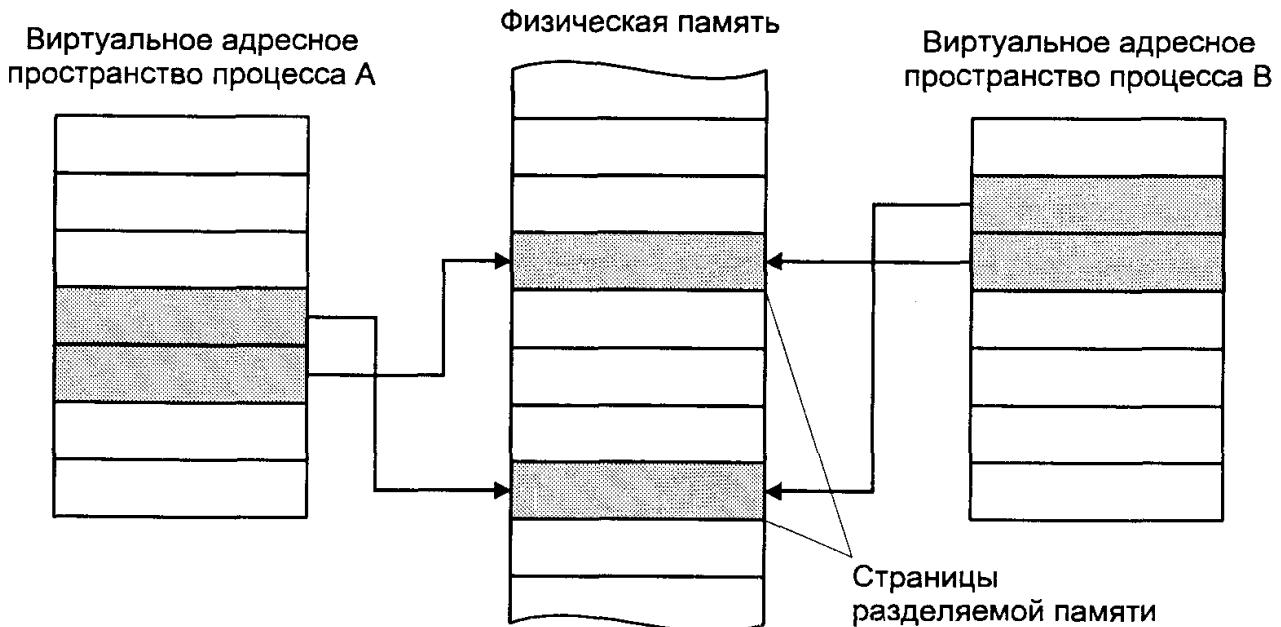
char *shmat(int shmid, char *shmaddr, int shmflag);
```

Вызов *shmat(2)* возвращает адрес начала области в адресном пространстве процесса размером *size*, заданным предшествующем вызовом *shmget(2)*. В этом адресном пространстве взаимодействующие процессы могут размещать требуемые структуры данных для обмена информацией. Правила получения этого адреса следующие:

- Если аргумент *shmaddr* нулевой, то система самостоятельно выбирает адрес.
- Если аргумент *shmaddr* отличен от нуля, значение возвращаемого адреса зависит от наличия флагка *SHM\_RND* в аргументе *shmflag*:
  - Если флагок *SHM\_RND* не установлен, система присоединяет разделяемую память к указанному *shmaddr* адресу.
  - Если флагок *SHM\_RND* установлен, система присоединяет разделяемую память к адресу, полученному округлением в меньшую сторону *shmaddr* до некоторой определенной величины *SHMLBA*.

По умолчанию разделяемая память присоединяется с правами на чтение и запись. Эти права можно изменить, указав флагок *SHM\_RDONLY* в аргументе *shmflag*.

Таким образом, несколько процессов могут отображать область разделяемой памяти в различные участки собственного виртуального адресного пространства, как это показано на рис. 3.20.



**Рис. 3.20.** Совместное использование разделяемой памяти

Окончив работу с разделяемой памятью, процесс отключает (detach) область вызовом *shmdt(2)*:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(char *shmaddr);
```

При работе с разделяемой памятью необходимо синхронизировать выполнение взаимодействующих процессов: когда один из процессов записывает данные в разделяемую память, остальные процессы ожидают завершения операции. Обычно синхронизация обеспечивается с помощью семафоров, назначение и число которых определяется конкретным использованием разделяемой памяти.

Можно привести примерную схему обмена данными между двумя процессами (клиентом и сервером) с использованием разделяемой памяти. Для синхронизации процессов использована группа из двух семафоров. Первый семафор служит для блокирования доступа к разделяемой памяти, его разрешающий сигнал — 0, а 1 является запрещающим сигналом. Второй семафор служит для сигнализации серверу о том, что клиент начал работу. Необходимость применения второго семафора обусловлена следующими обстоятельствами: начальное состояние семафора, синхронизирующего работу с памятью, является открытым (0), и вызов сервером операции *mem\_lock* заблокирует обращение к памяти для клиента. Таким образом,

сервер должен вызвать операцию `mem_lock` только после того, как разделяемую память заблокирует клиент. Назначение второго семафора заключается в уведомлении сервера, что клиент начал работу, заблокировал разделяемую память и начал записывать данные в эту область. Теперь, при вызове сервером операции `mem_lock` его выполнение будет приостановлено до освобождения памяти клиентом, который делает это после окончания записи строки "Здравствуй, Мир!".

### shmem.h:

```
ttdefine MAXBUFF      80
ttdefine PERM        0666
/*Структура данных в разделяемой памяти*/
typedef struct mem_msg{
    int segment;
    char  buff[MAXBUFF];
} Message;
/*Ожидание начала выполнения клиента*/
static struct sembuf proc_wait[1] = {
    1, -1, 0 };
/*Уведомление сервера о том, что клиент начал работу*/
static struct sembuf proc_start[1] = {
    1, 1, 0 };
/*Блокирование разделяемой памяти*/
static struct sembuf mem_lock[2] = {
    0, 0, 0,
    0, 1, 0 };

/*Освобождение ресурса*/
static struct sembuf mem_unlock[1] = {
    0, -1, 0 };
```

### Сервер:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

#include "shmem.h"
main()
{
    Message *msgptr;
    key_t      key;
    int shmid, semid;
    /*Получим ключ. Один и тот же ключ можно использовать как для
    семафора, так и для разделяемой памяти*/
```

```

    if ( (key = ftok("server", 'A')) < 0) {
        printf ("Невозможно получить ключ\n"); exit(1); }

/*Создадим область разделяемой памяти*/
    if ( (shmid= shmget(key, sizeof (Message) ,
                           PERM | IPC_CREAT)) < 0)
    {
        printf("Невозможно создать область \n"); exit(1); }

/*Присоединим ее*/
    if ( (msgptr = (Message *) shmat(shmid, 0, 0)) < 0){
        printf("Ошибка присоединения\n"); exit(1); }

/*Создадим группу из двух семафоров:
Первый семафор — для синхронизации работы с разделяемой памятью
Второй семафор — для синхронизации выполнения процессов*/

    if ( (semid= semget(key, 2, PERM | IPC_CREAT)) < 0){
        printf("Невозможно создать семафор\n"); exit(1); }

/*Ждем, пока клиент начнет работу и заблокирует разделяемую память*/
    if (semop(semid, &proc_wait[0], 1) < 0){
        printf("Невозможно выполнить операцию\n"); exit(1); }

/*Ждем, пока клиент закончит запись в разделяемую память и
освободит ее. После этого заблокируем ее*/
    if (semop(semid, &mem_lock[0], 2) < 0){
        printf ("Невозможно выполнить операцию\n"); exit(1); }

/*Выведем сообщение на терминал*/
    printf ("%s", msgptr->buff);

/*Освободим разделяемую память*/
    if (semop(semid, &mem_unlock[0], 1) < 0){
        printf("Невозможно выполнить операцию\n"); exit(1); }

/*Отключимся от области*/
    if (shmdt(msgptr) < 0){
        printf("Ошибка отключения\n"); exit(1); }

/*Всю остальную работу по удалению объектов сделает клиент*/
    exit(0);
}

```

**Клиент:**

```

ttinclude<sys/types.h>
ttinclude <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "shmem.h"
main()
{
Message *msgptr;

```

```

key_t key;
int shmid, semid;
/*Получим ключ. Один и тот же ключ можно использовать как для
семафора, так и для разделяемой памяти*/
if ( (key = ftok("server", 'A')) < 0) {
    printf("Невозможно получить ключ\n"); exit(1); }
/*Получим доступ к разделяемой памяти*/
if ( (shmid = shmget(key, sizeof(Message), 0)) < 0) {
    printf("Ошибка доступа\n"); exit(1); }

/*Присоединим ее*/
if ( (msgptr = (Message *) shmat(shmid, 0, 0)) < 0) {
    printf("Ошибка присоединения\n"); exit(1); }

/*Получим доступ к семафору*/
if ( (semid = semget(key, 2, PERM)) < 0) {
    printf("Ошибка доступа\n"); exit(1); }
/*Заблокируем разделяемую память*/
if (semop(semid, &mem_lock[0], 2) < 0) {
    printf("Невозможно выполнить операцию\n"); exit(1); }
/*Уведомим сервер о начале работы*/
if (semop(semid, &proc_start[0], 1) < 0) {
    printf("Невозможно выполнить операцию\n"); exit(1); }

/*Запишем в разделяемую память сообщение*/
sprintf(msgptr->buff, "Здравствуй, Мир!\n");

/*Освободим разделяемую память*/
if (semop(semid, &mem_unlock[0], 1) < 0) {
    printf("Невозможно выполнить операцию\n"); exit(1); }

/*Ждем, пока сервер в свою очередь не освободит разделяемую
память*/
if (semop(semid, &mem_lock[0], 2) < 0) {
    printf("Невозможно выполнить операцию\n"); exit(1); }

/*Отключимся от области*/
if (shmdt(msgptr) < 0) {
    printf("Ошибка отключения\n"); exit(1); }

/*Удалим созданные объекты IPC*/
if (shmctl(shmid, IPC_RMID, 0) < 0) {
    printf("Невозможно удалить область\n"); exit(1); }

if (semctl(semid, 0, IPC_RMID) < 0) {
    printf("Невозможно удалить семафор\n"); exit(1); }

exit(0);

```

## Межпроцессное взаимодействие в BSD UNIX. Сокеты

Разработчики системы межпроцессного взаимодействия BSD UNIX руководствовались рядом соображений:

Во-первых, взаимодействие между процессами должно быть унифицировано, независимо от того, выполняются ли они на одном компьютере или на разных хостах сети. Наиболее оптимальная реализация межпроцессного взаимодействия, удовлетворяющего этому требованию, должна иметь модульную структуру и базироваться на общей подсистеме поддержки сети UNIX. При этом могут быть использованы различные схемы адресации объектов, их расположение, протоколы передачи данных и т. д. В этой связи было введено понятие *коммуникационный домен* (communication domain), описывающее набор обозначенных характеристик взаимодействия.

Для обозначения коммуникационного узла, обеспечивающего прием и передачу данных для объекта (процесса), был предложен специальный объект — *сокет* (socket). Сокеты создаются в рамках определенного коммуникационного домена, подобно тому как файлы создаются в рамках файловой системы. Сокеты имеют соответствующий интерфейс доступа в файловой системе UNIX, и так же как обычные файлы, адресуются некоторым целым числом — дескриптором. Однако в отличие от обычных файлов, сокеты представляют собой виртуальный объект, который существует, пока на него ссылается хотя бы один из процессов.

Во-вторых, коммуникационные характеристики взаимодействия должны быть доступны процессам в некоторой унифицированной форме. Другими словами, приложение должно иметь возможность затребовать определенный тип связи, например, основанный на виртуальном канале (virtual circuit) или датаграммах (datagram), причем эти типы должны быть согласованы для всех коммуникационных доменов. Все сокеты условно можно разделить на несколько типов, в зависимости от предоставляемых коммуникационных характеристик. Полный набор этих характеристик включает:

- Упорядоченную доставку данных
- Отсутствие дублирования данных
- Надежную доставку данных
- Сохранение границ сообщений
- Поддержку передачи экстренных сообщений
- Предварительное установление соединения

Например, каналы, рассмотренные ранее, обеспечивают только первые три характеристики. При этом данные имеют вид сплошного потока, выделение сообщений из которого должно при необходимости быть обеспечено взаимодействующими приложениями.

Поддержка передачи экстренных сообщений предполагает возможность доставки данных вне нормального потока. Как правило, это сообщения,

связанные с некоторыми срочными событиями, требующими немедленной реакции.

Взаимодействие с предварительным установлением соединения предполагает создание виртуального канала между источником и получателем данных. Это избавляет от необходимости идентифицировать передающую сторону в каждом пакете данных. Идентификация происходит на начальном этапе установления связи и затем сохраняется для всех пакетов, принадлежащих данному виртуальному каналу.

В BSD UNIX реализованы следующие основные типы сокетов:

- Сокет датаграмм* (datagram socket), через который осуществляется теоретически ненадежная, несвязная передача пакетов.
- Сокет потока* (stream socket), через который осуществляется надежная передача потока байтов без сохранения границ сообщений. Этот тип сокетов поддерживает передачу экстренных данных.
- Сокет пакетов* (packet socket), через который осуществляется надежная последовательная передача данных без дублирования с предварительным установлением связи. При этом сохраняются границы сообщений.
- Сокет низкого уровня* (raw socket), через который осуществляется непосредственный доступ к коммуникационному протоколу.

Наконец, для того чтобы независимые процессы имели возможность взаимодействовать друг с другом, для сокетов должно быть определено *пространство имен*. Имя сокета имеет смысл только в рамках коммуникационного домена, в котором он создан. Если для IPC System V используются ключи, то имена сокетов представлены *адресами*.

### Программный интерфейс сокетов

Итак, сокеты являются коммуникационным интерфейсом взаимодействующих процессов. Конкретный характер взаимодействия зависит от типа используемых сокетов, а коммуникационный домен, в рамках которого создан сокет, определяет базовые свойства этого взаимодействия. В табл. 3.6 приведены типы сокетов и их названия.

**Таблица 3.6.** Типы сокетов в системе BSD UNIX

Название	Тип
SOCK_DGRAM	Сокет датаграмм
SOCK_STREAM	Сокет потока
SOCK_SEQPACKET	Сокет пакетов
SOCK_RAW	Сокет низкого уровня

Для создания сокета процесс должен указать тип сокета и коммуникационный домен, в рамках которого будет использоваться сокет. Поскольку коммуникационный домен может поддерживать использование нескольких протоколов, процесс может также указать конкретный коммуникационный протокол для взаимодействия. Если таковой не указан, система выберет наиболее подходящий из списка протоколов, доступных для данного коммуникационного домена. Если же в рамках указанного домена создание сокета данного типа невозможно, т. е. отсутствует соответствующий коммуникационный протокол, запрос процесса завершится неудачно.

Для создания сокета используется системный вызов *socket(2)*<sup>16</sup>, имеющий следующий вид:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Здесь аргумент *domain* определяет коммуникационный домен, *type* — тип сокета, а *protocol* — используемый протокол (может быть не указан, т. е. приравнен 0). В случае успеха системный вызов возвращает положительное целое число, аналогичное файловому дескриптору, которое служит для адресации данного сокета в последующих вызовах.

По существу коммуникационный домен определяет *семейство протоколов* (protocol family), допустимых в рамках данного домена. Возможные значения аргумента *domain* включают:

AF_UNIX	Домен локального межпроцессного взаимодействия в пределах единой операционной системы UNIX. Внутренние протоколы.
AF_INET	Домен взаимодействия процессов удаленных систем. Протоколы Internet (TCP/IP).
AF_NS	Домен взаимодействия процессов удаленных систем. Протоколы Xerox NS.

Поскольку домен и семейство протоколов определяют адресное пространство взаимодействия (допустимые адреса и их формат), то в названиях доменов присутствует префикс AF (от address family — семейство адресов). Допустимыми также являются названия с префиксом PF (protocol family) PF UNIX, PF\_INET и т. д.

<sup>16</sup> Поскольку сокеты являются неотъемлемой частью BSD UNIX, в системах этой ветви функции, связанные с этими объектами, в частности *socket(2)* и рассмотренные ниже, представляют собой системные вызовы. В UNIX ветви System V интерфейс сокетов сохранен для совместимости, но имеет совершенно отличную от принятой в BSD архитектуру (основанную на подсистеме STREAMS). Поэтому все его функции являются библиотечными и описываются, соответственно в разделе 3 электронного справочника. Однако, оставляя пальму первенства в этом вопросе за BSD UNIX, в этом разделе будем считать эти функции системными вызовами и связывать с ними раздел 2 справочника *man(1M)*.

Заметим, что домен может не поддерживать определенные типы сокетов. Для сравнения в табл. 3.7 приведены два основных коммуникационных домена — внутренний домен UNIX, предназначенный для взаимодействия процессов одной операционной системы, и домен TCP/IP, используемый в сетевых распределенных приложениях.

**Таблица 3.7.** Поддержка различных типов сокетов в доменах

Домен:	AF_UNIX	<b>AF_INET</b>
Тип сокета		
SOCK_STREAM	Да	Да
SOCK_DGRAM	Да	Да
SOCK_SEQPACKET	Нет	Нет
SOCK_RAW	Нет	Да

Также допустимы не все комбинации типа сокета и используемого коммуникационного протокола (если таковой явно указан в запросе). Так для домена AF\_INET возможны следующие комбинации:

Сокет	Протокол
SOCK_STREAM	IPPROTO_TCP (TCP)
SOCK_DGRAM	IPPROTO_UDP (UDP)
SOCK_RAW	IPPROTO_ICMP (ICMP)
SOCK_RAW	IPPROTO_RAW (IP)

Указанные протоколы принадлежат семейству сетевых протоколов TCP/IP и будут подробно рассмотрены в главе 6.

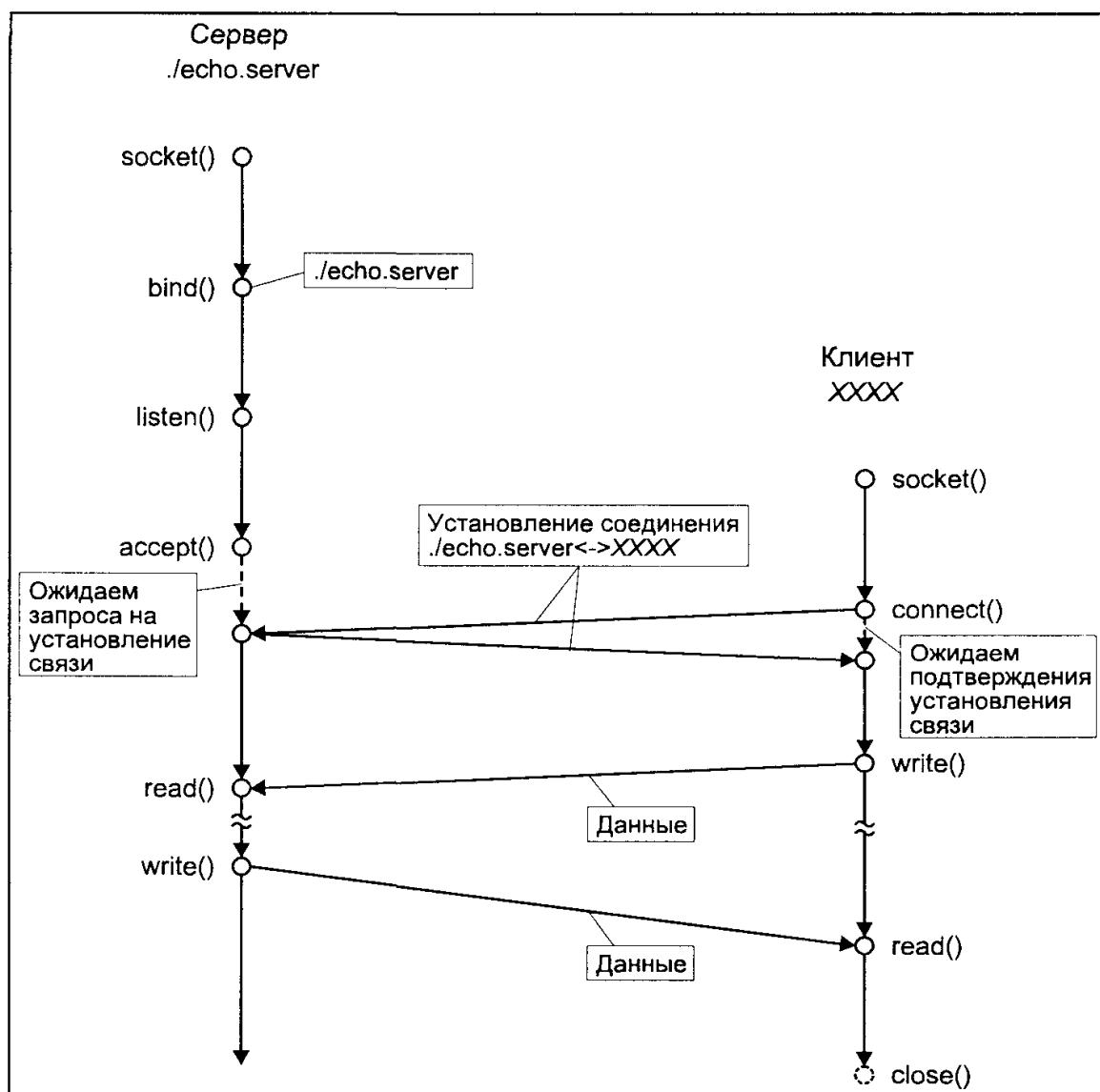
Создание сокета не означает создания коммуникационного узла. Для однозначной идентификации сокета его необходимо позиционировать в пространстве имен данного коммуникационного домена. В общем случае каждый коммуникационный канал определяется двумя узлами — источником и получателем данных, и может быть охарактеризован пятью параметрами:

1. Коммуникационным протоколом
2. Локальным адресом
3. Локальным процессом
4. Удаленным адресом
5. Удаленным процессом

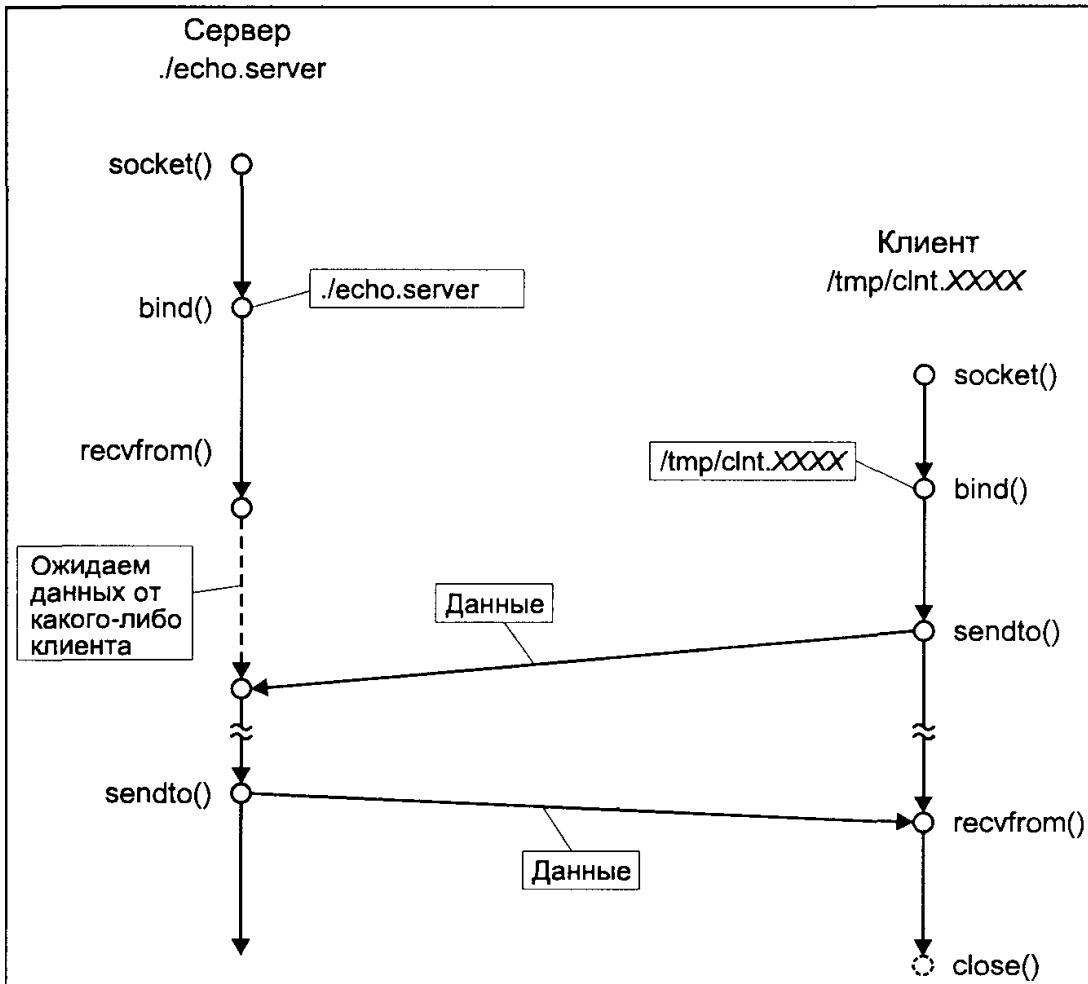
Как правило, адрес определяет операционную систему (или хост сети), а процесс — конкретное приложение, получающее или передающее данные. Однако конкретные значения и формат этих параметров определяются коммуникационным доменом.

Поскольку при создании сокета указывается только один параметр — коммуникационный протокол, прежде чем передача данных между взаимодействующими процессами станет возможной необходимо указать четыре дополнительных параметра для коммуникационного канала. Очевидно, что взаимодействующие стороны должны делать это согласованно, используя либо заранее определенные адреса, либо договариваясь о них в процессе установления связи. Процедура установки этих параметров существенным образом зависит и от типа создаваемого канала, определяемого типом используемого сокета и коммуникационного протокола.

Иллюстрация взаимодействия между процессами при виртуальном коммуникационном канале с предварительным установлением связи приведена на рис. 3.21, а взаимодействие, основанное на датаграммах без установления связи показано на рис. 3.22.



**Рис. 3.21.** Взаимодействие между процессами при создании виртуального канала (с предварительным установлением соединения)



**Рис. 3.22.** Взаимодействие между процессами, основанное на датаграммах (без предварительного установления соединения)

Как видно из рисунков, фактической передаче данных предшествует начальная фаза *связывания* (binding) сокета, когда устанавливается дополнительная информация, необходимая для определения коммуникационного узла. Связывание может быть осуществлено с помощью системного вызова *bind(2)*:

```

#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *localaddr, int addrlen);

```

Здесь *sockfd* является дескриптором сокета, полученным при его создании; аргумент *localaddr* определяет локальный адрес, с которым необходимо связать сокет; параметр *addrlen* определяет размер адреса. Заметим, что речь идет о связывании с локальным адресом, в общем случае определяющим два параметра коммуникационного канала (коммуникационный узел): локальный адрес и локальный процесс.

Как уже обсуждалось, адрес сокета зависит от коммуникационного домена, в рамках которого он определен. В общем случае адрес определяется следующим образом (в файле *<sys/socket.h>*):

```
struct sockaddr {
    u_short sa_family;
    char sa_data[14];
};
```

Поле `sa_family` определяет коммуникационный домен (семейство протоколов), а `sa_data` — содержит собственно адрес, формат которого определен для каждого домена.

Например, для внутреннего домена UNIX адрес выглядит следующим образом (определен в `<sys/un.h>`):

```
struct sockaddr_un {
    short sun_family; /* ==AF_UNIX */
    char sun_path[108];
};
```

Поскольку в данном домене взаимодействующие процессы выполняются под управлением одной операционной системы на одном и том же хосте, коммуникационный узел может быть однозначно определен одним параметром — локальным процессом. В качестве адреса в домене UNIX используются имена файлов.

В отличие от локального межпроцессного взаимодействия, для сетевого обмена данными необходимо указание как локального процесса, так и хоста, на котором выполняется данный процесс. Для домена Internet (семейство протоколов TCP/IP) используется следующий формат адреса (определен в файле `<netinet/in.h>`):

```
struct sockaddr_in {
    short sin_family; /* ==AF_INET */
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

Адреса этого домена (IP-адреса) будут рассмотрены подробнее в главе 6. Пока лишь заметим, что адрес хоста представляет собой 32-разрядное целое число `sin_addr`, а процесс (приложение) адресуется 16-разрядным номером порта `sin_port`.

На рис. 3.23 показаны рассмотренные форматы адресов сокетов.

Итак, связывание необходимо для присвоения сокету локального адреса и, таким образом, для определения коммуникационного узла. Можно выделить три случая использования для этого функции `bind(2)`:

1. Сервер регистрирует свой адрес. Этот адрес должен быть заранее известен клиентам, желающим "общаться" с сервером. Связывание необходимо, прежде чем сервер будет готов к приему запросов от клиентов.
2. При взаимодействии без предварительного установления связи и создания виртуального канала клиент также должен предварительно за-

регистрировать свой адрес. Этот адрес должен быть уникальным в рамках коммуникационного домена. В случае домена UNIX об этом должно позаботиться само приложение. Этот адрес не должен быть заранее известен серверу, поскольку запрос всегда инициирует клиент, автоматически передавая вместе с ним свой адрес. Полученный адрес удаленного узла затем используется сервером для мультиплексирования сообщений, отправляемым различным клиентам.

3. Даже в случае взаимодействия с использованием виртуального канала клиент может пожелать зарегистрировать собственный адрес, не полагаясь при этом на систему.

UNIX домен		Internet домен	
<b>sockaddr un</b>		<b>sockaddr in</b>	
AF_UNIX	2 байта	AF_INET	2 байта
Имя файла	до 10	port	2 байта
		IP-адрес	4 байта
		Не используется	8 байт

**Рис. 3.23.** Адреса сокетов

Назначение адреса для клиента также можно выполнить с помощью системного вызова *connect(2)*, устанавливающего связь с сервером и автоматически связывающего сокет клиента с локальным коммуникационным узлом. Вызов *connect(2)* имеет вид:

```
ttinclude<sys/types.h>
ttinclude<sys/socket.h>

int connect(int sockfd, struct sockaddr *servaddr, int addrlen);
```

Характер этого вызова предполагает создание виртуального канала и, таким образом, используется для предварительного установления связи между коммуникационными узлами. В этом случае клиенту нет необходимости

явно связывать сокет с помощью системного вызова *bind(2)*. Локальный узел коммуникационного канала указывается дескриптором сокета *sockfd*, для которого система автоматически выбирает приемлемые значения локального адреса и процесса. Удаленный узел определяется аргументом *servaddr*, который указывает на адрес сервера, а *addrlen* задает его длину.

Вызов *connect(2)* может также применяться и клиентами, использующими сокеты датаграмм без создания виртуального канала. В этом случае *connect(2)* не вызывает фактического соединения с сервером, а является удобным способом сохранения параметров адресата (сервера), которому будут направляться датаграммы. При этом клиент будет избавлен от необходимости указывать адрес сервера при каждом отправлении данных.

Следующие два вызова используются сервером только при взаимодействии, основанном на предварительном создании виртуального канала между сервером и клиентом.

Системный вызов *listen(2)* информирует систему, что сервер готов принимать запросы. Он имеет следующий вид:

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Здесь параметр *sockfd* определяет сокет, который будет использоваться для получения запросов. Предполагается, что сокет был предварительно связан с известным адресом. Параметр *backlog* указывает максимальное число запросов на установление связи, которые могут ожидать обработки сервером<sup>17</sup>.

Фактическую обработку запроса клиента на установление связи производит системный вызов *accept(2)*:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *clntaddr,
           int *addrallen);
```

Вызов *accept(2)* извлекает первый запрос из очереди и создает новый сокет, характеристики которого не отличаются от сокета *sockfd*, и таким

<sup>17</sup> Если в момент получения запроса на установление связи очередь ожидающих запросов достигла своего максимального значения, вызов *connect(2)* клиента завершится с ошибкой *ECONNREFUSED* для домена *UNIX* (*AF\_UNIX*). Для других доменов результат зависит от того, поддерживает ли протокол повторную передачу запроса. Например, протокол *TCP* (домен *AF\_INET*) будет передавать повторные запросы, пока число запросов в очереди не уменьшится, либо не произойдет тайм-аут, определенный для протокола. В последнем случае вызов клиента завершится с ошибкой *ETIMEDOUT*.

образом завершает создание виртуального канала со стороны сервера. Одновременно *accept(2)* возвращает параметры удаленного коммуникационного узла — адрес клиента *clntaddr* и его размер *addrulen*. Новый сокет используется для обслуживания созданного виртуального канала, а полученный адрес клиента исключает анонимность последнего. Дальнейший типичный сценарий взаимодействия имеет вид:

sockfd = socket(...);	Создать сокет
bind(sockfd, ...);	Связать его с известным локальным адресом
listen(sockfd, ...);	Организовать очередь запросов
for ( ; ; ) {	
newsockfd = accept(sockfd, ...);	Получить запрос
if (fork() == 0) {	Породить дочерний процесс
close (sockfd);	Дочерний процесс
}	
else	
close (newsockfd);	Родительский процесс
}	

В этом сценарии, в то время как дочерний процесс обеспечивает фактический обмен данными с клиентом, родительский процесс продолжает "прослушивать" поступающие запросы, порождая для каждого из них отдельный процесс-обработчик. Очередь позволяет буферизировать запросы на время, пока сервер завершает вызов *accept(2)* и затем создает дочерний процесс. Заметим, что новый сокет *newsockfd*, полученный в результате вызова *accept(2)*, адресует полностью определенный коммуникационный канал: протокол и полные адреса обоих узлов — клиента и сервера. Напротив, для сокета *sockfd* определена только локальная часть канала. Это позволяет серверу продолжать использовать *sockfd* для "прослушивания" последующих запросов.

Наконец, если для сокетов потока при приеме и передаче данных могут быть использованы стандартные вызовы *read(2)* и *write(2)*, то сокеты датаграмм должны пользоваться специальными системными вызовами (эти вызовы также доступны для сокетов других типов):

```
ftinclude<sys/types.h>
#include <sys/socket.h>
int send(int s, const char *msg, int len, int flags);
int sendto(int s, const char *msg, int len, int flags,
           const struct sockaddr *toaddr, int tolen);
int recv(int s, char *buf, int len, int flags);
int recvfrom(int s, char *buf, int len, int flags,
             struct sockaddr *fromaddr, int *fromlen);
```

Функции *send(2)* и *sendto(2)* используются для передачи данных удаленному узлу, а функции *recv(2)* и *recvfrom(2)*— для их приема. Основным различием между ними является то, что функции *send(2)* и *recv(2)* могут быть использованы только для "подсоединенного" сокета, т. е. после вызова *connect(2)*.

Все эти вызовы используют в качестве первого аргумента дескриптор сокета, через который производится обмен данными. Аргумент *msg* содержит сообщение длиной *len*, которое должно быть передано по адресу *toaddr*, длина которого составляет *tolen* байтов. Для функции *send(2)* используется адрес получателя, установленный предшествовавшим вызовом *connect(2)*. Аргумент *buf* представляет собой буфер, в который копируются полученные данные.

Параметр *flags* может принимать следующие значения:

MSG_OOB	Передать или принять экстренные данные (out-of-band) вместо обычных
MSG_PEEK	Просмотреть данные, не удаляя их из системного буфера (последующие операции чтения получат те же данные)

### Пример использования сокетов

В заключение приведем пример использования сокетов для организации межпроцессного взаимодействия. Поскольку в данном разделе не затрагиваются сетевые вопросы, то и сокеты, которые будут использованы в примере, принадлежат домену UNIX. Как и в предыдущих примерах, функциональность нашей распределенной системы не отличается разнообразием: клиент посыпает серверу сообщение "Здравствуй, Мир!", а сервер отправляет его обратно клиенту, который после получения выводит сообщение на экран.

В примере использованы сокеты датаграмм, которые в домене UNIX практически не отличаются от сокетов потока. В качестве адреса сервера предлагается имя файла **./echo.server** (мы полагаем, что в системе запущен только один сервер из данного каталога). Предполагается, что клиенты заранее знают этот адрес. Сервер связывает созданный сокет с этим локальным адресом и таким образом регистрируется в системе. Начиная с этого момента он готов к получению и обработке сообщений. Сервер начинает бесконечный цикл, ожидая сообщений от клиентов, блокируясь на вызове *recvfrom(2)*. При получении сообщения сервер отправляет его обратно, вызывая *sendto(2)*.

#### Сервер:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#define MAXBUF 256
char buf[MAXBUF];
```

```

    struct sockaddr_un serv_addr, clnt_addr;
    int sockfd;
    int saddrlen, caddrlen, max_caddrlen, n;
/* Создадим сокет*/
    if ((sockfd= socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
        printf("Невозможно создать сокет\n"); exit(1); }
/* Связем сокет с известным локальным адресом. Поскольку адрес в
домене UNIX представляет собой имя файла, который будет создан
системным вызовом bind(2), сначала удалим файл с этим именем в
случае, если он сохранился от предыдущего запуска сервера */
    unlink("./echo.server");
    bzero(&serv_addr, sizeof(serv_addr));
    serv_addr.sun_family = AF_UNIX;
    strcpy(serv_addr.sun_path, "./echo.server");
    saddrlen = sizeof(serv_addr.sun_family) +
                strlen(serv_addr.sun_path);

    if (bind(sockfd, (struct sockaddr *)&serv_addr,
              saddrlen) < 0)
    {
        printf("Ошибка связывания сокета с адресом\n");
        exit(1);
    }
/* Теперь запустим бесконечный цикл чтения сообщений от клиентов
и отправления их обратно */
    max_caddrlen = sizeof(clnt_addr);
    for ( ; ; ) {
        caddrlen = max_caddrlen;
        n = recvfrom(sockfd, buf, MAXBUF, 0,
                     (struct sockaddr *)&clnt_addr, &caddrlen);
        if ( n < 0 ) { printf("Ошибка приема\n"); exit(1); }
/* Благодаря вызову recvfrom (2), мы знаем адрес клиента, от
которого получено сообщение. Используем этот адрес для передачи
сообщения обратно отправителю*/
        if (sendto(sockfd, buf, n, 0,
                   (struct sockaddr *)&clnt_addr, caddrlen) != n) {
            printf("Ошибка передачи\n"); exit(1); }
    }
}
}

```

Клиент создает сокет датаграмм и связывает его со своим уникальным адресом. Уникальность адреса определяется уникальностью имени файла. Поскольку одновременно могут работать несколько клиентов, возникает задача выполнения условия уникальности. Для этого мы используем функцию *mktemp(3C)*, позволяющую по заданному шаблону **/tmp/clnt.XXXX** и на основании идентификатора текущего процесса получить уникальное имя, заменяя соответствующим образом символы 'X'. Связывание сокета позволяет при отправлении сообщения неявно указать его "адрес отправителя", так что серверу не составляется труда отправить сообщение обратно.

**Клиент:**

```

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
char *msg = "Здравствуй, Мир!\n";
#define MAXBUF 256
char buf[MAXBUF];
main()
{
    struct sockaddr_un serv_addr, clnt_addr;
    int sockfd;
    int saddrllen, caddrllen, msglen, n;
    /* Установим адрес сервера, с которым мы будем обмениваться
    данными. Для этого заполним структуру данных sockaddr_un, которую
    будем использовать при отправлении данных серверу с помощью
    вызова sendto(). Значение адреса известно по предварительной
    договоренности */
    bzero(&serv_addr, sizeof(serv_addr));
    serv_addr.sun_family = AF_UNIX;
    strcpy(serv_addr.sun_path, "./echo.server");
    saddrllen = sizeof(serv_addr.sun_family) +
                strlen(serv_addr.sun_path);
    /* Создадим сокет датаграмм */
    if ((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
        printf("Невозможно создать сокет\n"); exit(1);
    }
    /* Необходимо связать сокет с некоторым локальным адресом, чтобы
    сервер имел возможность возвратить посланное сообщение. Этот ад-
   рес должен быть уникальным в пределах коммуникационного домена —
    т. е. данной операционной системы. Для обеспечения этого условия,
    воспользуемся функцией mktemp(3С), которая возвращает уникальное
    имя, основанное на предоставленном шаблоне и идентификаторе нашего
    процесса PID*/
    bzero(&clnt_addr, sizeof(clnt_addr));
    clnt_addr.sun_family = AF_UNIX;
    strcpy(clnt_addr.sun_path, "/tmp/clnt.XXXX");
    mktemp(clnt_addr.sun_path);
    caddrllen = sizeof(clnt_addr.sun_family) +
                strlen(clnt_addr.sun_path);

    if (bind(sockfd, (struct sockaddr *)&clnt_addr,
              caddrllen) < 0)
    {
        printf("Ошибка связывания сокета\n"); exit(1);
    }
    /* Итак, отправляем скромное приветствие */
    msglen = strlen(msg);

    if (sendto(sockfd, msg, msglen, 0,
               (struct sockaddr *)&serv_addr, saddrllen) != msglen)
    {
        printf("Ошибка передачи сообщения\n"); exit(1);
    }
}

```

```

/* Прочитаем эхо*/
    if ((n = recvfrom(sockfd, buf, MAXBUF, 0, NULL, 0)) < 0)
    {
        printf ("Ошибка получения сообщения \n"); exit(1);
    }
/* И выведем его на экран */
    printf("Эхо: %s\n", buf);
/* Уберем за собой */
    close(sockfd);
    unlink(clnt_addr.sun_path);
    exit(0);
}

```

### Сравнение различных систем межпроцессного взаимодействия

Заканчивая разговор о межпроцессном взаимодействии в UNIX, приведем сводную сравнительную таблицу рассмотренных систем.

	Каналы	FIFO	Сообщения	Разделяемая память	Сокеты (домен UNIX)
Пространство имен	—	Имя файла	Ключ	Ключ	Имя файла
Объект	Системный канал	Именованный канал	Очередь сообщений	Разделяемая область памяти	Коммуникационный узел
Создание объекта	pipe()	mknod()	msgget()	shmget()	socket()
Связывание	pipe()	open()	msgget()	shmat()	bind() connect()
Передача данных	read() write()	read() write()	msgrecv() msgsnd()	Непосредственный доступ memcpuy()	read() write() recv() send() recvfrom() sendto()
Уничтожение	close()	close() unlink()	msgctl()	shmdt()	close() unlink()

Если говорить о производительности IPC, то наиболее быстрым способом передачи данных между неродственными процессами является разделяемая память. Разделяемая память является частью адресного пространства для каждого из взаимодействующих процессов, поэтому чтение и запись в эту область неотличимы, например, от чтения и записи в область собственных данных процесса. Однако при использовании разделяемой памяти необходимо

димо обеспечить синхронизацию процессов. При использовании семафоров, необходимо иметь в виду следующие обстоятельства:

- Применение семафоров может увеличить число процессов в очереди на выполнение, поскольку несколько процессов, ожидающих разрешающего сигнала семафора, будут одновременно разбужены и переведены в очередь на выполнение.
- Применение семафоров увеличивает число переключений контекста, что, в свою очередь, увеличивает нагрузку на систему.
- В то же время, использование семафоров является наиболее стандартным (POSIX.1b), хотя и неэффективным способом обеспечения синхронизации.

Очереди сообщений предназначены для обмена короткими (обычно менее 1 Кбайт) структурами данных. Если объем данных превышает эту величину, использование сообщений может значительно увеличить число системных вызовов и уменьшить производительность операционной системы.

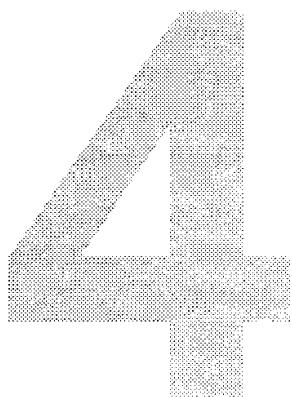
Интенсивность межпроцессного взаимодействия в системе можно определить с помощью команды *sar -m*. Вывод команды показывает число использования объектов IPC в секунду:

	msg/s	sema/s
17 47:58	0.20	20.00
17 48:03	0.60	12.20
17 48:08	2.20	10.40
17 13	0.80	25.10
17 18	0.00	15.60
Average	0.76	16.66

## Заключение

В этой главе начато обсуждение внутренней архитектуры ядра UNIX, которое будет продолжено в следующих главах. Поскольку процессы являются движущей силой операционной системы, мы начали обсуждение именно с этого вопроса. Действительно, не считая нескольких системных процессов, являющихся частью ядра и выполняющих узкосистемные функции, основная работа операционной системы происходит по запросам и в контексте прикладных процессов.

В главе обсуждается, каким образом прикладной процесс взаимодействует с ядром операционной системы, как происходит справедливое распределение системных ресурсов между задачами, и тем самым обеспечивается многозадачность UNIX. Также рассматриваются принципы организации виртуальной памяти, когда каждый процесс имеет независимое адресное пространство, размер которого в ряде случаев значительно превышает объем оперативной памяти компьютера. Наконец, здесь представлены структуры данных ядра, связанные с управлением процессами и памятью.



## Файловая подсистема

БОЛЬШИНСТВО данных в операционной системе UNIX хранится в файлах, организованных в виде дерева и расположенных на некотором носителе данных. Обычно это локальный (т. е. расположенный на том же компьютере, что и сама операционная система) жесткий диск, хотя специальный тип файловой системы — NFS (Network File System) обеспечивает хранение файлов на удаленном компьютере. Файловая система также может располагаться на CD-ROM, дисках и других типах носителей, однако для простоты изложения сначала мы рассмотрим традиционную файловую систему UNIX, расположенную на обычном жестком диске компьютера.

Исконной файловой системой UNIX System V является s5fs. Файловая система, разработанная в Беркли, FFS, появилась позже, в версии 4.2BSD UNIX. По сравнению с s5fs она обладает лучшей производительностью, функциональностью и надежностью. Файловые системы современных версий UNIX имеют весьма сложную архитектуру, различную для разных версий. Несмотря на это все они используют базовые идеи, заложенные разработчиками UNIX в AT&T и Калифорнийском университете в Беркли. Поэтому мы проиллюстрируем основные принципы организации файловой системы UNIX на примере базовых систем System V (s5fs) и BSD (FFS), которые, кстати, и сегодня поддерживаются в большинстве версий UNIX.

Когда появилась файловая система FFS, архитектура UNIX поддерживала работу только с одним типом файловой системы. Таким образом, создатели различных версий операционной системы UNIX вынуждены были выбирать одну файловую систему из нескольких возможных. Это неудобство было преодолено введением *независимой* или *виртуальной файловой системы* — архитектуры, позволяющей обеспечивать работу с несколькими "физическими" файловыми системами различных типов. В этой главе мы рассмотрим реализацию виртуальной файловой системы, разработанную фирмой Sun Microsystems. Данная архитектура является стандартом для SVR4, однако и другие версии UNIX используют подобные подходы. В качестве примера можно привести независимую файловую систему SCO UNIX.

Далее мы рассмотрим схему доступа прикладных процессов к файлам — всю цепочку структур данных от файловых дескрипторов процесса до фактических дисковых данных, которую операционная система создает в результате открытия процессом файла и которая затем используется для обмена данными.

В заключение мы рассмотрим буферный кэш — подсистему, которая позволяет значительно увеличить производительность работы с дисковыми данными.

## Базовая файловая система **System V**

Каждый жесткий диск состоит из одной или нескольких логических частей, называемых *разделами* (partitions). Расположение и размер раздела определяются при форматировании диска. В UNIX разделы выступают в качестве независимых устройств, доступ к которым осуществляется как к различным носителям данных.

Например, диск может состоять из четырех разделов, каждый из которых содержит свою файловую систему. Заметим, что в разделе может располагаться только одна файловая система, которая не может занимать несколько разделов. В другой конфигурации диск может состоять только из одного раздела, позволяя создание весьма емких файловых систем.

Файловая система *s5fs* занимает раздел диска и состоит из трех основных компонентов, как показано на рис. 4.1.

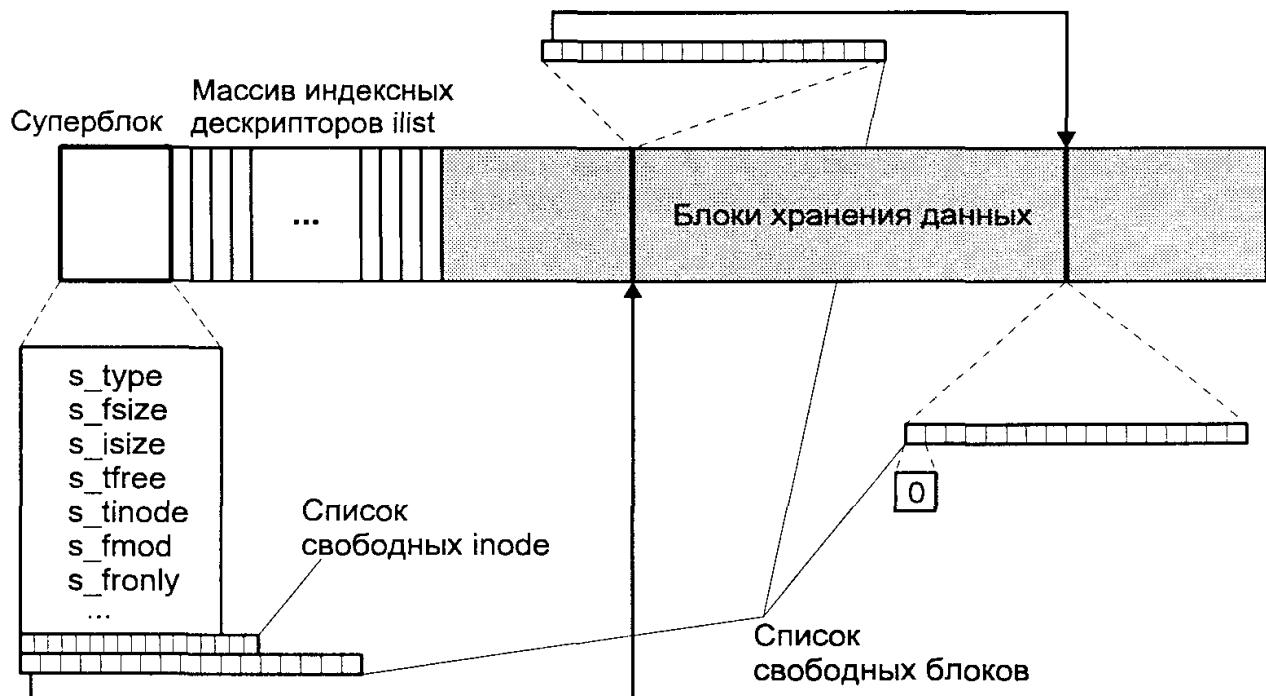


Рис. 4.1. Структура файловой системы *s5fs*

- **Суперблок (superblock).** Содержит общую информацию о файловой системе, например, об ее архитектуре, общем числе блоков и индексных дескрипторов, или метаданных (inode).
- П **Массив индексных дескрипторов (ilist).** Содержит метаданные всех файлов файловой системы. Индексный дескриптор содержит статусную информацию о файле и указывает на расположение данных этого файла. Ядро обращается к inode по индексу в массиве ilist. Один inode является корневым (root) inode файловой системы, через него обеспечивается доступ к структуре каталогов и файлов после монтирования файловой системы. Размер массива ilist является фиксированным и задается при создании файловой системы. Таким образом, файловая система s5fs имеет ограничение по числу файлов, которые могут храниться в ней, независимо от размера этих файлов.
- П **Блоки хранения данных.** Данные обычных файлов и каталогов хранятся в блоках. Обработка файла осуществляется через inode, содержащего ссылки на блоки данных. Блоки хранения данных занимают большую часть дискового раздела, и их число определяет максимальный суммарный объем файлов данной файловой системы. Размер блока кратен 512 байтам, например файловая система S51K SCO UNIX использует размер блока в 1 Кбайт (отсюда и название).

Рассмотрим подробнее каждый из перечисленных компонентов.

## Суперблок

Суперблок содержит информацию, необходимую для монтирования и управления работой файловой системы в целом (например, для размещения новых файлов). В каждой файловой системе существует только один суперблок, который располагается в начале раздела. Суперблок считывается в память при монтировании файловой системы и находится там до ее отключения (размонтирования).

Суперблок содержит следующую информацию:

- П Тип файловой системы (s\_type)
- П Размер файловой системы в логических блоках, включая сам суперблок, ilist и блоки хранения данных (s\_fsize)
- П Размер массива индексных дескрипторов (s\_isize)
- П Число свободных блоков, доступных для размещения (s\_tfree)
- П Число свободных inode, доступных для размещения (s\_tinode)
- П Флаги (флаг модификации s\_fmod, флаг режима монтирования s\_fronly)
- П Размер логического блока (512, 1024, 2048)
- П Список номеров свободных inode
- П Список адресов свободных блоков

Поскольку число свободных inode и блоков хранения данных может быть значительным, хранение двух последних списков целиком в суперблоке непрактично. Например, для индексных дескрипторов хранится только часть списка. Когда число свободных inode в этом списке приближается к 0, ядро просматривает `ilist` и вновь формирует список свободных inode. Для этого ядро анализирует поле `di_mode` индексного дескриптора, которое равно 0 у свободных inode.

К сожалению, такой подход неприменим в отношении свободных блоков хранения данных, поскольку по содержимому блока нельзя определить, свободен он или нет. Поэтому необходимо хранить список адресов свободных блоков целиком. Список адресов свободных блоков может занимать несколько блоков хранения данных, но суперблок содержит только один блок этого списка. Первый элемент этого блока указывает на блок, хранящий продолжение списка и т. д., как это показано на рис. 4.1.

Выделение свободных блоков для размещения файла производится с конца списка суперблока. Когда в списке остается единственный элемент, ядро интерпретирует его как указатель на блок, содержащий продолжение списка. В этом случае содержимое этого блока считывается в суперблок и блок становится свободным. Такой подход позволяет использовать дисковое пространство под списки, пропорциональное свободному месту в файловой системе. Другими словами, когда свободного места практически не остается, список адресов свободных блоков целиком помещается в суперблоке.

## Индексные дескрипторы

Индексный дескриптор, или inode, содержит информацию о файле, необходимую для обработки данных, т. е. *метаданные* файла. Каждый файл ассоциирован с одним inode, хотя может иметь несколько имен в файловой системе, каждое из которых указывает на один и тот же inode.

Индексный дескриптор не содержит:

- имени файла, которое содержится в блоках хранения данных каталога;
- содержимого файла, которое размещено в блоках хранения данных.

При открытии файла ядро помещает копию дискового inode в память в таблицу *in-core inode*, которая содержит несколько дополнительных полей. Структура дискового inode (`struct dinode`) приведена на рис. 4.2. Основные поля дискового inode следующие:

<code>di_mode</code>	Тип файла, дополнительные атрибуты выполнения и права доступа.
<code>di_nlinks</code>	Число ссылок на файл, т. е. количество имен, которые имеет файл в файловой системе.
<code>di_uid, di_gid</code>	Идентификаторы владельца-пользователя и владельца-группы.

di_size	Размер файла в байтах. Для специальных файлов это поле содержит старший и младший номера устройства.
di_atime	Время последнего доступа к файлу.
di_mtime	Время последней модификации.
di_ctime	Время последней модификации inode (кроме модификации полей di_atime, di_mtime).
di_addr[13]	Массив адресов дисковых блоков хранения данных.

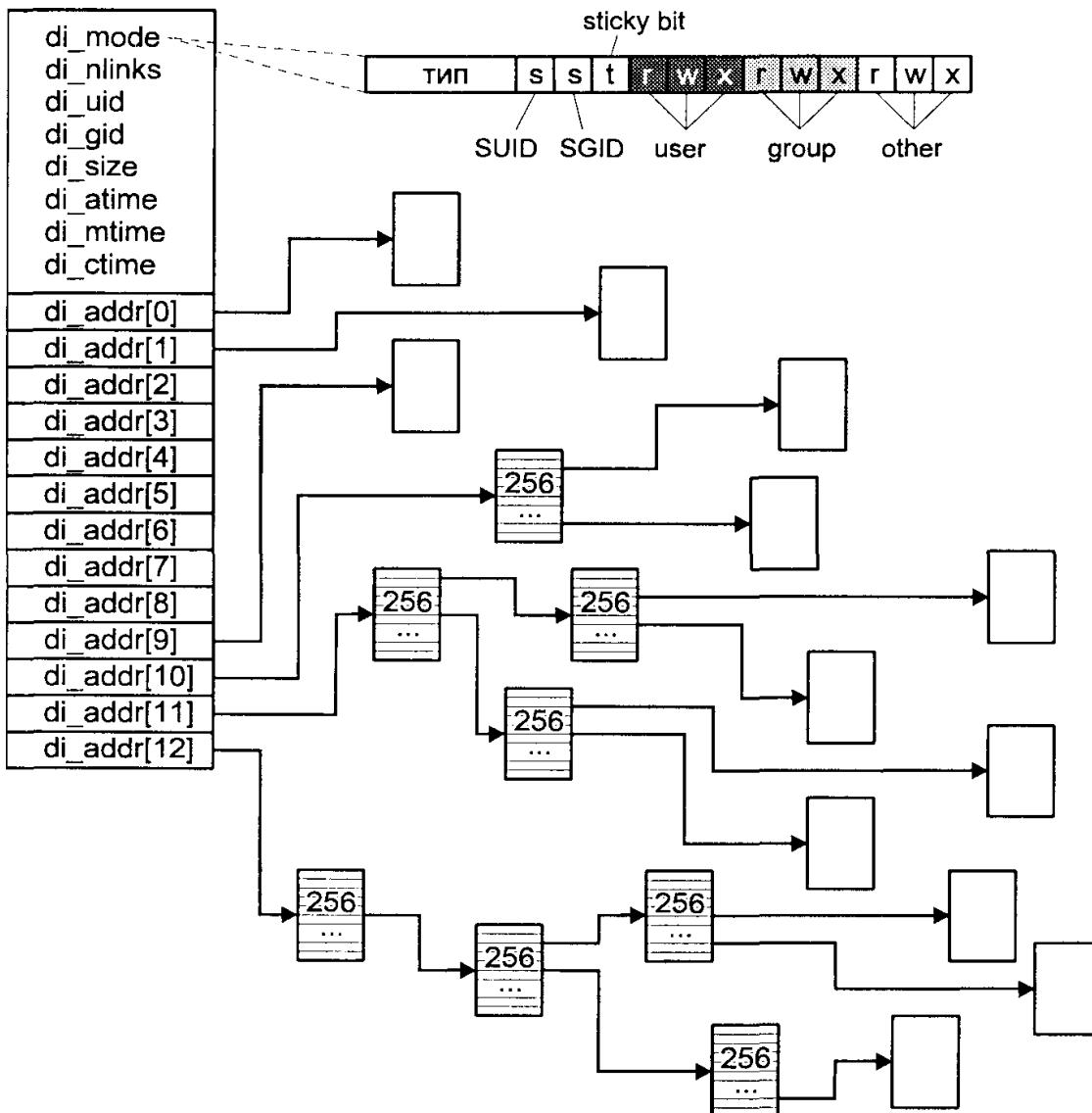


Рис. 4.2. Структура дискового inode

Поле di\_mode хранит несколько атрибутов файла: тип файла (IFREG для обычных файлов, IFDIR для каталогов, IFBLK или IFCHR для специальных файлов блочных и символьных устройств соответственно); права доступа к файлу для трех классов пользователей и дополнительные атрибуты выполнения (SUID, SGID и sticky bit), значения этих атрибутов были подробно рассмотрены в главе 1.

Заметим, что в индексном дескрипторе отсутствует информация о времени создания файла. Вместо этого inode хранит три значения времени: время последнего доступа (`di_atime`), время последней модификации содержимого файла (`di_mtime`) и время последней модификации метаданных файла (`di_ctime`). В последнем случае не учитываются модификации полей `di_atime` и `di_mtime`. Таким образом, `di_ctime` изменяется, когда изменяется размер файла, владелец, группа, или число связей.

Индексный дескриптор содержит информацию о расположении данных файла. Поскольку дисковые блоки хранения данных файла в общем случае располагаются не последовательно, inode должен хранить физические адреса всех блоков, принадлежащих данному файлу<sup>1</sup>. В индексном дескрипторе эта информация хранится в виде массива, каждый элемент которого содержит физический адрес дискового блока, а индексом массива является номер логического блока файла. Массив имеет фиксированный размер и состоит из 13 элементов. При этом первые 10 элементов адресуют непосредственно блоки хранения данных файла. Одиннадцатый элемент адресует блок, в свою очередь содержащий адреса блоков хранения данных. Двенадцатый элемент указывает на дисковый блок, также хранящий адреса блоков, каждый из которых адресует блок хранения данных файла. И, наконец, тринадцатый элемент используется для тройной косвенной адресации, когда для нахождения адреса блока хранения данных файла используются три дополнительных блока.

Такой подход позволяет при относительно небольшом фиксированном размере индексного дескриптора поддерживать работу с файлами, размер которых может изменяться от нескольких байтов до десятка мегабайтов. Для относительно небольших файлов (до 10 Кбайт при размере блока 1024 байтов) используется прямая индексация, обеспечивающая максимальную производительность. Для файлов, размер которых не превышает 266 Кбайт (10 Кбайт + 256x1024), достаточно простой косвенной адресации. Наконец, при использовании тройной косвенной адресации можно обеспечить доступ к 16777216 блокам (256x256x256).

Файлы в UNIX могут содержать так называемые *дыры*. Например, процесс может создать пустой файл, с помощью системного вызова `lseek(2)` сме-

Размещение данных файла в произвольно расположенных дисковых блоках позволяет эффективно использовать дисковое пространство, поскольку ядро может использовать любой свободный дисковый блок для размещения данных. Однако в файловой системе `ssfs` блок может использоваться только одним файлом, поэтому последний блок файла используется, как правило, не полностью. К тому же такой подход с течением времени приводит к увеличению фрагментации системы, когда данные файла оказываются произвольно разбросанными по диску, что, в свою очередь, увеличивает время доступа к файлу и уменьшает производительность обмена данными. Единственным способом уменьшения фрагментации файловой системы является создание полной резервной копии на другом носителе (или в другой файловой системе) и затем ее восстановления. При этом запись файлов будет производиться последовательно без фрагментации.

стить файловый указатель относительно начала файла и записать данные. При этом между началом файла и началом записанных данных образуется дыра — незаполненная область. При чтении этой области процесс получит обнуленные байты. Поскольку логические блоки, соответствующие дыре, не содержат данные, не имеет смысла размещать для них дисковые блоки. В этом случае соответствующие элементы массива адресов *inode* содержат нулевой указатель. Когда процесс производит чтение такого блока, ядро возвращает последовательность нулей. Дисковые блоки размещаются только при записи в соответствующие логические блоки файла<sup>2</sup>.

## Имена файлов

Как мы уже видели, ни метаданные, ни тем более блоки хранения данных, не содержат имени файла. Имя файла хранится в файлах специального типа — каталогах. Такой подход позволяет любому файлу, т. е. фактическим данным, иметь теоретически неограниченное число имен (названий), в файловой системе. При этом несколько имен файлов будут соответствовать одним и тем же метаданным иенным и являться жесткими связями.

Каталог файловой системы *s5fs* представляет собой таблицу, каждый элемент которой имеет фиксированный размер в 16 байтов: 2 байта хранят номер индексного дескриптора файла, а 14 байтов — его имя. Это накладывает ограничение на число *inode*, которое не может превышать 65 535. Также ограничена и длина имени файла: его максимальный размер — 14 символов. Структура каталога приведена на рис. 4.3.

Первые два элемента каталога адресуют сам каталог (текущий каталог) под именем “.” и родительский каталог под именем “..”.

При удалении имени файла из каталога (например, с помощью команды *rm(1)*), номер *inode* соответствующего элемента устанавливается равным 0. Ядро обычно не удаляет такие свободные элементы, поэтому размер каталога не уменьшается даже при удалении файлов. Это является потенциальной проблемой для каталогов, в которые временно было помещено большое количество файлов. После удаления большинства из них размер каталога останется достаточно большим, поскольку записи удаленных файлов будут по-прежнему существовать.

<sup>2</sup> Отсутствие размещенных дисковых блоков для части файла может привести к нежелательным результатам. Например, операция записи в “дыру” может закончиться неудачей из-за нехватки дискового пространства. При копировании файла с дырой, его копия будет занимать больше фактического места на диске, чем оригинал. Это связано с тем, что при копировании производится чтение содержимого оригинала, а затем — запись в другой файл. Это, в частности может привести к тому, что резервная копия файловой системы не сможет быть обратно распакована, поскольку вместо неразмещенных блоков будет хранить законные нулевые байты и, соответственно, занимать больше места.

Иллюстрацию этого явления в SCO UNIX можно привести, применив команду *hd(1M)*, обеспечивающую вывод неинтерпретированного содержимого файла (шестнадцатеричный дамп).

Можно заметить, что имен файлов, расположенных во второй части вывода команды `hd(1M)` на самом деле не существует — об этом свидетельствуют нулевые значения номеров `inode`, это же подтверждает вывод команды `ls(1)`:

```
$ ls -a
.news
bin
dead.letter
News
mail
```

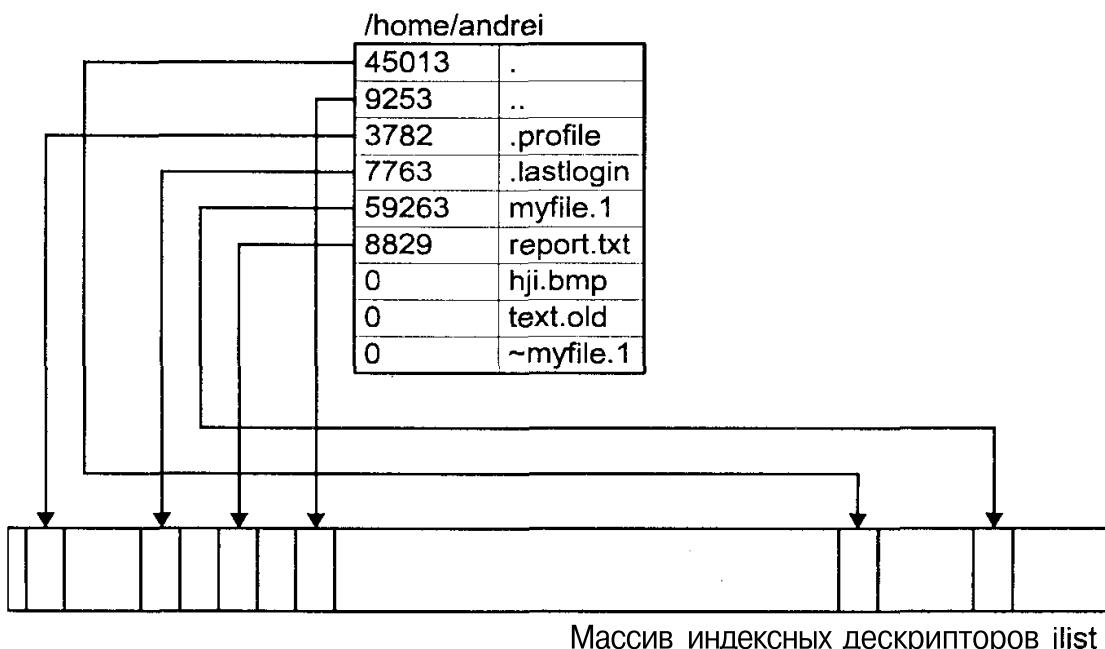


Рис. 4.3. Каталог файловой системы `s5fs`

## Недостатки и ограничения

Файловая систем s5fs привлекательна благодаря своей простоте. Однако обратной стороной медали является низкая надежность и производительность.

С точки зрения надежности слабым местом этой файловой системы является суперблок. Суперблок несет основную информацию о файловой системе в целом, и при его повреждении файловая система не может использоваться. Поскольку в файловой системе s5fs суперблок хранится в единственном варианте, вероятность возникновения ошибок достаточно велика.

Относительно низкая производительность связана с размещением компонентов файловой системы на диске. Метаданные файлов располагаются в начале файловой системы, а далее следуют блоки хранения данных. При работе с файлом, происходит обращение как к его метаданным, так и к дисковым блокам, содержащим его данные. Поскольку эти структуры данных могут быть значительно разнесены в дисковом пространстве, необходимость постоянного перемещения головки диска увеличивает время доступа и, как следствие, уменьшает производительность файловой системы в целом. К этому же эффекту приводит фрагментация файловой системы, поскольку отдельные блоки файла оказываются разбросанными по всему разделу диска.

Использование дискового пространства также не оптимально. Для увеличения производительности файловой системы более предпочтительным является использование блоков больших размеров. Это позволяет считывать большее количество данных за одну операцию ввода/вывода. Так, например, в UNIX SVR2 размер блока составлял 512 байтов, а в SVR3 — уже 1024 байтов. Однако поскольку блок может использоваться только одним файлом, увеличение размера блока приводит к увеличению неиспользуемого дискового пространства за счет частичного заполнения последнего блока файла. В среднем для каждого файла теряется половина блока.

Массив inode имеет фиксированный размер, задаваемый при создании файловой системы. Этот размер накладывает ограничение на максимальное число файлов, которые могут существовать в файловой системе. Расположение границы между метаданными файлов и их данными (блоками хранения данных) может оказаться неоптимальным, приводящим либо к нехватке inode, если файловая система хранит файлы небольшого размера, либо к нехватке дисковых блоков для хранения файлов большого размера. Поскольку динамически изменить эту границу невозможно, всегда останется неиспользованное дисковое пространство либо в массиве inode, либо в блоках хранения данных.

Наконец, ограничения, накладываемые на длину имени файла (14 символов) и общее максимальное число inode (65 535), также являются слишком жесткими.

Все эти недостатки привели к разработке новой архитектуры файловой системы, которая появилась в версии 4.2BSD UNIX под названием Berkeley Fast File System, или FFS.

## Файловая система BSD UNIX

В версии 4.3BSD UNIX были внесены существенные улучшения в архитектуру файловой системы, повышающие как ее производительность, так и надежность. Новая файловая система получила название Berkeley Fast File System (FFS).

Файловая система FFS, обладая полной функциональностью системы s5fs, использует те же структуры данных ядра. Основные изменения затронули расположение файловой системы на диске, дисковые структуры данных и алгоритмы размещения свободных блоков.

Как и в случае файловой системы s5fs, суперблок содержит общее описание файловой системы и располагается в начале раздела. Однако в суперблоке не хранятся данные о свободном пространстве файловой системы, такие как массив свободных блоков и inode. Поэтому данные суперблока остаются неизменными на протяжении всего времени существования файловой системы. Поскольку данные суперблока жизненно важны для работы всей файловой системы, он дублируется для повышения надежности.

Организация файловой системы предусматривает логическое деление дискового раздела на одну или несколько *групп цилиндров* (cylinder group). Группа цилиндров представляет собой несколько последовательных дисковых цилиндров. Каждая группа цилиндров содержит управляющую информацию, включающую резервную копию суперблока, массив inode, данные о свободных блоках и итоговую информацию об использовании дисковых блоков в группе (рис. 4.4).

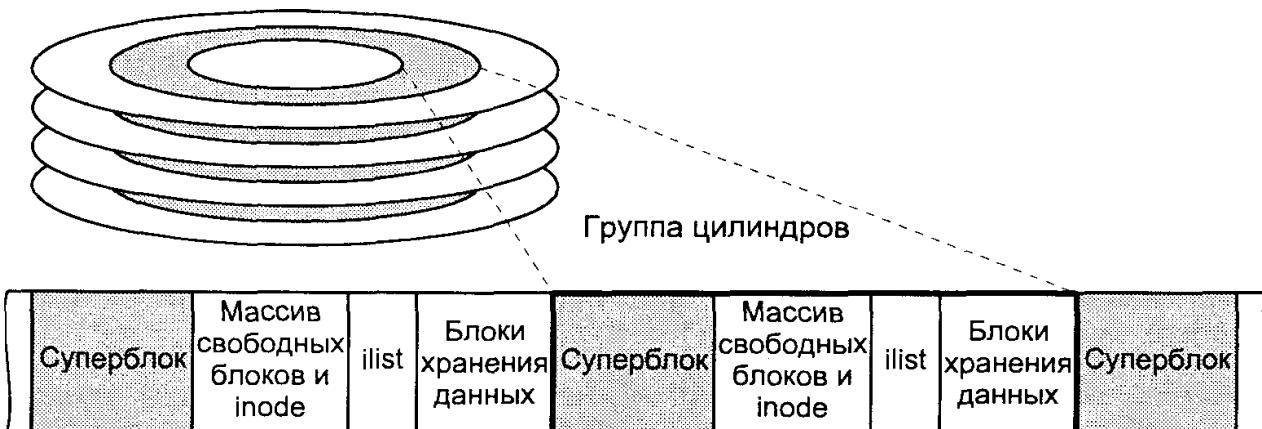


Рис. 4.4. Структура файловой системы FFS

Для каждой группы цилиндров при создании файловой системы выделяется место под определенное количество inode. При этом обычно на каждые 2 Кбайт блоков хранения данных создается один inode. Поскольку размеры группы цилиндров и массива inode фиксированы, в файловой системе *BSD UNIX* присутствуют ограничения, аналогичные *s5fs*.

Идея такой структуры файловой системы заключается в создании кластеров inode, распределенных по всему разделу, вместо того, чтобы группировать все inode в начале. Тем самым уменьшается время доступа к данным конкретного файла, поскольку блоки данных располагаются ближе к адресующем их inode. Такой подход также повышает надежность файловой системы, уменьшая вероятность потери всех индексных дескрипторов в результате сбоя.

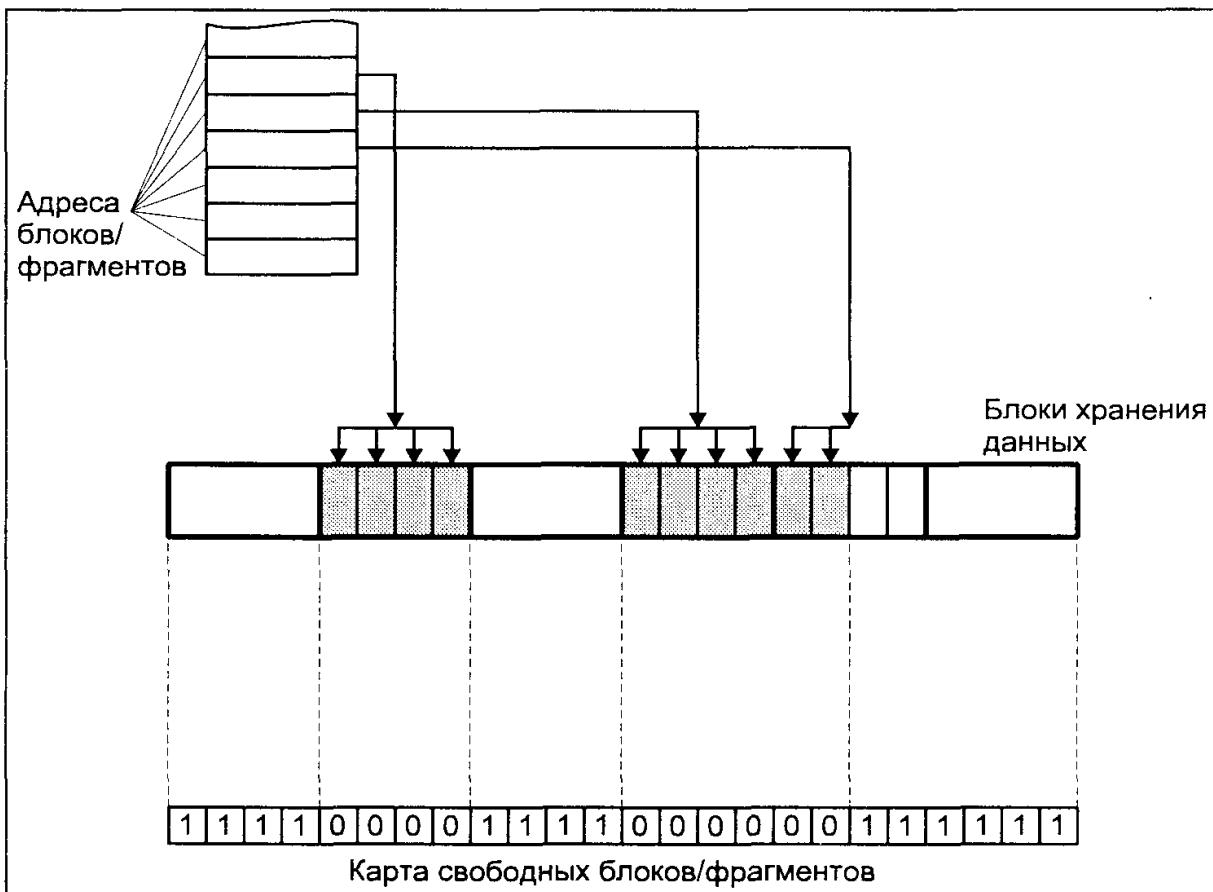
Управляющая информация располагается с различным смещением от начала группы цилиндров. В противном случае, например, при размещении в начале группы цилиндров, информация всех групп оказалась бы физически расположенной на одной пластине диска и могла бы быть уничтожена при выходе из строя этой пластины. Это смещение выбирается равным одному сектору относительно предыдущей группы, таким образом для соседних групп управляющая информация начинается на различных пластинах диска. В этом случае потеря одного сектора, цилиндра или пластины не приведет к потере всех копий суперблоков.

Производительность файловой системы существенным образом зависит от размера блока хранения данных. Чем больше размер блока, тем большее количество данных может быть прочитано без поиска и перемещения дисковой головки. Файловая система *FFS* поддерживает размер блока до 64 Кбайт. Проблема заключается в том, что типичная файловая система *UNIX* состоит из значительного числа файлов небольшого размера. Это приводит к тому, что частично занятые блоки используются неэффективно, что может привести к потере до 60% полезной емкости диска.

Этот недостаток был преодолен с помощью возможности фрагментации блока. Каждый блок может быть разбит на два, четыре или восемь фрагментов. В то время как блок является единицей передачи данных в операциях ввода/вывода, фрагмент определяет адресуемую единицу хранения данных на диске. Таким образом был найден компромисс между производительностью ввода/вывода и эффективностью хранения данных. Размер фрагмента задается при создании файловой системы, его максимальное значение определяется размером блока (0,5 размера блока), а минимальный — физическими ограничениями дискового устройства, а именно: минимальной единицей адресации диска — сектором.

Информация о свободном пространстве в группе хранится не в виде списка свободных блоков, а в виде *битовой карты блоков*. Карта блоков, связанная с определенной группой цилиндров, описывает свободное пространство в фрагментах, для определения того, свободен данный блок или

нет, ядро анализирует биты фрагментов, составляющих блок. На рис. 4.5 приведен пример карты свободных блоков и соответствия между битами карты, фрагментами и блоками группы цилиндров.



**Рис. 4.5.** Карта свободных блоков

Существенные изменения затронули алгоритмы размещения свободных блоков и inode, влияющие на расположение файлов на диске. В файловой системе s5fs используются весьма примитивные правила размещения. Свободные блоки и inode просто выбираются из конца соответствующего списка, что со временем приводит, как уже обсуждалось, к значительному разбросу данных файла по разделу диска.

В отличие от s5fs, файловая система FFS при размещении блоков использует стратегию, направленную на увеличение производительности. Некоторые из принципов приведены ниже:

- Файл по возможности размещается в блоках хранения данных, принадлежащих одной группе цилиндров, где расположены его метаданные. Поскольку многие операции файловой системы включают работу, связанную как с метаданными, так и с данными файла, это правило уменьшает время совершения таких операций.
- Все файлы каталога по возможности размещаются в одной группе цилиндров. Поскольку многие команды работают с несколькими

файлами одного и того же каталога, данный подход увеличивает скорость последовательного доступа к этим файлам.

- Каждый новый каталог по возможности помещается в группу цилиндров, отличную от группы родительского каталога. Таким образом достигается равномерное распределение данных по диску.
- Последовательные блоки размещаются исходя из оптимизации физического доступа. Дело в том, что существует определенный промежуток времени между моментом завершения чтения блока и началом чтения следующего. За это время диск успеет совершить оборот на некоторый угол. Таким образом, следующий блок должен по возможности располагаться с пропуском нескольких секторов. В этом случае при чтении последовательных блоков не потребуется совершать "холостые" обороты диска.

Таким образом, правила размещения свободных блоков, с одной стороны, направлены на уменьшение времени перемещения головки диска, т. е. на локализацию данных в одной группе цилиндров, а с другой — на равномерное распределение данных по диску. От разумного баланса между этими двумя механизмами зависит, в конечном итоге, производительность файловой системы. Например в предельном варианте, когда все данные локализованы в одной большой группе цилиндров, мы получаем типичную файловую систему *s5fs*.

Описанная архитектура является весьма эффективной с точки зрения надежности и производительности. К сожалению, эти параметры файловой системы *FSS* начинают значительно ухудшаться по мере уменьшения свободного места. В этом случае системе не удается следовать вышеприведенным правилам и размещение блоков далеко от оптимального. Практика показывает, что *FSS* имеет удовлетворительные характеристики при наличии более 10% свободного места.

## Каталоги

Структура каталога файловой системы *FFS* была изменена для поддержки длинных имен файлов (до 255 символов). Вместо записей фиксированной длины запись каталога *FFS* представлена структурой, имеющей следующие поля:

<i>d_ino</i>	Номер <i>inode</i> (индекс в массив <i>ilist</i> )
<i>d_reclen</i>	Длина записи
<i>d_namlen</i>	Длина имени файла
<i>d_name[]</i>	Имя файла

Имя файла имеет переменную длину, дополненную нулями до 4-байтной границы. При удалении имени файла принадлежавшая ему запись присоединяется к предыдущей, и значение поля *d\_reclen* увеличивается на со-

ответствующую величину. Удаление первой записи выражается в присвоении нулевого значения полю `d_ino`. Структура каталога файловой системы FFS приведена на рис. 4.6.

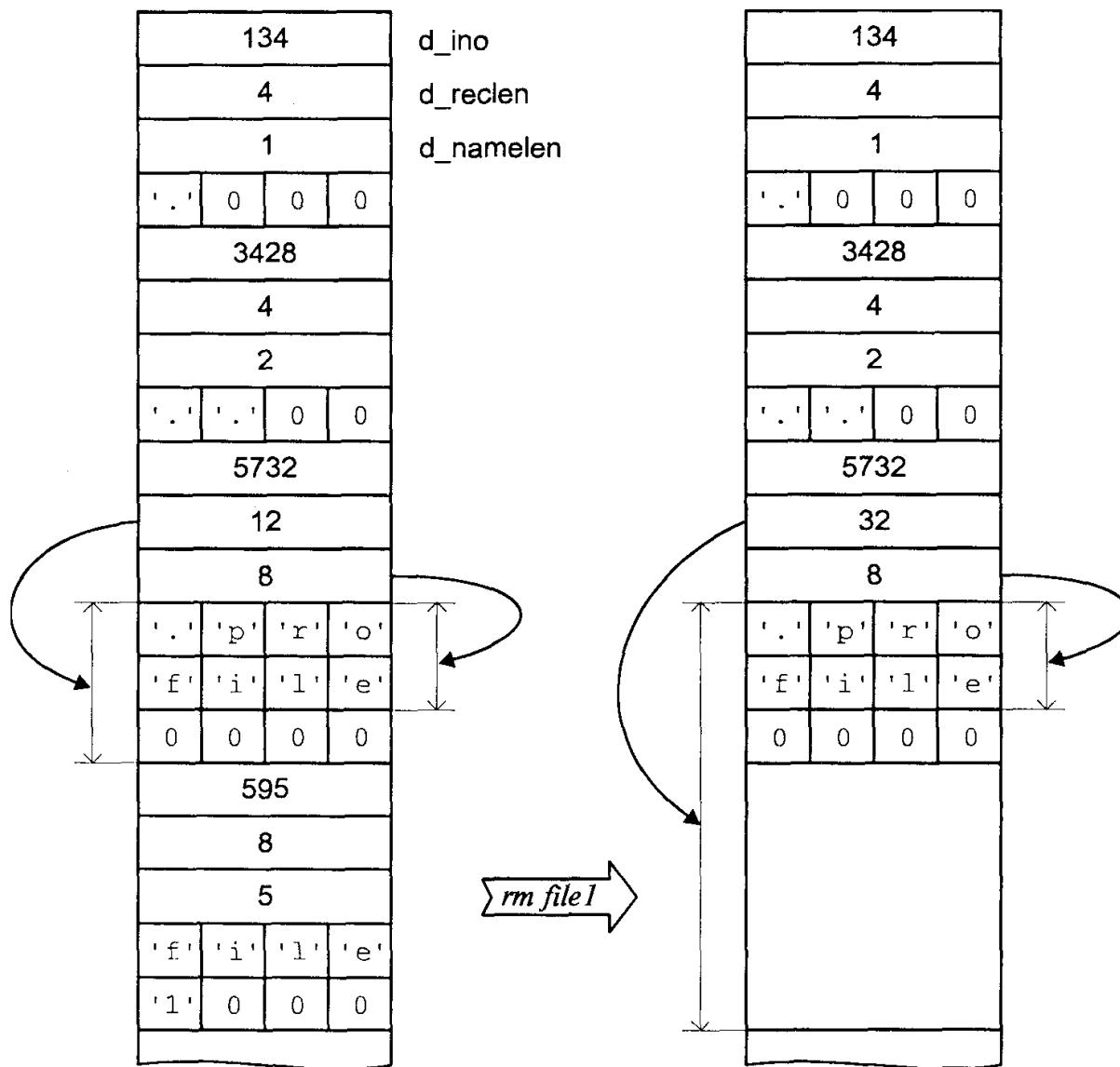


Рис. 4.6. Каталог файловой системы FFS

## Архитектура виртуальной файловой системы

Как было показано, различные типы файловых систем существенно отличаются по внутренней архитектуре. В то же время современные версии UNIX обеспечивают одновременную работу с несколькими типами файловых систем. Среди них можно выделить локальные файловые системы различной архитектуры, удаленные и даже отличные от файловой системы UNIX, например DOS. Такое сосуществование обеспечивается путем разделения каждой файловой системы на *зависимый* и *независимый* от реали-

зации уровня, последний из которых является общим и представляет для остальных подсистем ядра некоторую абстрактную файловую систему. Независимый уровень также называется *виртуальной файловой системой* (рис. 4.7). При этом дополнительные файловые системы различных типов могут быть встроены в ядро UNIX подобно тому, как это происходит с драйверами устройств.

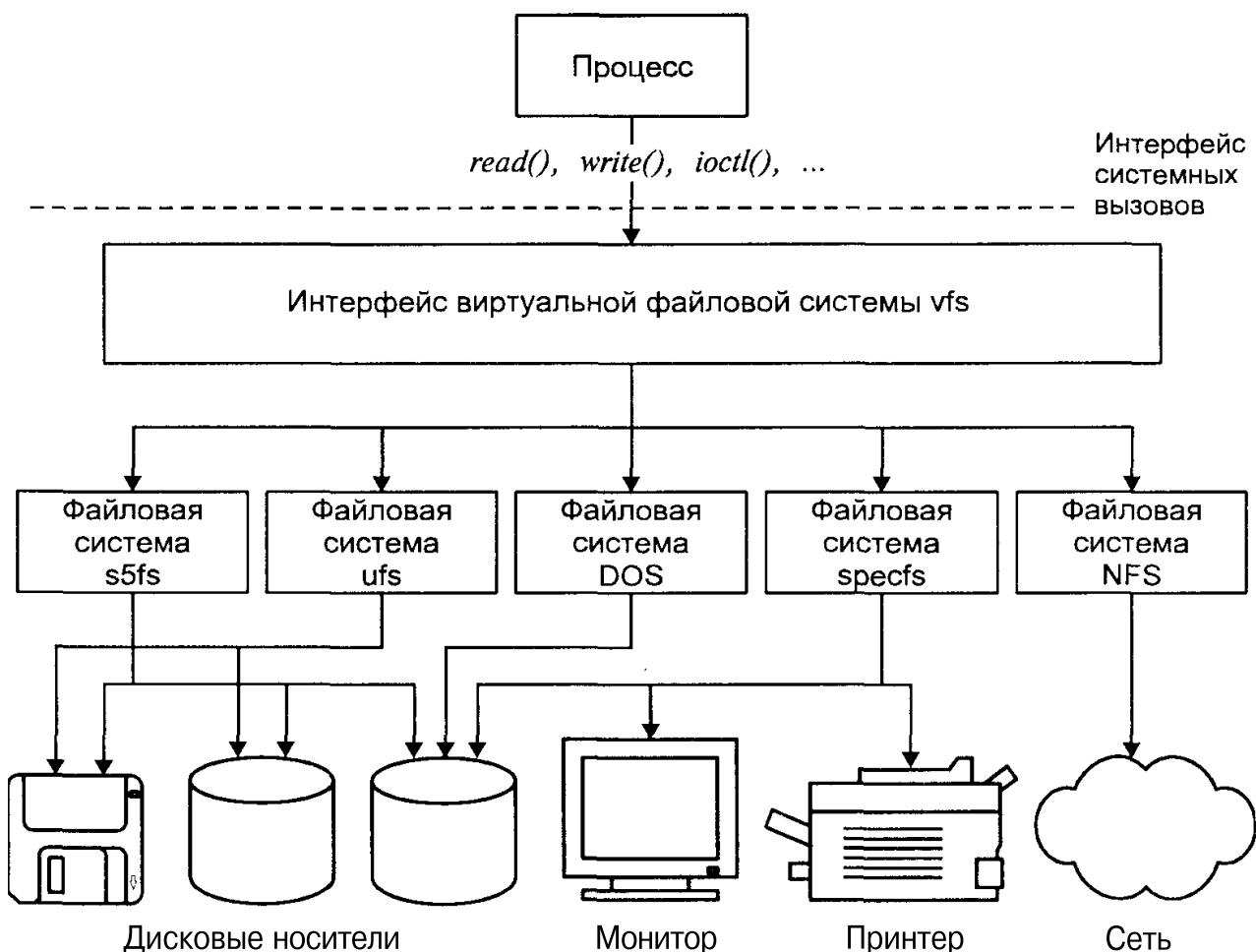


Рис. 4.7. Архитектура виртуальной файловой системы

## Виртуальные индексные дескрипторы

Дисковый файл обычно имеет связанную с ним структуру данных, называемую метаданными или inode, где хранятся основные характеристики данного файла и с помощью которой обеспечивается доступ к его данным. Одним из исключений из этого правила является файловая система DOS, в которой структуры файла и его метаданных существенно отличаются от принятых в UNIX. Тем не менее виртуальная файловая система основана на представлении метаданных файла в виде, сходном с традиционной семантикой UNIX. Интерфейсом работы с файлами является vnode (от virtual inode — виртуальный индексный дескриптор).

Первоначально этот интерфейс был разработан в 1984 году фирмой Sun Microsystems для обеспечения требуемой унификации работы с файловыми системами различных типов, в частности, с NFS и ufs (FFS). Сегодня виртуальная файловая система является стандартом в SVR4, хотя ряд других версий UNIX также реализуют подобную архитектуру (например, независимая файловая система SCO UNIX).

Метаданные всех активных файлов (файлов, на которые ссылаются один или более процессов) представлены в памяти в виде *in-core inode*, в качестве которых в виртуальной файловой системе выступают *vnode*. Структура данных *vnode* одинакова для всех файлов, независимо от типа реальной файловой системы, где фактически располагается файл. Данные *vnode* содержат информацию, необходимую для работы виртуальной файловой системы, а также неизменные характеристики файла, например, такие как тип файла.

Основные поля *vnode* приведены в табл. 4.1.

**Таблица 4.1.** Поля *vnode*

Поле		Описание
u short	vflag	Флаги <i>vnode</i>
u short	v_count	Число ссылок на <i>vnode</i>
struct filock	*v_filocks	Блокировки файла
struct vfs	*v_vfsmountedhere	Указатель на подключенную файловую систему, если <i>vnode</i> является точкой монтирования
struct vfs	*v_vfsp	Указатель на файловую систему, в которой находится файл
enum vtype	v_type	Тип <i>vnode</i> : обычный файл, каталог, специальный файл устройства, символическая связь, сокет
caddr_t	v_data	Указатель на данные, относящиеся к реальной файловой системе
struct vnodeops	*v_op	Операции <i>vnode</i>

Каждый *vnode* содержит число ссылок *v\_count*, которое увеличивается при открытии процессом файла и уменьшается при его закрытии. Когда число ссылок становится равным нулю, вызывается операция *vn\_inactive()*, которая сообщает реальной файловой системе, что на *vnode* никто больше не ссылается. После этого файловая система может освободить *vnode* (и, например, соответствующий ему *inode*) или поместить его в кэш для дальнейшего использования.

Поле *v\_vfsp* указывает на файловую систему (структуре *vfs*, о которой мы поговорим в следующем разделе), в которой расположен файл, адресо-

ванный данным vnode. Если vnode является точкой монтирования, то поле `v_vfsmountedhere` указывает на подключенную файловую систему, "перекрывающую" данный vnode.

Поле `v_data` указывает на данные, относящиеся к конкретной реализации реальной файловой системы. Например, для дисковой файловой системы `ufs`, `v_data` указывает на запись в таблице in-core inode.

Набор операций над vnode указан полем `v_op`. В терминах объектно-ориентированного программирования этот набор представляет собой виртуальные методы класса vnode. Он является своего рода шлюзом к реальной файловой системе, позволяя предоставить общий интерфейс виртуальной файловой системы и в то же время обеспечить специфические реализации функций работы с файлами, необходимые для различных типов файловых систем. Некоторые операции, большинство из которых уже знакомы читателю по системным вызовам, приведены в табл. 4.2.

**Таблица 4.2.** Операции с vnode виртуальной файловой системы

<code>int (*vn_open) ()</code>	Открыть vnode. Если операция предусматривает создание клона (размножение), то в результате будет размещен новый vnode. Обычно операции такого типа характерны для специальных файлов устройств.
<code>int (*vn_close) ()</code>	Закрыть vnode.
<code>int (*vn_read) ()</code>	Чтение данных файла, адресованного vnode.
<code>int (*vn_write) ()</code>	Запись в файл, адресованный vnode.
<code>int (*vn_ioctl) ()</code>	Задание управляющей команды.
<code>int (*vn_getaddr) ()</code>	Получить атрибуты vnode: тип vnode, права доступа, владелец-пользователь, владелец-группа, идентификатор файловой системы, номер inode, число связей, размер файла, оптимальный размер блока для операций ввода/вывода, время последнего доступа, время последней модификации, время последней модификации vnode, число занимаемых блоков.
<code>int (*vn_setaddr) ()</code>	Установить атрибуты vnode. Могут быть изменены UID, GID, размер файла и времена доступа и модификации.
<code>int (*vn</code>	Проверить права доступа к файлу, адресованному vnode. При этом производится отображение между атрибутами доступа файлов UNIX и атрибутами реальной файловой системы (например, DOS).
<code>int (*vn_lookup) ()</code>	Произвести трансляцию имени файла в соответствующий ему vnode.
<code>int (*vn_create) ()</code>	Создать новый файл и соответствующий ему vnode.
<code>int (*vn</code>	Удалить имя файла в указанном vnode каталоге.

Таблица 4.2 (продолжение)

<b>int</b>	<b>(*vn_link)()</b>	Создать жесткую связь между именем файла и vnode.
<b>int</b>	<b>(*vn_mkdir)()</b>	Создать новый каталог в указанном vnode каталоге.
<b>int</b>	<b>(*vn_rmdir)()</b>	Удалить каталог.
<b>int</b>	<b>(*vn_readdir)()</b>	Считать записи каталога, адресованного vnode.
<b>int</b>	<b>(*vn_symlink)()</b>	Создать символическую связь между новым именем и именем файла, расположенным в указанном vnode каталоге.
<b>int</b>	<b>(*vn_readlink)()</b>	Чтение файла — символической связи.
<b>int</b>	<b>(*vn_fsync)()</b>	Синхронизировать содержимое файла — записать все кэшированные данные.
<b>int</b>	<b>(*vn_inactive)()</b>	Разрешить удаление vnode, т. к. число ссылок на vnode из виртуальной файловой системы стало равным нулю.

Взаимосвязь между независимыми дескрипторами (vnode) и зависимыми от реализации метаданными файла показана на рис. 4.8.

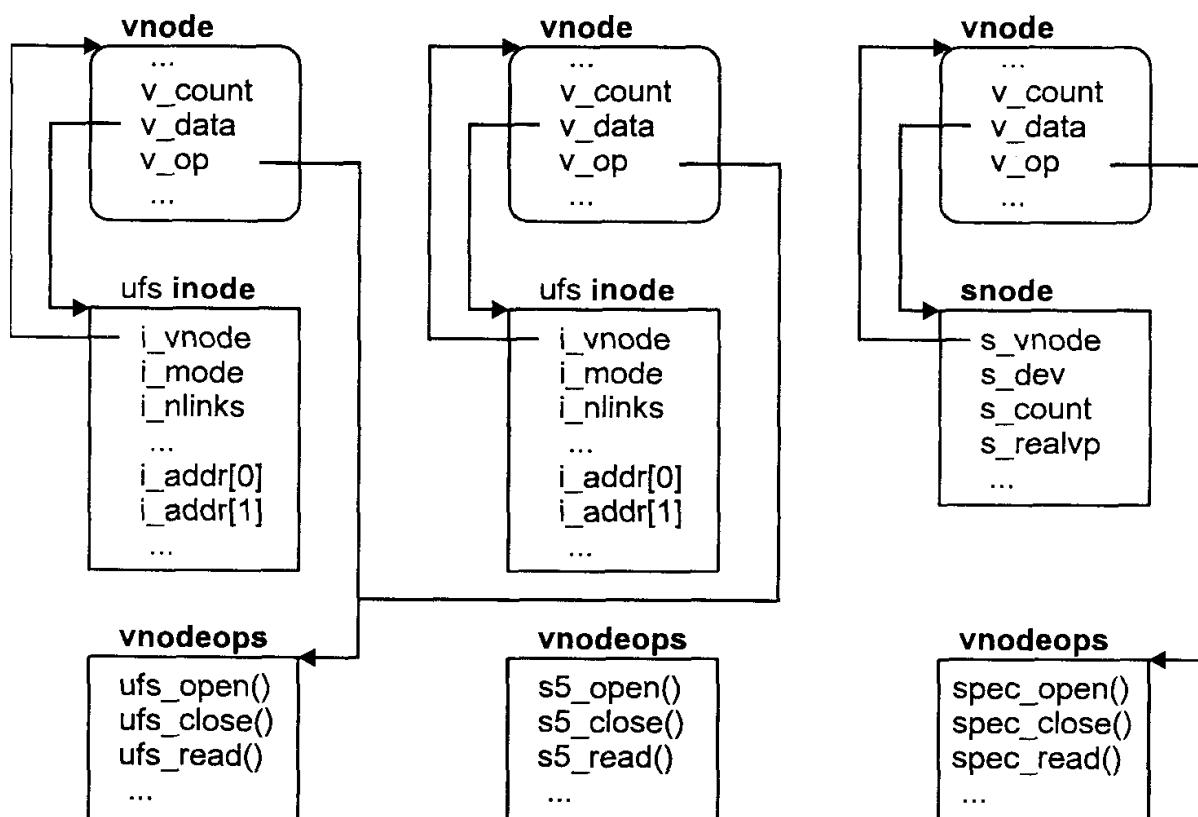


Рис. 4.8. Метаданные файла виртуальной файловой системы

## Монтирование файловой системы

Прежде чем может состояться работа с файлами, соответствующая файловая система должна быть встроена в существующее иерархическое дерево.

Только после этого ядро сможет выполнять файловые операции, такие как создание, открытие, чтение или запись в файл. Эта операция встраивания получила название *подключения* или *монтирования файловой системы*.

Каждая подключенная файловая система представлена на независимом уровне в виде структуры `vfs`, аналоге записи таблицы монтирования дисковой файловой системы. Структуры `vfs` всех подключенных файловых систем организованы в виде односвязного списка, в совокупности обеспечивая информацию, необходимую для обслуживания всего иерархического дерева, а также информацию о реальной файловой системе, которые не изменяются на протяжении работы. Первой записью списка всегда является корневая файловая система. В дальнейшем, список `vfs` мы будем называть устоявшимся термином — таблица монтирования. Поля структуры `vfs` приведены в табл. 4.3.

**Таблица 4.3.** Поля структуры `vfs`

<code>struct vfs</code>	<code>*vfs next</code>	Следующая файловая система в списке монтирования.
<code>struct vfsops</code>	<code>*vfs_op</code>	Операции файловой системы.
<code>struct vnode</code>	<code>*vfs vnodecovered</code>	<code>vnode</code> , перекрываемый файловой системой.
<code>int</code>	<code>vfs_flag</code>	Флаги: только для чтения, запрещен бит SUID и т. д.
<code>int</code>	<code>vfs_bsize</code>	Размер блока файловой системы.
<code>caddr_t</code>	<code>vfs_data</code>	Указатель на специфические данные, относящиеся к реальной файловой системе.

Поле `vfs_data` содержит указатель на данные реальной файловой системы. Например, для дисковой файловой системы `s5fs`, это поле указывает на суперблок, размещенный в памяти.

Поле `vfs_op` указывает на операции файловой системы, которые в терминах объектно-ориентированного подхода могут быть названы виртуальными методами объекта `vfs`. Возможные операции файловой системы приведены в табл. 4.4. Поскольку они существенным образом зависят от архитектуры и конкретной реализации, поля `vfs_op` заполняются указателями на соответствующие функции реальной файловой системы при ее монтировании.

**Таблица 4.4.** Операции файловой системы

<code>int (*vfs mount)</code>	Подключает файловую систему. Обычно операция включает размещение суперблока в памяти и инициализацию записи в таблице монтирования.
<code>int (*vfs unmount)()</code>	Отключает файловую систему. Операция включает актуализацию данных файловой системы на накопителе (например, синхронизацию дискового суперблока и его образа в памяти).

Таблица 4.4 (продолжение)

<b>int</b>	<b>(*vfs_root) ()</b>	Возвращает корневой vnode файловой системы.
<b>int</b>	<b>(*vfs_statfs) ()</b>	Возвращает общую информацию о файловой системе, в частности: размер блока хранения данных, число блоков, число свободных блоков, число inode.
<b>int</b>	<b>(*vfs_sync) ()</b>	Актуализирует все кэшированные данные файловой системы.
<b>int</b>	<b>(*vfs_fid) ()</b>	Возвращает <i>файловый идентификатор</i> (fid — file Identifier), однозначно адресующий файл в данной файловой системе. В качестве fid может, например, выступать номер inode реальной файловой системы.
<b>int</b>	<b>(*vfs_vget) ()</b>	Возвращает указатель на vnode для файла данной файловой системы, адресованного fid.

Для инициализации и монтирования реальной файловой системы UNIX хранит *коммутатор файловых систем* (File System Switch), адресующий процедурный интерфейс для каждого типа файловой системы, поддерживаемой ядром. UNIX System V для этого использует глобальную таблицу, каждый элемент которой соответствует определенному типу реальной файловой системы, например s5fs, ufs или nfs. Элемент этой таблицы vfssw имеет поля, указанные в табл. 4.5.

Таблица 4.5. Коммутатор файловых систем

char	<b>*vsw name</b>	Имя типа файловой системы
<b>int</b>	<b>(*vsw init) ()</b>	Адрес процедуры инициализации
struct vfsops	<b>*vsw vfsops</b>	Указатель на вектор операций файловой системы
long	<b>vsw flag</b>	Флаги

Взаимодействие структур виртуальной файловой системы показано на рис. 4.9.

Монтирование файловой системы производится системным вызовом *mount(2)*. В качестве аргументов передаются тип монтируемой файловой системы, имя каталога, к которому подключается файловая система (*точка монтирования*), флаги (например, доступ к файловой системе только для чтения) и дополнительные данные, конкретный вид и содержимое которых зависят от реализации реальной файловой системы. При этом производится поиск vnode, соответствующего файлу — точке монтирования (операция *lookup()* или *namei()* трансляции имени), и проверяется, что файл является каталогом и не используется в настоящее время для монтирования других файловых систем.

Затем происходит поиск элемента коммутатора файловых систем *vfssw[]*, соответствующего типу монтируемой файловой системы. Если такой эле-

мент найден, вызывается операция инициализации, адресованная полем `vsw_init()`. При этом выполняется размещение специфических для данного типа файловой системы данных, после чего ядро размещает структуру `vfs` и помещает ее в связанный список подключенных файловых систем, как это показано на рис. 4.11. Поле `vfs_vnodecovered` указывает на `vnode` точки монтирования. Это поле устанавливается нулевым для корневой (root) файловой системы, элемент `vfs` которой всегда расположен первым в списке подключенных файловых систем. Поле `vfs_op` адресует вектор операций, определенный для данного типа файловой системы. Наконец, указатель на данный элемент `vfs` сохраняется в поле `v_vfsmountedhere` виртуального индексного дескриптора каталога — точкамонтирования.

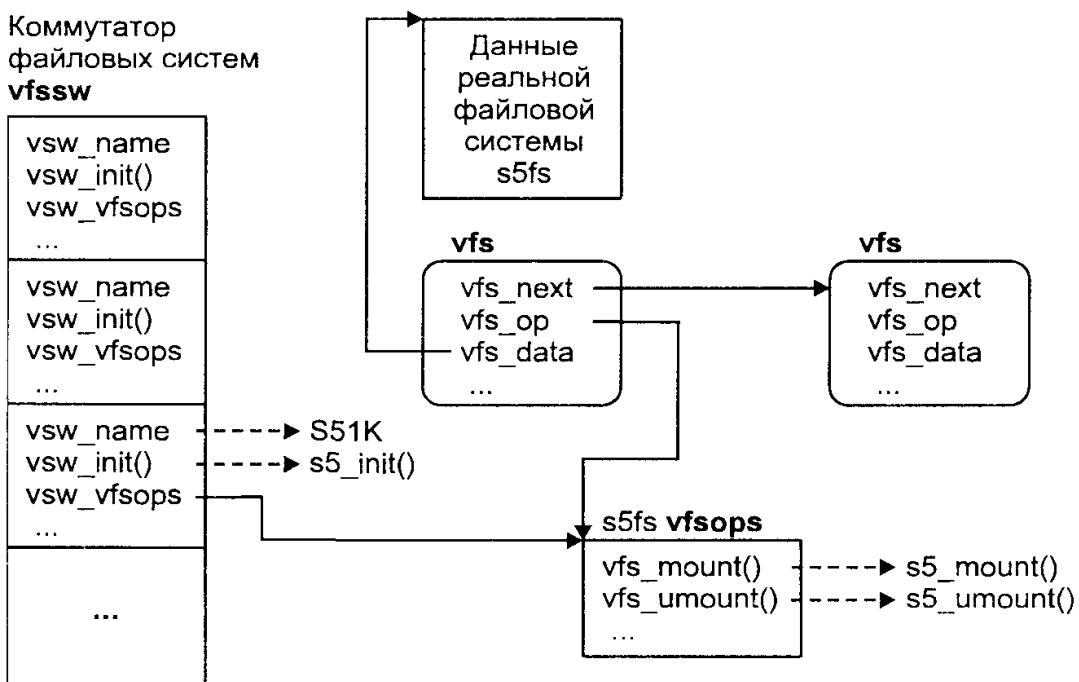


Рис. 4.9. Структуры данных виртуальной файловой системы

После этого вызывается операция `vfs_mount()`, соответствующая данному типу файловой системы. Конкретные действия определяются реализацией файловой системы и могут существенно различаться. Например, операция монтирования локальной файловой системы `ufs` предусматривает считывание в память метаданных системы, таких как суперблок, в то время как монтирование удаленной `NFS` файловой системы включает передачу сетевого запроса файловому серверу. Однако монтирование предусматривает выполнение и ряда общих операций, включающих:

- проверку соответствующих прав на выполнение монтирования;
- размещение и инициализацию специфических для файловой системы данного типа данных, сохранение адреса этих данных в поле `vfs_data` элемента `vfs`;

- размещение vnode для корневого каталога подключаемой файловой системы, доступ к которому осуществляется с помощью операции `vfs_root()`.

После подключения файловая система может быть адресована по имени точки монтирования. В частности, при отключении файловой системы с помощью системного вызова `umount(2)`, в качестве аргумента ему передается имя точки монтирования. Адресация с помощью специального файла устройства, как это происходило раньше, нарушает унифицированный вид виртуальной файловой системы, так как некоторые типы вообще не имеют такого устройства (например, NFS).

Определение корневого vnode для подключенной файловой системы производится с помощью операции `vfs_root()`. Заметим, что в некоторых реализациях независимой файловой системы (например, в SCO UNIX, хотя там используется другая терминология) одно из полей записи таблицы монтирования явно указывало на корневой vnode. Подход, предложенный фирмой Sun Microsystems, позволяет не хранить корневой vnode постоянно, размещая его только при необходимости работы с файловой системой. Это минимизирует ресурсы, занимаемые подключенными файловыми системами, которые продолжительное время не используются.

На рис. 4.10 приведен вид логического файлового дерева до и после монтирования файловой системы A к каталогу `/usr/local`. На рис. 4.11 приведен вид виртуальной файловой системы после этой операции монтирования.

Исследовать описанные структуры данных можно с помощью утилиты `crash(1M)`. Для этого применяются команды `vfs` и `vnode`, отображающие содержимое соответствующих структур данных. Приведем пример такого исследования файлового дерева операционной системы Solaris 2.5:

```
# crash
dumpfile = /dev/mem, namelist = /dev/ksyms, outfile = stdout
> !mount
/ on /dev/dsk/c0t3d0s0 read/write on Tue Feb 25 15:29:11 1997
/usr/local on /dev/dsk/c0t0d0s0 read/write on Tue Feb 25 15:29:13 1997
/tmp on swap read/write on Tue Feb 25 15:29:13 1997
/dev/fd on fd read/write/setuid on Tue Feb 25 15:29:11 1997
/proc on /proc read/write/setuid on Tue Feb 25 15:29:11 1997
/cdrom/unnamed_cdrom on /dev/dsk/c0t6d0 ronly on Mon Mar 25 15:29:43 1997
> vfs
  FSTYP  BSZ    MAJ/MIN      FSID  VNCOVERED      PDATA  BCOUNT  FLAGS
    ufs   8192    32,24    800018          0  f5b79b78      0  notr
    ufs   8192    32,0     800000  f5c29ado  f5c28c88      0  notr
  tmpfs  4096     0,0      0  f5958d18  f5d16ee0      0  notr
    fd   1024    158,0   2780000  f5c4f5d8      0      0
   proc  1024    156,0   2700000  f5c4f718      0  283920
   hsfs  2048    91,1  b9d02de5  f5f20698  f5b60d98      0      rd
```

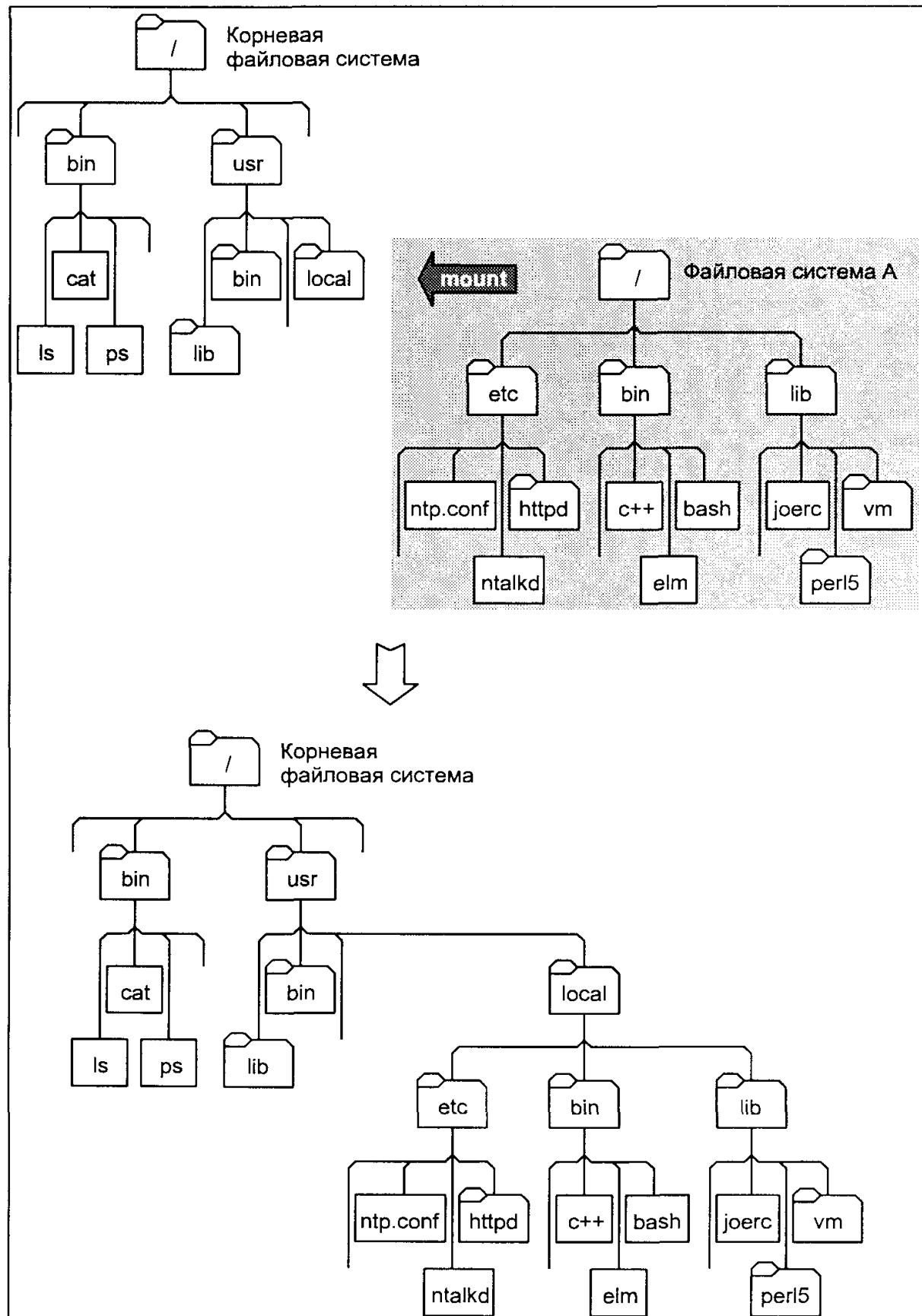


Рис. 4.10. Монтируемая файловой системы А к корневой файловой системе

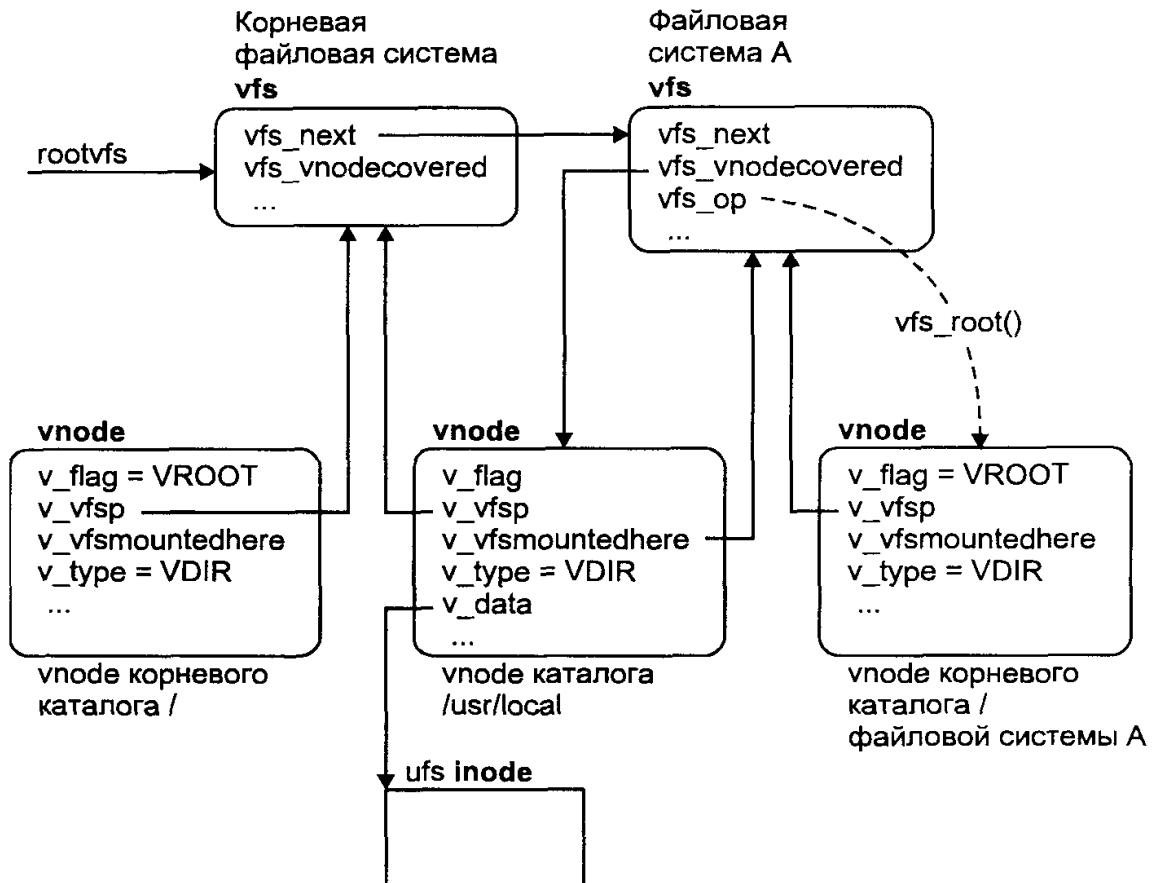


Рис. 4.11. Схема монтирования файловых систем различных типов

Мы распечатали список подключенных файловых систем (команда `mount(1M)`) и элементы `vfs` таблицы монтирования. Рассмотрим подробнее `vnode` точки монтирования файловой системы раздела `/dev/dsk/c0t0d0s0`.

**> vnode f5c29ad0**

```
VCNT VFSMNTED VFSP  STREAMP VTYPE  RDEV  VDATA  VFILEOCKS VFLAG
 2   f5c25c60  f0286570  0      d      -   f5c29ac8  0
```

Удостоверимся, что поле `v_vfsmountedhere` (VFSMNTED) адресует элемент `vfs` подключенной файловой системы, а поле `v_fsp` (VFSP) указывает на элемент корневой файловой системы.

**> vfs f5c25c60**

```
FSTYP BSZ MAJ/MIN   FSID  VNCOVERED  PDATA  BCOUNT  FLAGS
ufs 8192 32,0     800000  f5c29ad0  f5c28c88    0  notr
```

**> vfs f0286570**

```
FSTYP BSZ MAJ/MIN   FSID  VNCOVERED  PDATA  BCOUNT  FLAGS
ufs 8192 32,24    800018    0      f5b79b78    0  notr
```

Наконец, посмотрим на содержимое `inode` файловой системы `ufs`, адресованного полем `v_data` (VDATA) виртуального индексного дескриптора:

**> ui f5c29ac8**

```
UFS INODE TABLE SIZE = 1671
SLOT MAJ/MIN  INUMB RCNT LINK  UID  GID  SIZE  MODE  FLAGS
 32,24    7552    2   2    0    0   512 d---755  rf
```

Полученная информация показывает, что запись таблицы inode ufs адресует дисковый индексный дескриптор с номером 7552 (INUMB). Для того чтобы узнать имя файла, используем команду *ncheck(1M)*:

```
> !ncheck -i 7552
/dev/dsk/c0t3d0s0:
7552  /usr/local
```

## Трансляция имен

Прикладные процессы, запрашивая услуги файловой системы, обычно имеют дело с именем файла или файловым дескриптором, полученным в результате определенных системных вызовов. Однако ядро системы для обеспечения работы с файлами использует не имена, а индексные дескрипторы. Таким образом, необходима трансляция имени файла, передаваемого, например, в качестве аргумента системному вызову *open(2)*, в номер соответствующего vnode.

В табл. 4.6 приведены системные вызовы, для выполнения которых требуется трансляция имени файла.

**Таблица 4.6.** Системные вызовы, требующие трансляции имени

<i>exec(2)</i>	Запустить программу на выполнение
<i>chown(2)</i>	Изменить владельца-пользователя
<i>chgrp(2)</i>	Изменить владельца-группу
<i>chmod(2)</i>	Изменить права доступа
<i>statfs(2)</i>	Получить метаданные файла
<i>rmdir(2)</i>	Удалить каталог
<i>mkdir(2)</i>	Создать каталог
<i>mknode(2)</i>	Создать специальный файл устройства
<i>open(2)</i>	Открыть файл
<i>link(2)</i>	Создать жесткую связь

Говоря формально, полное имя файла представляет собой последовательность слов, разделенных символом '/'. Каждый компонент имени, кроме последнего, является именем каталога. Последний компонент определяет собственно имя файла. При этом полное имя может быть *абсолютным* или *относительным*. Если полное имя начинается с символа '/', представляющего корневой каталог общего логического дерева файловой системы, то оно является абсолютным, однозначно определяющим файл из любого места файловой системы. В противном случае, имя является относительным и адресует файл относительно текущего каталога. Примером относительного имени может служить **include/sys/user.h**, а абсолютное имя этого файла — **/usr/include/sys/user.h**. Как следует из этих рассуждений, два ка-

толога играют ключевую роль при трансляции имени: корневой каталог и текущий каталог. Каждый процесс адресует эти каталоги двумя полями структуры *u\_area*:

<code>struct vnode *u_cdir</code>	<b>Указатель на vnode текущего каталога</b>
<code>struct vnode *u_rdir</code>	<b>Указатель на vnode корневого каталога</b>

В зависимости от имени файла трансляция начинается с *vnode*, адресованного либо полем *u\_cdir*, либо *u\_rdir*. Трансляция имени осуществляется покомпонентно, при этом для *vnode* текущего каталога вызывается соответствующая ему операция *vn\_lookup()*, в качестве аргумента которой передается имя следующего компонента. В результате операции возвращается *vnode*, соответствующий искомому компоненту.

Если для *vnode* каталога установлен указатель *vn\_vfsmountedhere*, то данный каталог является точкой монтирования. Если имя файла требует дальнейшего спуска по дереву файловой системы (т. е. пересечения точки монтирования), то операция *vn\_lookup()* следует указателю *vn\_vfsmountedhere* для перехода в подключенную файловую систему и вызывает для нее операцию *vfs\_root()* для получения ее корневого *vnode*. Трансляция имени затем продолжается с этого места.

Пересечение границы файловых систем возможно и при восхождении по дереву, например, если имя файла задано указанием родительского каталога — *../../myfile.txt*. Если при движении в этом направлении по пути встречается корневой *vnode* подключенной файловой системы (установлен флаг *VR0OT* в поле *v\_flag*), то операция *vn\_lookup()* следует указателю *vfs\_vnodecovered*, расположенному в записи *vfs* этой файловой системы. При этом происходит пересечение границы файловых систем, и дальнейшая трансляция продолжается с точки монтирования.

Если искомый файл является символической связью, и системный вызов, от имени которого происходит трансляция имени, "следует" символической связи, операция *vn\_lookup()* вызывает *vn\_readlink()* для получения имени целевого файла. Если оно является абсолютным (т. е. начинается с *"/"*), то трансляция начинается с *vnode* корневого каталога, адресованного полем *u\_rdir* области *u-area*.

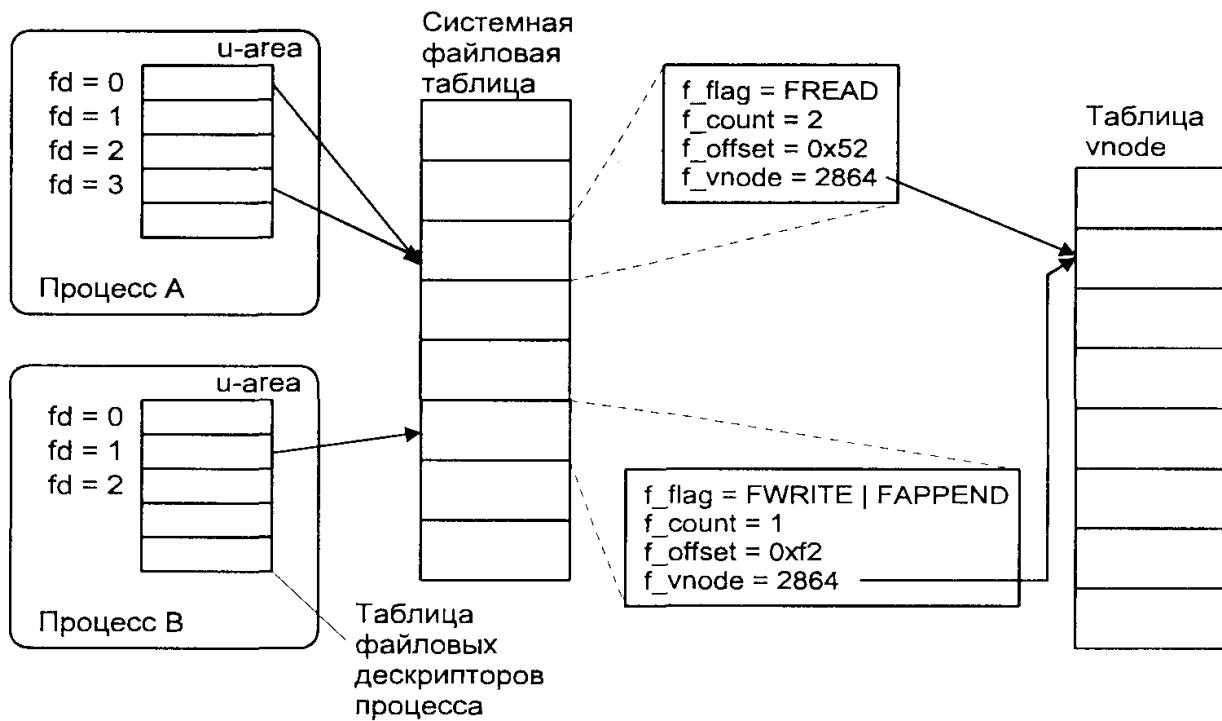
Процесс трансляции имени продолжается, пока не просмотрены все компоненты имени или не обнаружена ошибка (например, отсутствие прав доступа). В случае удачного завершения возвращается *vnode* искомого файла.

## Доступ к файловой системе

Как было показано в главе 2, процесс совершает операции с файлами, адресуя их при помощи файловых дескрипторов — целых чисел, имеющих ло-

кальное для процесса значение. Это значит, что файловый дескриптор одного процесса может адресовать совершенно другой файл, нежели файловый дескриптор с таким же номером, используемый другим процессом. Процесс получает файловый дескриптор с помощью ряда системных вызовов, например, *open(2)* или *creat(2)*, выполняющих операцию трансляции имени, в результате которой выделяемый файловый дескриптор адресует определенный *inode* (или *vnode*) и, соответственно, файл файловой системы.

На рис. 4.12 показаны основные структуры ядра, необходимые для доступа процесса к файлу.



**Рис. 4.12.** Внутренние структуры доступа к файлу

Файловый дескриптор, используемый для доступа процесса к файлу, является индексом таблицы файловых дескрипторов (file descriptor table). Каждый процесс имеет собственную таблицу файловых дескрипторов, которая расположена в его *u-area*. На рис. 4.12 показаны два процесса, каждый из которых использует собственную таблицу файловых дескрипторов.

Каждая активная запись этой таблицы, представляющая открытый файл, адресует запись системной файловой таблицы (system file table), в которой хранятся такие параметры, как режим доступа к файлу (запись, чтение, добавление и т. д.), текущее смещение в файле (файловый указатель), а также указатель на *vnode* этого файла. Системная файловая таблица одна и совместно используется всеми процессами.

Как следует из рис. 4.12, несколько записей системной файловой таблицы могут адресовать один и тот же файл, который представлен единственной записью в таблице *vnode*.

## Файловые дескрипторы

Файловый дескриптор представляет собой неотрицательное целое число, возвращаемое системными вызовами, такими как *creat(2)*, *open(2)* или *pipe(2)*. После получения файлового дескриптора процесс может использовать его для дальнейшей работы с файлом, например с помощью системных вызовов *read(2)*, *write(2)*, *close(2)* или *fcntl(2)*.

Ядро обеспечивает работу процесса с файлами, используя различные структуры данных, часть из которых расположена в и-area процесса. Напомним, что эта область описывается структурой *user*. В табл. 4.7 приведены поля структуры *user*, которые используются ядром для обеспечения доступа процесса к файлу.

**Таблица 4.7.** Поля структуры *user*, связанные с файловым дескриптором

Поле	Описание
<i>u_ofile</i>	Указатель на системную файловую таблицу
<i>u_pofile</i>	Флаги файлового дескриптора

Файловый дескриптор связан с этими двумя полями и, таким образом, обеспечивает доступ к соответствующему элементу файловой таблицы (структуре данных *file*).

В настоящее время в качестве единственного флага файлового дескриптора определен флаг *FD\_CLOEXEC*. Если этот флаг установлен, то производится закрытие файлового дескриптора (аналогично явному вызову *close(2)*) при выполнении процессом системного вызова *exec(2)*). При этом для запущенной программы не происходит наследования файлового дескриптора и доступа к файлу.

Более старые версии UNIX используют статическую таблицу дескрипторов, которая целиком хранится в и-area. Номер дескриптора является индексом этой таблицы. Таким образом, размер таблицы, которая обычно содержит 64 элемента, накладывает ограничение на число одновременно открытых процессом файлов. В современных версиях таблица размещается динамически и может увеличиваться при необходимости. Следует, однако, иметь в виду, что и в этом случае максимальное число одновременно открытых файлов регламентируется пределом *RLIMIT\_NOFILE*, который рассматривался в разделе "Ограничения" главы 2. В некоторых версиях, например, Solaris 2.5, данные файловых дескрипторов хранятся не в виде таблицы, а в виде блоков структур *uf\_entry*, поля которой аналогичны приведенным в табл. 4.7.

Содержимое таблицы дескрипторов процесса можно посмотреть с помощью утилиты *crash(1M)*. Команда *user* покажет содержимое и-area процесса.

са. Например, для текущего командного интерпретатора мы получим следующую информацию:

```
# crash
dumpfile = /dev/mem, namelist = /dev/ksyms, outfile = stdout
> proc #8591
PROC TABLE SIZE = 1498
SLOT ST PID PPID PGID SID UID PRI NAME FLAGS
121 s 8591 8589 8591 8591 286 48 bash load jctl
> user 121
PER PROCESS USER AREA FOR PROCESS 121
PROCESS MISC:
  command: bash, psargs: -bash
  start: 00 Mon 24 18:11:31 1997
  mem: 1ebe, type: exec
  vnode of current directory: f5b95e40
OPEN FILES, POFILE FLAGS, AND THREAD REFCNT:
  [0]: F 0xf62b6030, 0, 0 [1]: F 0xf62b6030, 0, 0
  [2]: F 0xf62b6030, 0, 0
  cmask: 0022
RESOURCE LIMITS:
  cpu time: unlimited/unlimited
  file size: unlimited/unlimited
  swap size: 2147479552/2147479552
  stack size: 8388608/2147479552
  coredump size: unlimited/unlimited
  file descriptors: 64/1024
  address space: unlimited/unlimited
SIGNAL DISPOSITION:
```

## Файловая таблица

Поля файлового дескриптора `u_ofile` и `u_pofile` содержат начальную информацию, необходимую для доступа процесса к данным файла. Дополнительная информация находится в системной файловой таблице и таблице индексных дескрипторов. Для обеспечения доступа процесса к данным файла ядро должно полностью создать цепочку от файлового дескриптора до `vnode` и, соответственно, до блоков хранения данных, как показано на рис. 4.12.

Каждый элемент файловой таблицы содержит информацию, необходимую для управления работой с файлом. Если несколько процессов открывают один и тот же файл, каждый из них получает собственный элемент файловой таблицы, хотя все они будут работать с одним и тем же файлом. Важнейшие поля элемента файловой таблицы приведены ниже:

Поле	Описание
<code>f_flag</code>	Флаги, указанные при открытии файла (системные вызовы <code>open(2)</code> , <code>creat(2)</code> ). Каждая операция с файлом проверяется на допустимость согласно указанным режимам. Другими словами, если процесс открыл файл только для чтения (флаг <code>FREAD</code> ), ему будет отказано в операции записи, даже если он имеет на это необходимые права доступа.
<code>FREAD</code>	Файл открыт только для чтения. То же, что и <code>O_RDONLY</code> при открытии файла.
<code>FWRITE</code>	Файл открыт только на запись. То же, что и <code>O_WRONLY</code> при открытии файла.
<code>FAPPEND</code>	Режим добавления. Перед началом операции записи файловый указатель будет установлен в конец файла. То же, что и <code>O_APPEND</code> при открытии файла.
<code>FNONBLOCK</code> , <code>FNDELAY</code>	Возврат без блокирования. Системный вызов не будет ожидать завершения операции. То же, что и <code>O_NONBLOCK</code> или <code>O_NDELAY</code> при открытии файла.
<code>FSYNC</code>	Обеспечить синхронизацию с соответствующими дисковыми структурами для метаданных и данных файла при совершении операции записи. То же, что и <code>o_SYNC</code> при открытии файла.
<code>FDSYNC</code>	Обеспечить синхронизацию с соответствующими дисковыми структурами только для данных файла при совершении операции записи. То же, что и <code>O_DSYNC</code> при открытии файла.
<code>FRSYNC</code>	Совместно с флагами <code>FSYNC</code> и <code>FDSYNC</code> определяет процесс синхронизации для соответствующих компонентов файла при операции чтения.
<code>f_count</code>	Число файловых дескрипторов, адресующих данный элемент файловой таблицы. Один и тот же элемент файловой таблицы может совместно использоваться при дублировании дескрипторов с помощью системного вызова <code>dup(2)</code> или в результате <code>fork(2)</code> .
<code>f_vnode</code>	Указатель на виртуальный индексный дескриптор файла.
<code>f_offset</code>	Текущее смещение в файле. Начиная с этого места будет произведена следующая операция чтения или записи.

Для иллюстрации обсуждения продолжим работу с утилитой `crash(1M)`. С помощью команды `user` в предыдущем разделе были получены адреса элементов файловой таблицы для стандартного ввода (`fd=0`), вывода (`fd=1`) и вывода сообщений об ошибках (`fd=2`). Заметим, что все они указывают на один и тот же элемент. С помощью команды `file` исследуем его содержимое:

```
> file 0xf62b6030
ADDRESS  RCNT  TYPE/ADDR      OFFSET  FLAGS
f62b6030  9  SPEC/f5e91clc  15834  read write
> vnode f5e91clc
VCNT  VFSMNTED  VFSP  STREAMP  VTYPE  RDEV  VDATA  VFILOCKS  VFLAG
2      0        f0286570  f5c6b2a0  c      24,26  f5e91c18  0
```

Поскольку это специальный файл устройства (об этом свидетельствует поле TYPE элемента файловой таблицы), поле v\_data (VDATA) vnode указывает не на inode файловой системы ufs, а на snode — индексный дескриптор логической файловой системы specfs, обслуживающей специальные файлы устройств. Более подробно этот интерфейс будет рассматриваться в следующей главе. Таким образом, для продолжения путешествия по структурам данных ядра, следует обратиться к snode, адрес которого указан в поле VDATA.

```
> snode f5e91c18
SNODE TABLE SIZE = 256
HASH-SLOT MAJ/MIN  REALVP  COMMONVP NEXTR  SIZE  COUNT  FLAGS
      24,26    f5f992e8    f636b27c    O    O    O    up ac
```

Поле s\_realvp (REALVP) указывает на vnode файла реальной файловой системы (в данном случае ufs). Поэтому далее поиск аналогичен проделанному при исследовании таблицы монтирования.

```
> vnode f5f992e8
VCNT VFSMNTED VFSP  STREAMP VTYPE  RDEV    VDATA      VFLOCKS VFLAG
  2      0    f0286570    0      c    24,26    f5f992e0      0
> ui f5f992e0
UFS INODE TABLE SIZE = 1671
SLOT MAJ/MIN  INUMB RCNT LINK  UID  GID  SIZE  MODE    FLAGS
  32,24    317329    2    1    286    7    0    c---620    rf
> ! ncheck -i 317329
/dev/dsk/c0t3d0s0:
317329 /devices/pseudo/pts@0:26
```

В результате мы определили имя специального файла устройства (в данном случае — это псевдотерминал), на которое производится ввод и вывод командного интерпретатора.

## Блокирование доступа к файлу

Традиционно архитектура файловой подсистемы UNIX разрешает нескольким процессам одновременный доступ к файлу для чтения и записи. Хотя операции записи и чтения, осуществляемые с помощью системных вызовов *read(2)* или *write(2)*, являются атомарными, в UNIX по умолчанию отсутствует синхронизация между отдельными вызовами. Другими словами, между двумя последовательными вызовами *read(2)* одного процесса другой процесс может модифицировать данные файла. Это, в частности, может привести к несогласованным операциям с файлом, и как следствие,

ченная специально для управления блокированием. При этом перед фактической файловой операцией (чтения или записи) процесс устанавливает блокирование соответствующего типа (для чтения или для записи). Если блокирование завершилось успешно, это означает, что требуемая файловая операция не создаст конфликта или нарушения целостности данных, например, при одновременной записи в файл несколькими процессами.

По умолчанию блокирование является *рекомендательным* (advisory lock). Это означает, что кооперативно работающие процессы могут руководствоваться созданными блокировками, однако ядро не запрещает чтение или запись в заблокированный участок файла. При работе с рекомендательными блокировками процесс должен явно проверять их наличие с помощью тех же функций *fcntl(2)* и *lockf(3C)*.

Мы уже встречались с использованием системного вызова *fcntl(2)* для блокирования записей файла в главе 2. Там же была упомянута структура *flock*, служащая для описания блокирования. Поля этой структуры описаны в табл. 4.8.

**Таблица 4.8.** Поля структуры *flock*

Поле	Описание
short <i>l_type</i>	Тип блокирования: <i>F_RDLCK</i> обозначает блокирование для чтения (read lock), <i>F_WRLCK</i> — блокирование для записи (write lock), <i>F_UNLCK</i> обозначает снятие блокирования.
short <i>l_whence</i>	Точка отсчета смещения записи в файле. Может принимать значения, аналогичные рассмотренным при разговоре о функции <i>lseek(2)</i> в главе 2: <i>SEEK_SET</i> , <i>SEEK_CUR</i> , <i>SEEK_END</i> .
off_t <i>l_start</i>	Смещение блокируемой записи относительно точки отсчета, указанной полем <i>l_whence</i> .
off_t <i>l_len</i>	Длина блокируемой записи. Нулевое значение <i>l_len</i> указывает, что запись всегда распространяется до конца файла, независимо от возможного изменения его размера.
pid_t <i>l_pid</i>	Идентификатор процесса, установившего блокирование, возвращаемый при вызове команды <i>GETLK</i> .

Как следует из описания поля *l\_type* структуры *flock*, существуют два типа блокирования записи: для чтения (*F\_RDLCK*) и для записи (*F\_WRLCK*). Правила блокирования таковы, что может быть установлено несколько блокирований для чтения на конкретный байт файла, при этом в установке блокирования для записи на этот байт будет отказано. Напротив, блокирование для записи на конкретный байт должно быть единственным, при этом в установке блокирования для чтения будет отказано.

Приведем фрагмент программы, использующей возможность блокирования записей:

```
struct flock      lock;

/*Заполним описание lock с целью блокирования всего файла для записи*/
lock.l_type = F_WRLCK;
lock.l_start = 0;
lock.whence = SEEK_SET;
lock.len = 0;
/*Заблокируем файл. Если блокирования, препятствующие данной операции,
уже существуют — ждем их снятия*/
fcntl(fd, SETLKW, &lock);
/*Запишем данные в файл — нам никто не помешает*/
write(fd, record, sizeof(record));
/*Снимем блокирование*/
lock.l_type = F_UNLCK;
fcntl(fd, SETLKW, &lock);
```

В отличие от рекомендательного в UNIX существует *обязательное блокирование* (mandatory lock), при котором ограничение на доступ к записям файла накладывается самим ядром. Реализация обязательных блокировок может быть различной. Например, в SCO UNIX (SVR3) снятие бита x для группы и установка бита SGID для группы приводит к тому, что блокировки, установленные *fcntl(2)* или *lockf(3C)*, станут обязательными. UNIX SVR4 поддерживает установку блокирования отдельно для записи и для чтения, обеспечивая тем самым доступ для чтения многим, а для записи — только одному процессу. Эти установки также осуществляются с помощью системного вызова *fcntl(2)*. Следует иметь в виду, что использование обязательного блокирования таит потенциальную опасность. Например, если процесс блокирует доступ к жизненно важному системному файлу и по каким-либо причинам теряет контроль, это может привести к аварийному останову операционной системы.

## Буферный кэш

В введении отмечалось, что работа файловой подсистемы тесно связана с обменом данными с периферийными устройствами. Для обычных файлов и каталогов — это устройство, на котором размещается соответствующая файловая система, для специальных файлов устройств — это принтер, терминал, или сетевой адаптер. Не вдаваясь в подробности подсистемы ввода/вывода, рассмотрим, как во многих версиях UNIX организован обмен данными с дисковыми устройствами — традиционным местом хранения подавляющего большинства файлов<sup>3</sup>.

<sup>3</sup> На самом деле файловые системы могут располагаться на удаленных компьютерах (например, в случае NFS). Хотя при работе с такими файловыми системами дисковый ввод/вывод отсутствует, тем не менее и в этом случае кэширование блоков данных значительно повышает производительность.

Не секрет, что операции дискового ввода/вывода являются медленными по сравнению, например, с доступом к оперативной или сверхоперативной памяти. Время чтения данных с диска и копирования тех же данных в памяти может различаться в несколько тысяч раз. Поскольку основные данные хранятся на дисковых накопителях, дисковый ввод/вывод является узким местом операционной системы. Для повышения производительности дискового ввода/вывода и, соответственно, всей системы в целом, в UNIX используется кэширование дисковых блоков в памяти.

Для этого используется выделенная область оперативной памяти, где кэшируются дисковые блоки файлов, к которым наиболее часто осуществляется доступ. Эта область памяти и связанный с ней процедурный интерфейс носят название *буферного кэша*, и через него проходит большинство операций файлового ввода/вывода. Схема взаимодействия различных подсистем ядра с буферным кэшем приведена на рис. 4.13.

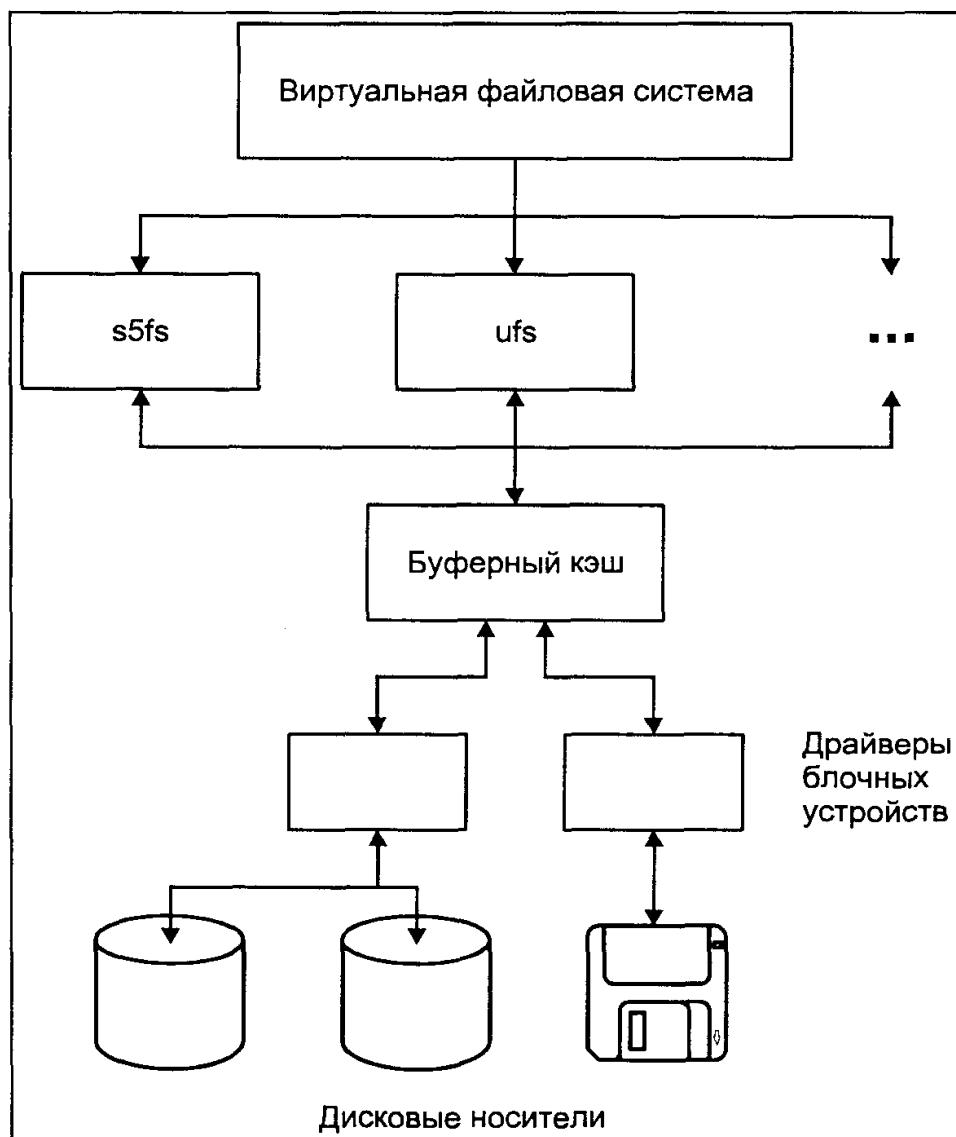


Рис. 4.13. Роль буферного кэша

## Внутренняя структура буферного кэша

Буферный кэш состоит из буферов данных, размер которых достаточен для размещения одного дискового блока. С каждым блоком данных связан заголовок *буфера*, представленный структурой *buf*, с помощью которого ядро производит управление кэшем, включая идентификацию и поиск буферов, а также синхронизацию доступа. Заголовок также используется при обмене данными с драйвером устройства для выполнения фактической операции ввода/вывода. Когда возникает необходимость чтения или записи буфера на диск, ядро заносит параметры операции ввода/вывода в заголовок и передает его функции драйвера устройства. После завершения операции ввода/вывода заголовок содержит информацию о ее результатах.

Основные поля структуры *buf* приведены в табл. 4.9.

**Таблица 4.9.** Поля структуры *buf*

Поле	Описание
<i>b_f_lags</i>	Флаги. Определяют состояние буфера в каждый момент времени (например, <i>B_BUSY</i> — буфер занят или <i>B_DONE</i> — закончена операция ввода/вывода с буфером) и направление передачи данных ( <i>B_READ</i> , <i>B_WRITE</i> , <i>B_PHYS</i> )
<i>av_f_orw</i> , <i>av_back</i>	Указатели двухсвязного рабочего списка буферов, ожидающих обработки драйвером
<i>b_bcount</i>	Число байтов, которое требуется передать
<i>b_un.b_addr</i>	Виртуальный адрес буфера
<i>b_blkno</i>	Номер блока начала данных на устройстве
<i>b_dev</i>	Старший и младший номера устройства

Поле *b\_flags* хранит различные флаги связанного с заголовком буфера. Часть флагов используется буферным кэшем, а часть — драйвером устройства. Например, с помощью флага *B\_BUSY* осуществляется синхронизация доступа к буферу. Флаг *B\_DELWR1* отмечает буфер как модифицированный, или "грязный", требующий сохранения на диске перед повторным использованием. Флаги *B\_READ*, *B\_WRITE*, *B\_ASYNC*, *B\_DONE* и *B\_ERROR* используются драйвером диска. Более подробно операция ввода/вывода для драйвера будет рассмотрена в следующей главе.

Буферный кэш использует механизм *отложенной записи* (*write-behind*), при котором модификация буфера не вызывает немедленной записи на диск. Такие буферы отмечаются как "грязные", а синхронизация их содержимого с дисковыми данными происходит через определенные промежутки времени. Примерно одна треть операций дискового ввода/вывода приходится на запись, причем один и тот же буфер может на протяжении ограниченного промежутка времени модифицироваться несколько раз. Поэтому буферный кэш позволяет значительно уменьшить интенсивность записи на

диск<sup>4</sup> и реорганизовать последовательность записи отдельных буферов для повышения производительности ввода/вывода (например, уменьшая время поиска, группируя запись соседних дисковых блоков). Однако этот механизм имеет свои недостатки, поскольку может привести к нарушению целостности файловой системы в случае неожиданного останова или сбоя операционной системы.

## Операции ввода/вывода

На рис. 4.14 представлена схема выполнения операций ввода/вывода с использованием буферного кэша. Важной особенностью этой подсистемы является то, что она обеспечивает независимое выполнение операций чтения или записи данных процессом как результат соответствующих системных вызовов, а также фактический обмен данными с периферийным устройством.

Когда процессу требуется прочитать или записать данные он использует системные вызовы *read(2)* или *write(2)*, направляя тем самым запрос файловой подсистеме. В свою очередь файловая подсистема транслирует этот запрос в запрос на чтение или запись соответствующих дисковых блоков файла и направляет его в буферный кэш. Прежде всего кэш просматривается на предмет наличия требуемого блока в памяти. Если соответствующий буфер найден, его содержимое копируется в адресное пространство процесса в случае чтения и наоборот при записи, и операция завершается. Если блок в кэше не найден, ядро размещает буфер, связывает его с дисковым блоком с помощью заголовка *buf* и направляет запрос на чтение драйверу устройства. Обычно используется схема *чтения вперед* (read-ahead), когдачитываются не только запрашиваемые блоки, но и блоки, которые с высокой вероятностью могут потребоваться в ближайшее время (рис. 4.14, а). Таким образом, последующие вызовы *read(2)* скорее всего не потребуют дискового ввода/вывода, а будут включать лишь копирование данных из буферов в память процесса, — операция, которая, как отмечалось, обладает на несколько порядков большей производительностью (рис. 4.14, б—в). При запросе на модификацию блока изменения также затрагивают только буфер кэша. При этом ядро помечает буфер как "грязный" в заголовке *buf* (рис. 4.14, г). Перед освобождением такого буфера для повторного использования, его содержимое должно быть предварительно сохранено на диске (рис. 4.14, д).

Перед фактическим использованием буфера, например при чтении или записи буфера процессом, или при операции дискового ввода/вывода, доступ к нему для других процессов должен быть заблокирован. При обращении к уже заблокированному буферу процесс переходит в состояние сна, пока данный ресурс не станет доступным.

<sup>4</sup> Использование буферного кэша позволяет избежать 95% операций чтения с диска и 85% операций записи на диск для типичной конфигурации операционной системы.

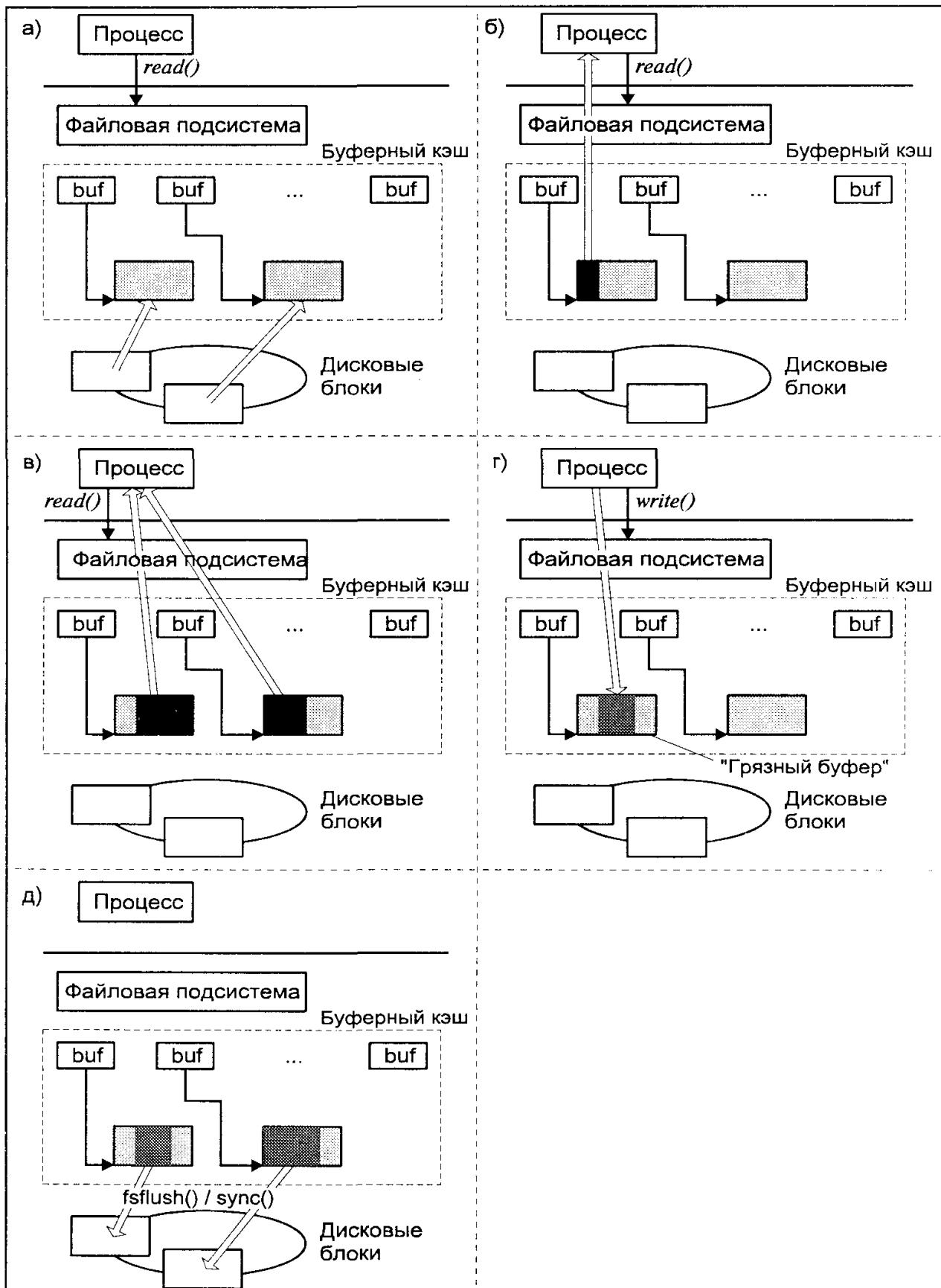


Рис. 4.14. Схема работы буферного кэша

Не заблокированные буферы помечаются как свободные и помещаются в специальный список. Буферы в этом списке располагаются в порядке наименее частого использования (Least Recently Used, LRU). Таким образом, когда ядру необходим буфер, оно выбирает тот, к которому не было обращений в течение наиболее продолжительного промежутка времени. После того как работа с буфером завершена, он помещается в конец списка и является наименее вероятным кандидатом на освобождение и повторное использование. Поэтому, если процесс вскоре опять обратится к тому же блоку данных, операция ввода/вывода по-прежнему будет происходить с буфером кэша. С течением времени буфер перемещается в направлении начала очереди, но при каждом последующем обращении к нему, будет помещен в ее конец.

Основной проблемой, связанной с буферным кэшем, является "старение" информации, хранящейся в дисковых блоках, образы которых находятся в буферном кэше. Как следует из схемы работы кэша, большинство изменений затрагивают только данные в соответствующих буферах, в то время, как дисковые блоки хранят уже устаревшую информацию. Разумеется в нормально работающей системе проблемы как таковой не возникает, поскольку в операциях ввода/вывода всегда используются свежие данные буферного кэша. Однако при аварийном останове системы, это может привести к потере изменений данных файлов, сделанных процессами непосредственно перед остановом.

Для уменьшения вероятности таких потерь в UNIX имеется несколько возможностей:

- Во-первых, может использоваться системный вызов *sync(2)*, который обновляет все дисковые блоки, соответствующие "грязным" буферам. Необходимо отметить, что *sync(2)* не ожидает завершения операции ввода/вывода, таким образом после возврата из функции не гарантируется, что все "грязные" буферы сохранены на диске<sup>5</sup>.
- Во-вторых, процесс может открыть файл в синхронном режиме (указав флаг *O\_SYNC* в системном вызове *open(2)*). При этом все изменения в файле будут немедленно сохраняться на диске.
- Наконец, через регулярные промежутки времени в системе пробуждается специальный системный процесс — диспетчер буферного кэша (в различных версиях UNIX его названия отличаются, чаще всего используется *fsflush* или *bdflush*). Этот процесс освобождает

В распоряжении администратора имеется командный интерфейс к системному вызову — утилита *sync(1M)*. Поскольку выполнение команды еще не свидетельствует о фактическом завершении ввода/вывода, администраторы практикуют вызов *sync(1M)* несколько раз. Повторные вызовы повышают вероятность того, что ввод/вывод будет завершен прежде, чем будет введена другая команда или остановлена система, поскольку набор команды занимает определенное время. Тот же эффект может быть достигнут просто ожиданием нескольких секунд после ввода *sync(1M)*, но набор команды позволяет "скрасить ожидание".

"грязные" буферы, сохраняя их содержимое в соответствующих дисковых блоках<sup>6</sup> (рис. 4.14, д).

## Кэширование в SVR4

Центральной концепцией в архитектуре виртуальной памяти SVR4 является отображение файлов. При этом подходе все адресное пространство может быть представлено набором отображений различных файлов в память. Действительно, в страницы памяти, содержащие кодовые сегменты, отображаются соответствующие секции исполняемых файлов. Процесс может задать отображение с помощью системного вызова  *mmap(2)*, при этом страницам памяти будут соответствовать определенные участки отображаемого файла. Даже области памяти, содержимое которых изменяется и не связано ни с каким файлом файловой системы, т. н. *анонимные страницы*, можно отобразить на определенные участки специального файла устройства, отвечающего за область swapинга (именно там сохраняются анонимные объекты памяти). При этом фактический обмен данными между памятью и устройствами их хранения, инициируется возникновением страничной ошибки. Такая архитектура позволяет унифицировать операции ввода/вывода практически для всех случаев.

При этом подходе, когда процесс выполняет вызовы  *read(2)* или  *write(2)*, ядро устанавливает отображение части файла, адресованного этими вызовами, в собственное адресное пространство. Затем эта область копируется в адресное пространство процесса. При копировании возникают страничные ошибки, приводящие в фактическому считыванию дисковых блоков файла в память. Поскольку все операции кэширования данных в этом случае обслуживаются подсистемой управления памятью, необходимость в буферном кэше, как отдельной подсистеме, отпадает.

## Целостность файловой системы

Значительная часть файловой системы находится в оперативной памяти. А именно, в оперативной памяти расположены суперблок примонтированной системы, метаданные активных файлов (в виде системно-зависимых *inode* и соответствующих им *vnode*) и даже отдельные блоки хранения данных файлов, временно находящиеся в буферном кэше.

Работа диспетчера буферного кэша зависит от версии UNIX и конкретных настроек ядра системы. Например, в SCO UNIX для этого используются несколько параметров. Параметр *BDFLUSHR* задает интервал между последовательными пробуждениями *bdflush*, его значение по умолчанию составляет 30 секунд. Параметр *NAUTOP* задает промежуток времени, который буфер должен оставаться "грязным", прежде чем *bdflush* сохранит его на диске.

Для операционной системы рассогласование между буферным кэшем и блоками хранения данных отдельных файлов, не приведет к катастрофическим последствиям даже в случае внезапного останова системы, хотя с точки зрения пользователя все может выглядеть иначе. Содержимое отдельных файлов не вносит существенных нарушений в целостность файловой системы.

Другое дело, когда подобные несоответствия затрагивают метаданные файла или другую управляющую информацию файловой системы, например, суперблок. Многие файловые операции затрагивают сразу несколько объектов файловой системы, и если на диске будут сохранены изменения только для части этих объектов, целостность файловой системы может быть существенно нарушена.

Рассмотрим пример создания жесткой связи для файла. Для этого файловой подсистеме необходимо выполнить следующие операции:

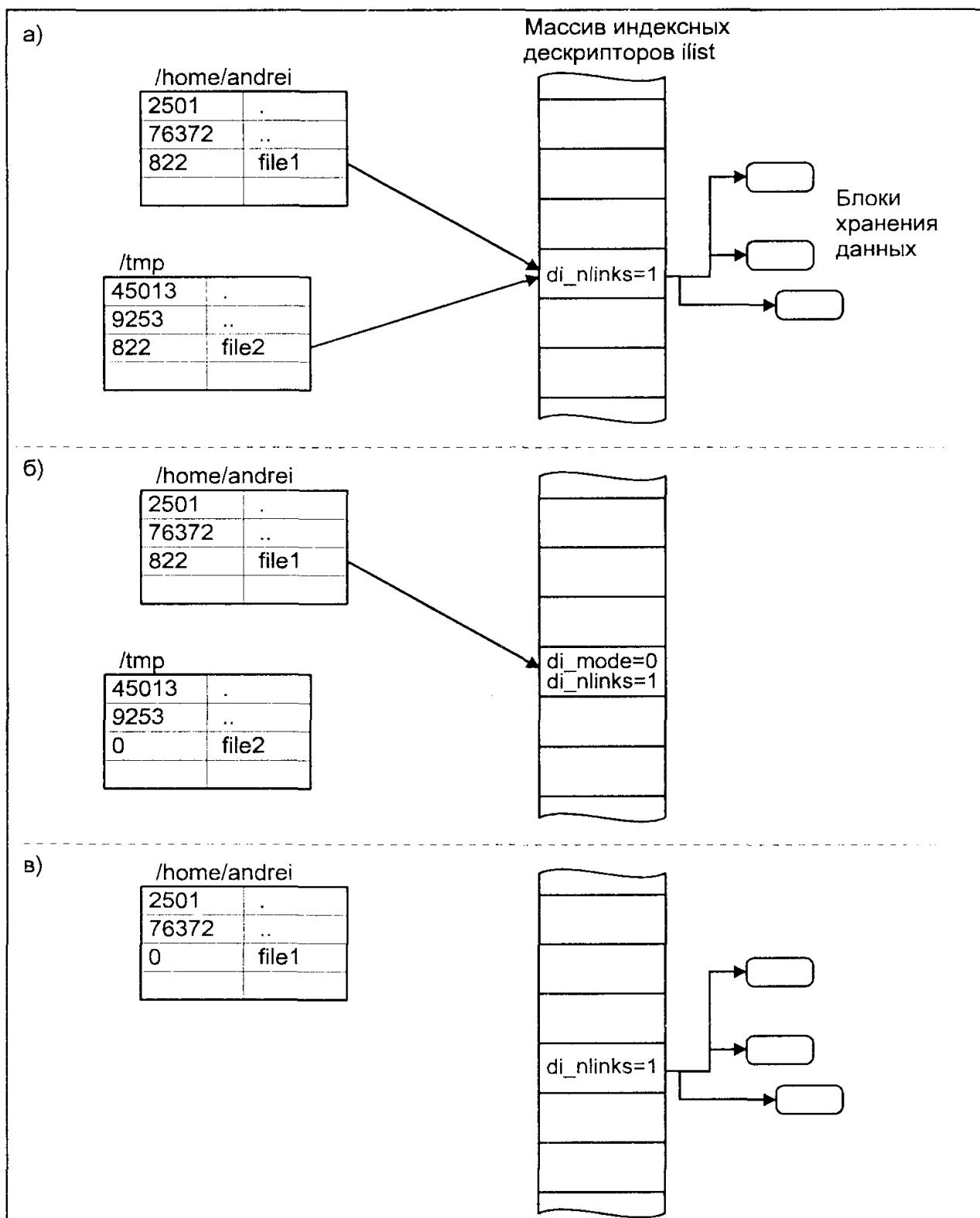
1. Создать новую запись в необходимом каталоге, указывающую на inode файла.
2. Увеличить счетчик связей в inode.

Предположим, что аварийный останов системы произошел между первой и второй операциями. В этом случае после запуска в файловой системе будут существовать два имени файла (две записи каталогов), адресующие inode со счетчиком связей `di_nlinks`, равным 1. Эта ситуация показана на рис. 4.15 (а). Если теперь будет удалено одно из имен, это приведет к удалению файла как такового, т. е. к освобождению блоков хранения данных и inode, поскольку счетчик связей `di_nlinks` станет равным 0. Оставшаяся запись каталога будет указывать на неразмещенный индексный дескриптор, или inode, адресующий уже другой файл (рис. 4.15, б).

Порядок операций с метаданными может иметь существенное влияние на целостность файловой системы. Рассмотрим, например, предыдущий пример. Допустим, порядок операций был изменен и, как и прежде, останов произошел между первой и второй операциями. После запуска системы файл будет иметь лишнюю жесткую связь, но существующая запись каталога останется правильной. Тем не менее при удалении имени файла фактически файл удален не будет, поскольку число связей останется равным 1 (рис. 4.15, в). Хотя это также является ошибкой, результатом которой является засорение дискового пространства, ее последствия все же менее катастрофичны, чем в первом случае.

Ядро выбирает порядок совершения операций с метаданными таким образом, чтобы вред от ошибок в случае аварии был минимальным. Однако проблема нарушения этого порядка все же остается, т. к. драйвер может изменять очередность выполнения запросов для оптимизации ввода/вывода. Единственной возможностью сохранить выбранный порядок является синхронизация операций со стороны файловой подсистемы.

В нашем примере файловая подсистема будет ожидать, пока на диск не будет записано содержимое индексного дескриптора, и только после этого произведет изменения каталога.



**Рис. 4.15.** Нарушение целостности файловой системы

Отсутствие синхронизации между образом файловой системы в памяти и ее данными на диске в случае аварийного останова может привести к появлению следующих ошибок:

1. Один блок адресуется несколькими mode (принадлежит нескольким файлам).
2. Блок помечен как свободный, но в то же время занят (на него ссылается inode).
3. Блок помечен как занятый, но в то же время свободен (ни один inode на него не ссылается).
4. Неправильное число ссылок в inode (недостаток или избыток ссылающихся записей в каталогах).
5. Несовпадение между размером файла и суммарным размером адресуемых inode блоков.
6. Недопустимые адресуемые блоки (например, расположенные за пределами файловой системы).
7. "Потерянные" файлы (правильные inode, на которые не ссылается записи каталогов).
8. Недопустимые или неразмещенные номера inode в записях каталогов.

Эти ошибки схематически показаны на рис. 4.16.

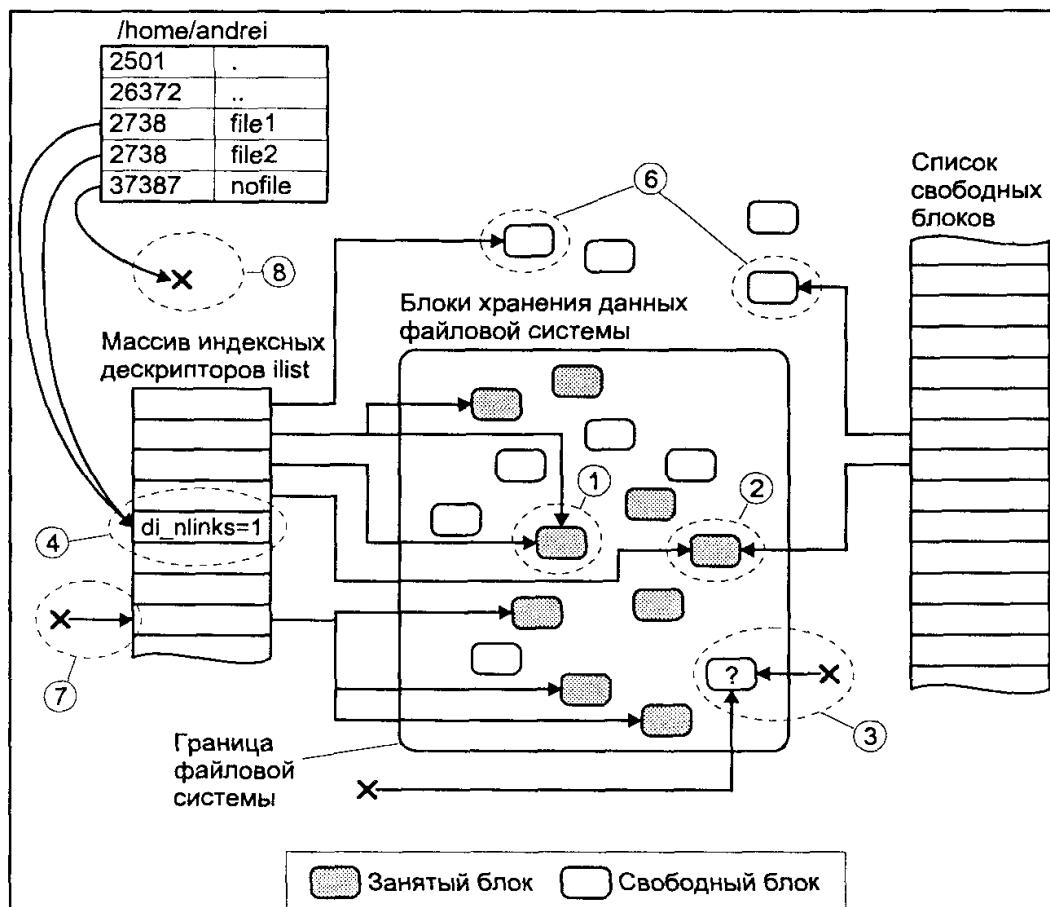


Рис. 4.16. Возможные ошибки файловой системы

Если нарушение все же произошло, на помощь может прийти утилита *fsck(1M)*, производящая исправление файловой системы. Запуск этой утилиты может производиться автоматически каждый раз при запуске системы, или администратором, с помощью команды:

```
fsck [options] filesystem
```

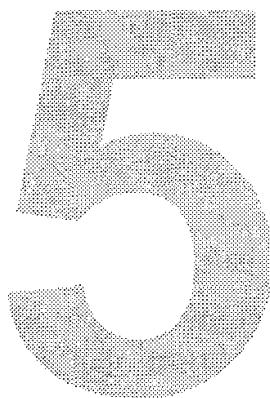
где *filesystem* — специальный файл устройства, на котором находится файловая система.

Проверка и исправление должны производиться только на размонтированной файловой системе. Это связано с необходимостью исключения синхронизации таблиц в памяти (ошибочных) с их дисковыми эквивалентами (исправленными). Исключение составляет корневая файловая система, которая не может быть размонтирована. Для ее исправления необходимо использовать опцию *-b*, обеспечивающую немедленный перезапуск системы после проведения проверки.

## **Заключение**

В этой главе описана организация файловой подсистемы UNIX. Начав разговор с обсуждения архитектуры традиционных файловых систем UNIX, мы остановились на анализе т. н. виртуальной файловой системы, обеспечивающей единый интерфейс доступа к различным типам физических файловых систем.

Мы также рассмотрели, каким образом происходит доступ процесса к данным, хранящимся в файлах, вплотную подошли к разговору о подсистеме ввода/вывода, который и продолжим в следующей главе.



## Подсистема ввода/вывода

Фактическая архитектура ввода/вывода скрыта от прикладного процесса несколькими интерфейсами. Один из них — интерфейс файловой системы был рассмотрен в предыдущей главе. Взаимодействие с удаленными ресурсами обеспечивается сетевыми интерфейсами сокетов или ТЫ (Transport Layer Interface), которые описываются в главе 6. Однако возможны ситуации, когда прикладному процессу требуется взаимодействие с периферийными устройствами на более низком уровне. Хотя в этом случае роль файловой подсистемы не столь велика, как при работе с обычными файлами, все равно ядро предоставляет процессу унифицированную схему, скрывающую истинную архитектуру того или иного устройства.

В конечном итоге работа всех этих интерфейсов, как высокого уровня, (файловая система), так и более низкого (взаимодействие с физическим устройством), обеспечивается подсистемой ввода/вывода ядра операционной системы.

В данной главе мы ознакомимся с архитектурой этой подсистемы, основным компонентом которой являются драйверы — модули ядра, обеспечивающие непосредственную работу с периферийными устройствами. Поскольку характеристики периферийных устройств значительно различаются, то UNIX использует два основных типа драйверов — *символьные* и *блочные*. Как следует из названия, драйверы первого типа обеспечивают обмен сравнительно небольшими объемами данных с устройством, что имеет место при работе, например, с терминалами или принтерами. Драйверы второго типа производят передачу данных блоками, что характерно для дисковых носителей данных. Эти типы драйверов входят в традиционную подсистему ввода/вывода и присутствуют во всех версиях UNIX.

Во второй части главы мы подробно остановимся на архитектуре драйверов подсистемы STREAMS, которая является неотъемлемой частью ядра в версиях UNIX System V. Эти драйверы представляют собой отдельный тип, обладающий такими ценными возможностями, как буферизация и управление потоком данных. К подсистеме STREAMS мы также вернемся в следующей главе при обсуждении архитектуры сетевого доступа в UNIX System V.

## Драйверы устройств

Драйверы устройств обеспечивают интерфейс между ядром UNIX и аппаратной частью компьютера. Благодаря этому от остальной части ядра скрыты архитектурные особенности компьютера, что значительно упрощает перенос системы и поддержку работы различных периферийных устройств.

В UNIX существует большое количество драйверов. Часть из них обеспечивает доступ к физическим устройствам, например, жесткому диску, принтеру или терминалу, другие предоставляют аппаратно-независимые услуги. Примером последних могут служить драйверы **/dev/kmem** для работы с виртуальной памятью ядра **/dev/null**, представляющий "нулевое" устройство.

В процессе запуска системы ядро вызывает соответствующие процедуры инициализации установленных драйверов. Во многих версиях UNIX эти процедуры выводят на консоль сообщение о том, что драйвер найден, и инициализация прошла успешно, а также параметры драйвера и устройства.

### Типы драйверов

Драйверы различаются по возможностям, которые они предоставляют, а также по тому, каким образом обеспечивается к ним доступ и управление. Можно рассматривать три основные типа драйверов:

**Символьные драйверы** Этот тип драйверов обеспечивает работу с устройствами с побайтовым доступом и обменом данными. К таким устройствам можно отнести модемы, терминалы, принтеры, манипуляторы мышь и т. д.

Доступ к таким драйверам не включает использование буферного кэша, таким образом ввод и вывод как правило не буферизуется. При необходимости буферизации для символьных драйверов обычно используется подход, основанный на структурах данных, называемых *clist*.

**Блочные драйверы** Этот тип драйверов позволяет производить обмен данными с устройством фиксированными порциями (блоками). Например, для жесткого диска данные можно адресовать и, соответственно, читать только секторами, размер которых составляет несколько сотен байтов. Для блочных драйверов обычно используется буферный кэш, который и является интерфейсом между файловой системой и устройством.

Хотя операции чтения и записи для процесса допускают обмен данными, размер которых меньше размера блока, на системном уровне это все равно приводит к считыванию всего блока, изменению части его данных и записи измененного блока обратно на диск.

### Драйверы низкого уровня (raw drivers)

Этот тип интерфейса блочных драйверов позволяет производить обмен данными с блочными устройствами, минуя буферный кэш. Это, в частности, означает, что устройство может быть адресовано элементами, размер которых не совпадает с размером блока.

Обмен данными происходит независимо от файловой подсистемы и буферного кэша, что позволяет ядру производить передачу непосредственно между пользовательским процессом и устройством, без дополнительного копирования.

На рис. 5.1 приведена упрощенная схема взаимодействия драйверов устройств с другими подсистемами операционной системы UNIX.

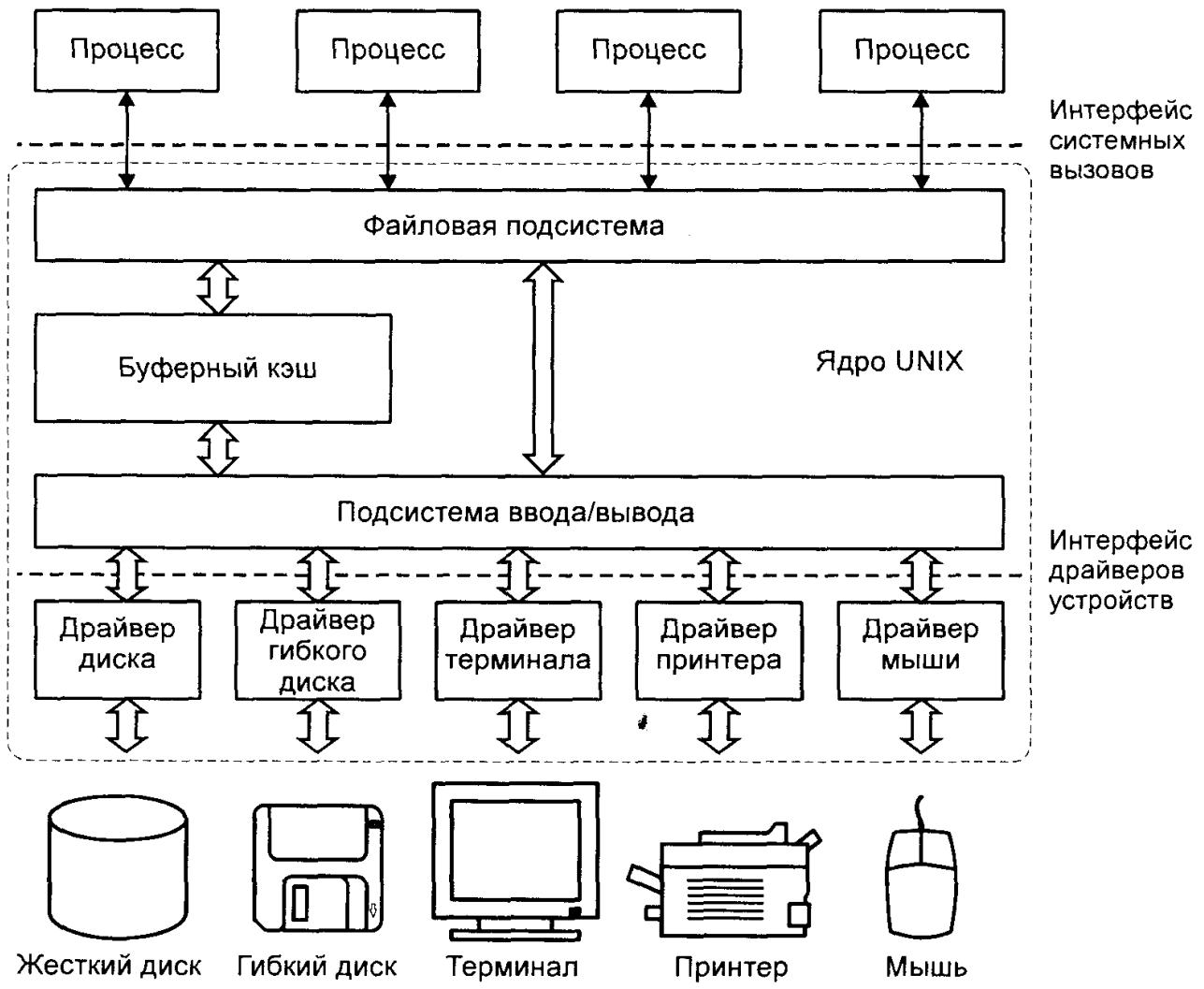


Рис. 5.1. Драйверы устройств UNIX

Не все драйверы служат для работы с физическими устройствами, такими как сетевой адаптер, последовательный порт или монитор. Часть драйве-

ров служат для предоставления различных услуг ядра прикладным процессам и не имеют непосредственного отношения к аппаратной части компьютера. Такие драйверы называются *программными* или драйверами *псевдоустройств*. Можно привести несколько примеров псевдоустройств и соответствующих им программных драйверов:

<b>/dev/kmem</b>	Обеспечивает доступ к виртуальной памяти ядра. Зная виртуальные адреса внутренних структур ядра, процесс может считывать хранящуюся в них информацию. С помощью этого драйвера может, например, быть реализована версия утилиты <i>ps(1)</i> , выводящей информацию о состоянии процессов в системе.
<b>/dev/ksyms</b>	Обеспечивает доступ к разделу исполняемого файла ядра, содержащего таблицу символов. Совместно с драйвером <b>/dev/kmem</b> обеспечивает удобный интерфейс для анализа внутренних структур ядра.
<b>/dev/mem</b>	Обеспечивает доступ к физической памяти компьютера.
<b>/dev/null</b>	Является "нулевым" устройством. При записи в это устройство данные просто удаляются, а при чтении процессу возвращается 0 байтов. Примеры использования этого устройства рассматривались в главе 1, когда с помощью <b>/dev/null</b> мы подавляли вывод сообщений об ошибках.
<b>/dev/zero</b>	Обеспечивает заполнение нулями указанного буфера. Этот драйвер часто используется для инициализации области памяти.

## Базовая архитектура драйверов

Драйвер устройства адресуется *старшим номером* (major number) устройства. Напомним, что среди атрибутов специальных файлов устройств, которые обеспечивают пользовательский интерфейс доступа к периферии компьютера, это число присутствует наряду с другим, также имеющим отношение к драйверу, — *младшим номером* (minor number). Младший номер интерпретируется самим драйвером (например, для клонов, оно задает старшее число устройства, которое требуется "размножить"). Другим примером использования младших номеров может служить драйвер диска. В то время как доступ к любому из разделов диска осуществляется одним и тем же драйвером и, соответственно, через один и тот же старший номер, младший номер указывает, к какому именно разделу требуется обеспечить доступ.

Доступ к драйверу осуществляется ядром через специальную структуру данных (*коммутатор устройств*), каждый элемент которой содержит указатели на соответствующие функции драйвера — *точки входа*. Старшее число, по существу, является указателем на элемент коммутатора устройств, обеспечивая, тем самым, ядру возможность вызова необходимой функции указанного драйвера. Таким образом, коммутатор устройств определяет базовый интерфейс драйвера устройств.

Этот интерфейс различен для блочных и символьных устройств. Ядро содержит коммутаторы устройств двух типов: `bdevsw` для блочных и `cdevsw` для символьных устройств. Ядро размещает отдельный массив для каждого типа коммутатора, и любой драйвер устройства имеет запись в соответствующем массиве. Если драйвер обеспечивает как блочный, так и символьный интерфейсы, его точки входа будут представлены в обоих массивах.

Типичное описание этих двух массивов имеет следующий вид (назначение различных точек входа мы рассмотрим далее в этом разделе):

```
struct bdevsw[] {
    int (*d_open)();
    int (*d_close)();
    int (*d_strategy)();
    int (*d_size)();
    int (*d_xhalt)();
}

} bdevsw[];

struct cdevsw[] {
    int (*d_open)();
    int (*d_close)();
    int (*d_read)();
    int (*d_write)();
    int (*d_ioctl)();
    int (*d_xpoll)();
    int (*d_xhalt)();
    struct streamtab *d_str;
}

} cdevsw[];
```

Ядро вызывает функцию `open()` требуемого драйвера следующим образом:

```
(*bdevsw[getmajor(dev)].d_open)(dev, ...);
```

передавая ей в качестве одного из параметров переменную `dev` (типа `dev_t`), содержащую старший и младший номера. Макрос `getmajor()` служит для извлечения старшего номера из переменной `dev`. Благодаря этому драйвер имеет возможность определить, с каким младшим номером была вызвана функция `open()`, и выполнить соответствующие действия.

Коммутатор определяет абстрактный интерфейс драйвера устройства. Каждый драйвер обеспечивает соответствующую реализацию функций этого интерфейса. Если драйвер не поддерживает каких-либо функций стандартного интерфейса, он заменяет соответствующие точки входа специальными заглушками, предоставляемыми ядром. Когда ядру требуется запросить какую-либо операцию у драйвера устройства, оно определяет элемент коммутатора, соответствующий данному драйверу (используя его старший номер), и вызывает требуемую функцию.

В названиях точек входа драйвера используются определенные соглашения. Поскольку в ядре системы одновременно присутствует большое ко-

личество различных драйверов, каждый из них должен иметь уникальное имя во избежание проблем при компиляции (точнее, при редактировании связей) ядра. Каждый драйвер имеет уникальное двухсимвольное обозначение, используемое в качестве префикса названий функций. Например, драйвер виртуальной памяти ядра **/dev/kmem** имеет префикс *mm*, таким образом функции этого драйвера будут иметь названия *mmopen()*, *mmclose()*, *mmread()* и *mmwrite()*.

В табл. 5.1 приведены некоторые точки входа, общие для различных типов драйверов, а символами *xx*, с которых начинается имя каждой функции, обозначен уникальный префикс драйвера. Стандартные точки входа драйвера отличаются для разных версий UNIX. Например, некоторые версии имеют расширенный коммутатор блочных устройств, включающий такие функции, как *xxioctl()*, *xxread()* и *xxwrite()*. В некоторых версиях включены точки входа для инициализации и сброса шины данных.

**Таблица 5.1.** Типичные точки входа в драйвер устройства

Точка входа	Символьный	Блочный	Низкого уровня	Назначение
<i>xxopen()</i>	+	+	+	Вызывается при каждой операции открытии устройства. Обеспечивает необходимую инициализацию физического устройства и внутренних данных драйвера. Например, для каждого последующего открытия драйвера могут размещаться дополнительные буферы, обеспечивающие возможность независимой работы с устройством нескольким процессам.
<i>xxclose()</i>	+	+	+	Вызывается, когда число ссылок на данный драйвер становится равным нулю, т. е. ни один из процессов системы не работает с устройством (не имеет открытым соответствующий файл устройства). Может вызывать отключение физического устройства. Например, драйвер накопителя на магнитной ленте может перемотать ленту в начало.
<i>xxread()</i>	+	—	+	Производит чтение данных от устройства.
<i>xxwrite()</i>	+	—	+	Производит запись данных на устройство.

Таблица 5.1 (продолжение)

Точка входа	Символьный	Блоч-ный	Низкого уровня	Назначение
xxioctl()	+		+	Является общим интерфейсом управления устройством. Драйвер может определить набор команд, которые могут быть переданы ему, например с помощью системного вызова <i>ioctl(2)</i> .
xxintr()	+	+	+	Вызывается при поступлении прерывания, связанного с данным устройством. Может выполнить копирование данных от устройства в промежуточные буфера, которые затемчитываются функцией <i>xxread()</i> по запросу прикладного процесса.
xxpoll()	+		+	Производит опрос устройства. Обычно используется для устройств, не поддерживающих прерывания, например, для определения поступления данных для чтения.
xxhalt()	+	+	+	Вызывается для останова драйвера при останове системы или при выгрузке драйвера.
xxstrategy()		+	+	Общая точка входа для операций блочного ввода/вывода. Название функции говорит о том, что устройство может обеспечивать собственную стратегию обработки поступающих запросов, например, изменять их порядок для повышения производительности ввода/вывода. Если устройство занято, функция помещает запросы в очередь. В этом случае фактический ввод/вывод инициирует функция обработки прерывания, которая вызывается, когда устройство закончит предыдущую операцию ввода/вывода.
xxprint()		+	+	Выводит сообщение драйвера на консоль, обычно при запуске системы.

Ядро вызывает те или иные функции драйвера в зависимости от запроса. Например, если процесс выполняет системный вызов `read(2)` для специального файла символьного устройства, ядро вызовет функцию `xxread()` для соответствующего символьного драйвера. Если же процесс запрашивает ту же операцию для обычного дискового файла, ядро вызовет процедуру `xxstrategy()` для блочного драйвера, обслуживающего данную файловую систему.

Вообще говоря, можно выделить пять основных случаев, в которых ядро обращается к функциям драйвера:

- Автоконфигурация.* Обычно происходит в процессе инициализации UNIX, когда ядро определяет, какие устройства доступны в системе.
- Ввод/вывод.* Запрос на операцию ввода/вывода может быть инициирован как прикладным процессом, так и некоторыми подсистемами ядра, например, подсистемой управления памятью.
- Обработка прерываний.* Ядро вызывает соответствующую функцию драйвера для обработки прерывания, поступившего от устройства (если устройство способно генерировать прерывания).
- Специальные запросы.* Ядро вызывает соответствующую функцию драйвера для обработки специальных команд, полученных с помощью системного вызова `ioctl(2)`.
- Реинициализация/Останов.* Некоторые типы аппаратных архитектур могут требовать сброса и реинициализации устройства. Определенные функции драйвера также вызываются при останове операционной системы.

На рис. 5.2 и 5.3 приведены схемы доступа к драйверам символьного и блочного устройств.

Как видно из рисунков, схема обработки запроса ядром UNIX различна для символьных и блочных устройств.

При обсуждении точек входа драйверов устройств следует иметь в виду, что большинство функций драйвера, отвечающих за передачу данных, осуществляют копирование информации из адресного пространства ядра, в котором находится сам драйвер, в адресное пространство задачи. Когда ядро вызывает функцию драйвера, все действия выполняются в системном контексте процесса. Однако схема вызова функций может быть различной:

- Функция может быть вызвана по запросу процесса.* Например, если процесс выполняет системный вызов `read(2)`, ядро вызывает соответствующую точку входа драйвера `xxread()`, обеспечивающую работу с файлом. В этом случае говорят, что функция имеет *контекст задачи*.
- Функция может быть вызвана другой подсистемой ядра операционной системы.* Например, для блочного драйвера функция `xxstrategy()` может быть вызвана страничным демоном, для со-

хранения страниц во вторичной памяти (как правило, на жестком диске). Поскольку страничный демон представляет собой системный процесс, выполняющийся только в контексте ядра, функция `xxstrategy()` в этом случае имеет *системный контекст*.

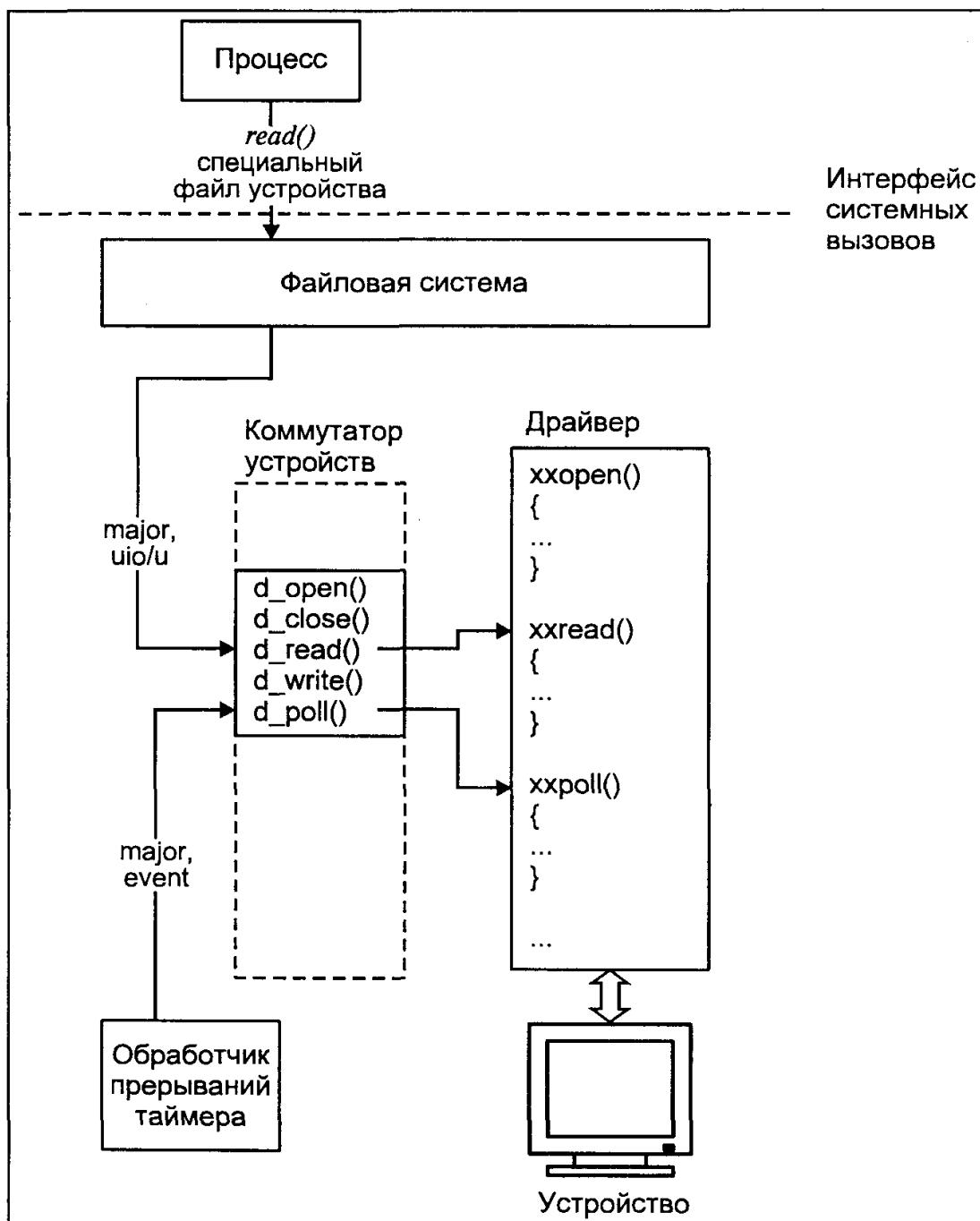


Рис. 5.2. Доступ к драйверу символьного устройства

- Если функция вызывается в процессе обработки прерывания, то она имеет *контекст прерывания* — специальный вид системного контекста. Функции драйвера, отвечающие за обработку прерывания, например `xxintr()` имеют этот тип контекста.

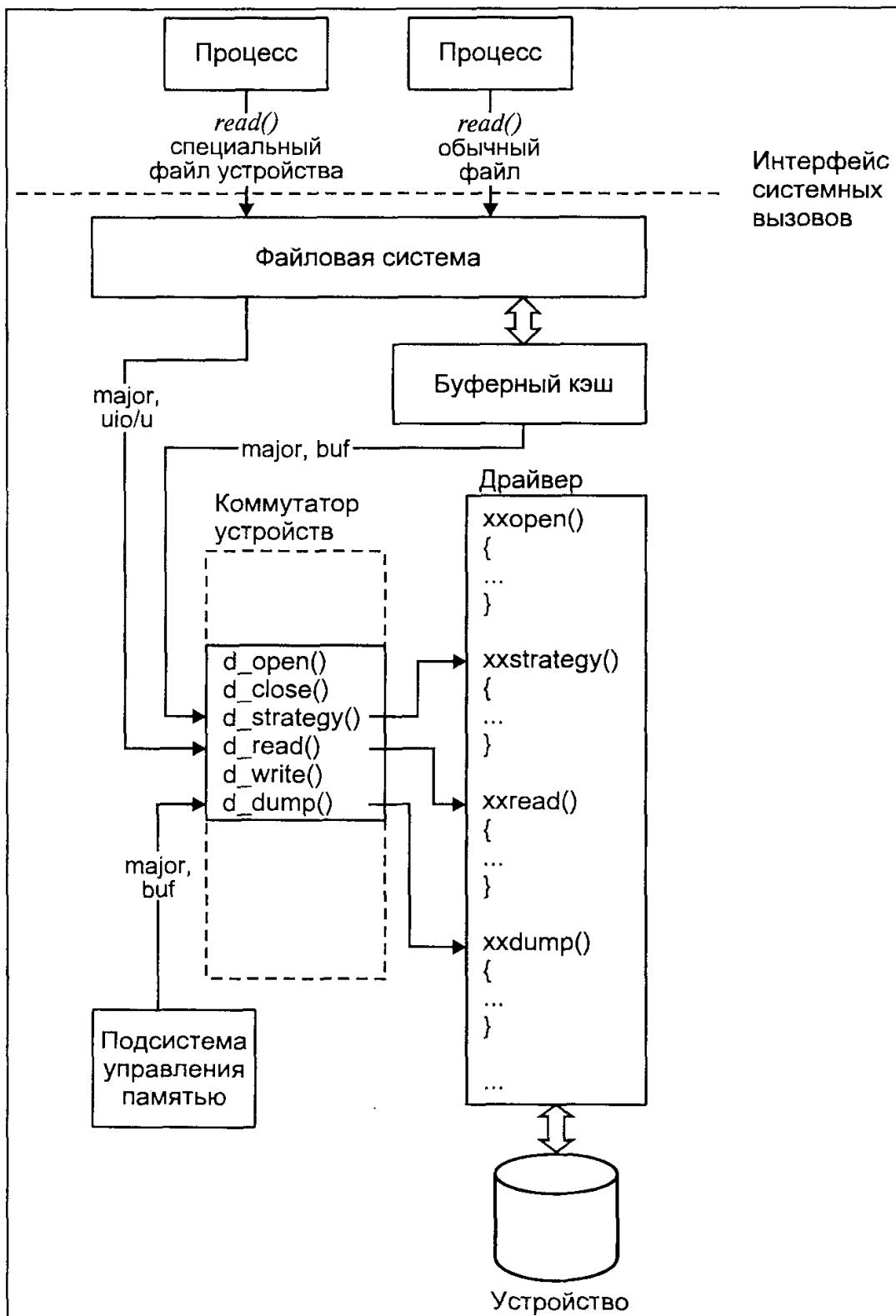


Рис. 5.3. Доступ к драйверу блочного устройства

Различия в контексте и причинах вызова тех или иных функций драйвера позволяют представить драйвер устройства состоящим из двух частей: *верхней части* (top half) и *нижней части* (bottom half). Функции верхней

части драйвера имеют синхронный характер, т. е. вызываются по определенным запросам прикладного процесса и выполняются в его контексте. Таким образом, для этих функций доступно адресное пространство и *u-area* процесса, и при необходимости эти функции могут перевести процесс в состояние сна (вызовом функции `sleep()` ядра). Функции ввода/вывода и управления принадлежат верхней части драйвера.

Вызов функций нижней части носит асинхронный характер. Например, момент вызова функции обработки прерываний нельзя предугадать, и ядро не может контролировать, когда эта функция будет вызвана. Выполнение таких функций происходит в контексте ядра и обычно не имеет никакого отношения к контексту текущего процесса. Таким образом, функции системного контекста не имеют права адресовать структуры данных текущего процесса, например его *u-area*, а также не могут перевести процесс в состояние сна, поскольку это заблокирует процесс, не имеющий непосредственного отношения к работе драйвера.

Две части драйвера требуют синхронизации. Например, в случае, когда функции обеих частей используют одну и ту же структуру данных, функция верхней части при выполнении должна заблокировать прерывания на период работы с "разделяемой" областью памяти. В противном случае, прерывание может поступить в тот момент, когда целостность структуры данных нарушена, что приведет к непредсказуемым результатам.

Все представленные выше функции, за исключением `xxhalt()`, `xxpoll()` и `xxintr()`, принадлежат верхней части драйвера. Функция `xxhalt()` вызывается ядром при останове системы и, таким образом, имеет системный контекст, не связанный с контекстом прикладного процесса.

Функция `xxpoll()` обычно вызывается при обработке ядром прерывания таймера для всех устройств, указанных как опрашиваемые. Это необходимо, в частности, для устройств, которые не могут или "не хотят" использовать аппаратные прерывания. Вместо этого `xxpoll()` может использоваться для эмуляции прерываний, например вызывая функцию `xxintr()` на каждый *n*-ный тик системного таймера. Поэтому и функция `xxpoll()` и функция обработки прерывания `xxintr()` не могут рассчитывать на контекст прикладного процесса. В большинстве версий UNIX функции опроса и обработки прерываний вызываются не через коммутатор устройств, а через специальные таблицы ядра.

В UNIX SVR4 определены две дополнительные точки входа — `init()` и `start()`. Драйвер регистрирует эти функции в таблицах ядра `io_init[]` и `io_start[]`. Код начальной загрузки системы запускает функции `xxinit()` перед инициализацией ядра, а функции `xxstart()` сразу же после инициализации.

## Файловый интерфейс

В главе 4 мы рассмотрели интерфейс т. н. независимой или виртуальной файловой системы, обеспечивающей унифицированный интерфейс работы с различными типами физических файловых систем (например, ufs или s5fs), имеющих разные внутренние структуры и возможности. При этом подходе используется унифицированный формат метаданных активных файлов, которые хранятся в памяти (в *in-core* — таблице индексных дескрипторов) и не зависят от конкретной реализации файловой системы. Эти объекты получили название виртуальных индексных дескрипторов или *vnode*. Для каждого *vnode* определен набор абстрактных операций, которые реализованы функциями реальных файловых систем. Например, *vnode* файла, расположенного в файловой системе s5fs, адресует вектор операций (или коммутатор файловых систем, *FSS*) *s5fsops*, содержащий конкретные функции этой файловой системы — *s5fs\_close()*, *s5fs\_open()* или *s5fs\_unlink()*.

Этот подход, используемый в большинстве современных версий UNIX, требует соответствующей архитектуры файлового интерфейса к драйверам устройств. Как уже обсуждалось, доступ к периферии в UNIX осуществляется с помощью специальных файлов устройств, расположенных в корневой файловой системе некоторого типа, например ufs. В соответствии с архитектурой виртуальной файловой системы, все операции с этими файлами будут обслуживаться соответствующими функциями реальной файловой системы, в данном случае — ufs.

Однако такой схеме недостает традиционного для UNIX изящества. Специальный файл устройства не является обычным файлом системы ufs. Фактически все операции со специальным файлом устройства выполняются драйвером и не зависят от типа файловой системы. Поэтому было бы логичнее отобразить операции *vnode* не на вектор файловой системы, а непосредственно на коммутатор устройств.

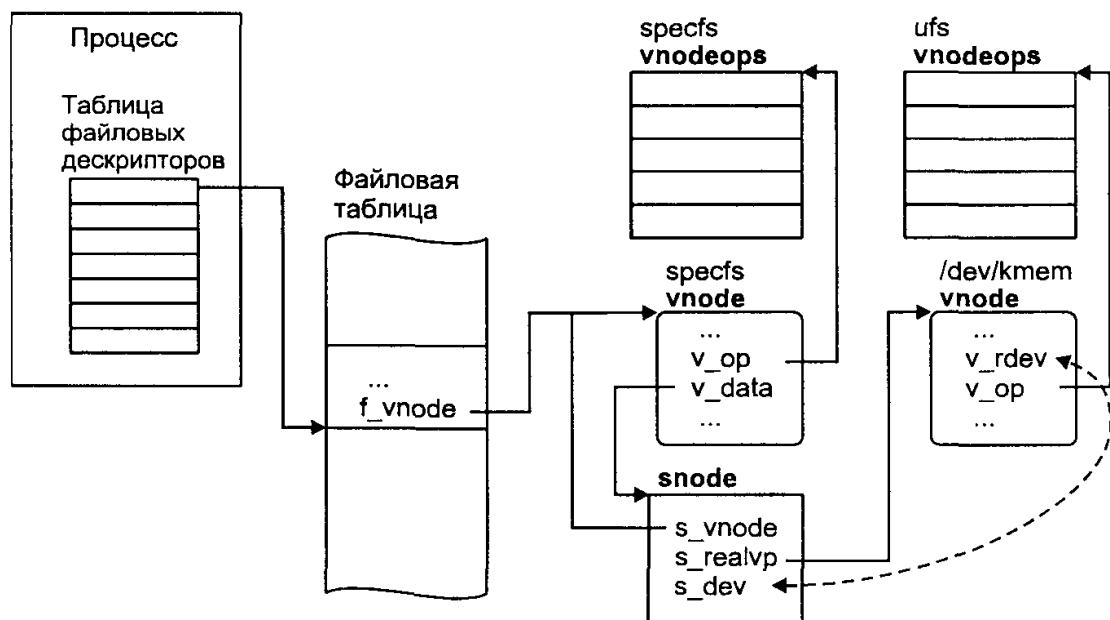
Современные системы ветви System V используют для этого специальный тип файловой системы, называемый *devfs* или *specfs*<sup>1</sup>. Для этого типа файловой системы все операции *vnode* адресуют соответствующие функции требуемого элемента коммутатора устройств. После первоначального открытия файла, когда создается *vnode*, все запросы, связанные со специальным файлом устройства, проходят через *vnode* файловой системы *specfs*.

В то же время открытие файла, например с помощью системного вызова *open(2)*, предусматривает ряд операций, реализованных реальной файловой

В системах SVR4 принятая терминология *specfs*, операционная система SCO UNIX, которая формально является SVR3.2, но фактически имеет многие черты SVR4, называет этот тип файловой системы *devfs*.

системой, в которой находится специальный файл устройства (в нашем примере `ufs`). Одной из таких операций является трансляция имени, которая не может быть реализована файловой системой `specfs`, по существу являющейся виртуальной.

Решение данной проблемы рассмотрим на конкретном примере. Допустим, процесс вызывает функцию *open(2)* для специального файла устройства **/dev/kmem** для работы с виртуальной памятью ядра. Функция трансляции имени файловой системы ufs — *ufs\_lookup()* сначала откроет inode файла **/dev**, а затем, прочитав каталог, обнаружит inode файла **kmem**, при этом будет размещен vnode этого файла. Однако *ufs\_lookup()* определит, что тип этого файла IFCCHR, т. е. специальный файл символьного устройства. Поэтому вместо функции *ufs\_open()*, бессмысленной для этого типа файла, будет вызвана специальная функция файловой системы *specfs*, которая создаст собственный индексный дескриптор, описываемой структурой *snode* (от special inode), для этого файла, если таковой уже не находится в памяти. Согласно стандартной процедуре, также будет создан и виртуальный индексный дескриптор *vnode*, который будет указывать на вектор операций *specops*, которые специально предназначены для работы с драйверами устройств. Например, функции *spec\_open()*, *spec\_read()* или *spec\_write()* в свою очередь вызовут соответствующие точки входа драйвера — функции *xxopen()*, *xxread()* или *xxwrite()*. После этого функции *ufs\_open()* будет передан адрес этого *vnode*, который она, в свою очередь, передаст системному вызову *open(2)*. В результате, *open(2)* вернет процессу файловый дескриптор, адресующий *vnode* файловой системы *specfs*, а не *vnode* файла **/dev/kmem**. Таким образом, все дальнейшие операции с **/dev/kmem** будут перехватываться файловой системой *specfs*. Схема связи процесса с этим *vnode* приведена на рис. 5.4.



**Рис. 5.4.** Связь процесса с файлом /dev/kmem после его открытия

Однако изложенная схема является неполной и имеет ряд существенных недостатков. Дело в том, что драйвер конкретного устройства может адресоваться несколькими специальными файлами устройств, возможно, расположеными в различных физических файловых системах. В этом случае ядро бессильно определить фактическое число связей прикладных процессов с данным устройством, что может потребоваться, например, при вызове функции `xxclose()`, когда все процессы закончили работу с устройством.

Для решения этой проблемы файловая система `specfs` предусматривает наличие дополнительного `snode`, позволяющего контролировать доступ к конкретному устройству. Этот объект, получивший название *общего snode* (`common snode`), является единственным интерфейсом доступа к драйверу устройства. Для каждого устройства (драйвера устройства) существует единственный `common snode`, который создается при первом доступе к устройству. Каждый специальный файл устройства, в свою очередь, имеет собственный `snode` в файловой системе `specfs` и соответствующий ему `vnode`, а также `inode` физической файловой системы, где расположен специальный файл устройства, и соответствующий ему `vnode`.

Для связи всех этих индексных дескрипторов между собой `snode` имеет два поля: `s_commonvp`, указывающее на `common snode`, и `s_realvp`, указывающее на `vnode` специального файла устройства файловой системы, где расположен последний.

Использование тех или иных `vnode` и связанных с ними `inode` или `snode` зависит от конкретных операций, выполняемых процессом с устройством. Большинство из этих операций не зависят от имени специального файла устройства и, соответственно, от реальной файловой системы, в которой он расположен. Эти операции выполняются через `vnode`, соответствующий `common snode`. Однако существует ряд операций, выполнение которых зависит от конкретного специального файла устройства, через который процесс взаимодействует с драйвером. Примером может служить проверка прав доступа при открытии специального файла устройства, которые расположены в `vnode/inode` реальной файловой системы. В этом случае используется `vnode` соответствующего специального файла устройства.

Схема описанной архитектуры приведена на рис. 5.5.

## Клоны

Как уже обсуждалось, старший номер устройства адресует драйвер, в то время как младший номер интерпретируется самим драйвером и может использоваться для различных целей. Например, используя различные младшие номера, процесс может получить доступ к разным разделам жесткого диска, обслуживаемого одним драйвером.

Во многих случаях использование различных младших номеров позволяет нескольким процессам осуществлять одновременную независимую работу с устройством (или псевдоустройством). Каждый младший номер при этом соответствует логическому драйверу, поддерживающему собственные

структуры данных при работе с конкретным процессом. Типичным примером могут служить псевдотерминалы. В таких случаях процессу требуется получить доступ к устройству, при этом его не интересует его младший номер, поскольку разница в младших номерах не отражает разницу в функциональности. Типичным примером являются сетевые протоколы, чаще всего реализованные в виде соответствующих драйверов. Сетевые соединения, основанные на одном и том же протоколе (и, следовательно, работающие с одним и тем же драйвером), используют различные младшие номера для доступа к драйверу. Это позволяет драйверу обеспечивать обработку нескольких сетевых соединений, для каждого из которых поддерживаются собственные структуры данных. Если процессу необходимо установить сетевую связь, ему безразлично, какой младший номер будет у драйвера, главное, чтобы он еще не использовался.

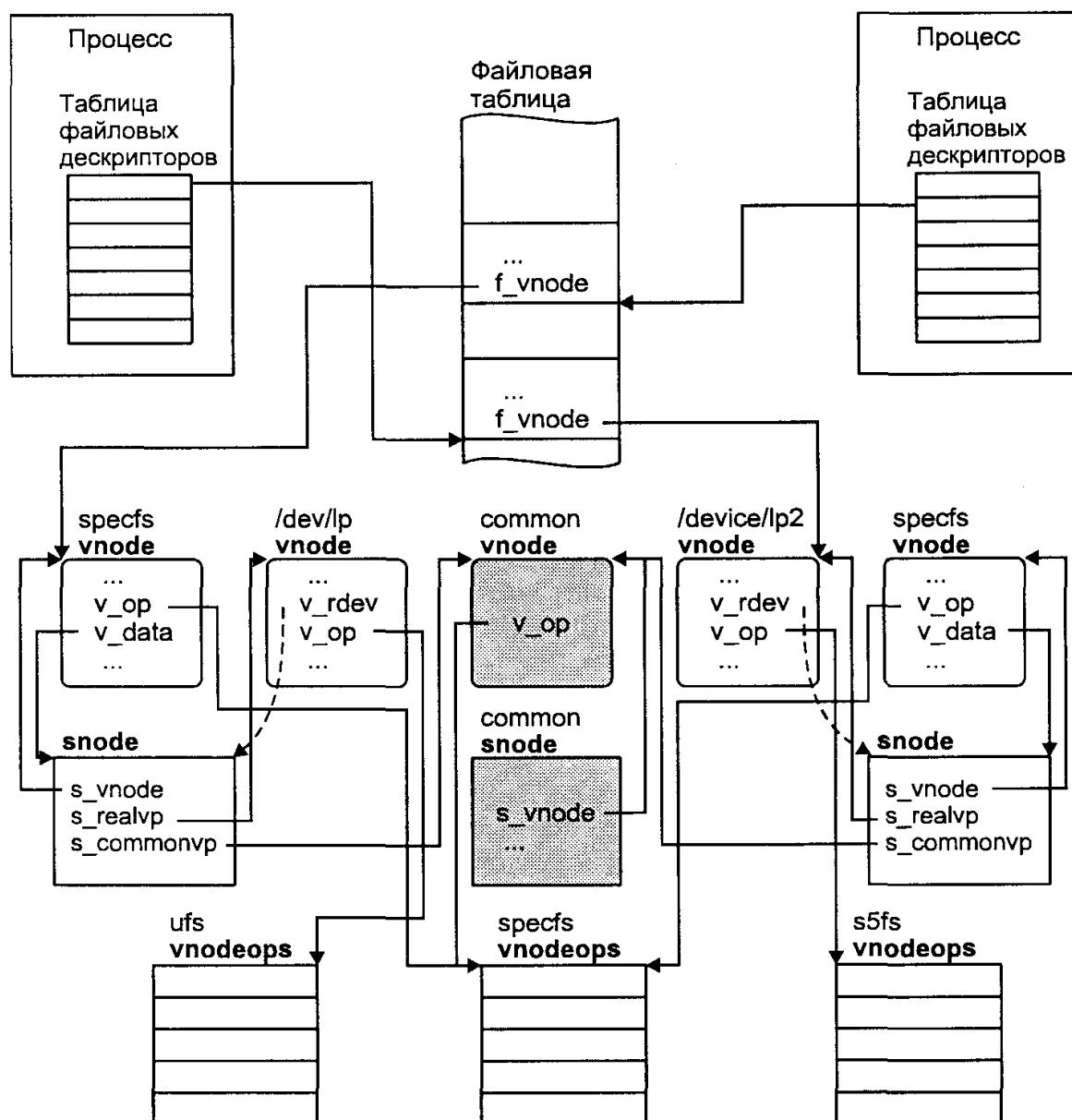


Рис. 5.5. Доступ к устройству через различные специальные файлы

Возможным сценарием доступа к такому устройству может являться перебор различных младших номеров (соответствующих специальных файлов), пока операция `open()` не завершится успешно. Это будет гарантировать, что процесс получил в свое распоряжение отдельное логическое устройство. Другой сценарий возлагает всю работу по поиску неиспользуемого младшего номера устройства на специальные драйверы, получившие названия *клонов*<sup>2</sup>.

Когда процесс открывает специальный файл устройства, происходит инициализация соответствующего *snode* и вызов функции `spec_open()`, реализованной в файловой системе `specfs`, о которой только что говорилось. Эта функция, в свою очередь, вызывает функцию драйвера `xxopen()`, передавая ей в качестве аргумента указатель на номера устройства, сохраненного в поле `s_dev` *snode*. Одной из схем реализации клонов является использование зарезервированного младшего номера. Когда процесс открывает специальный файл устройства с этим номером, функция `xxopen()` выбирает неиспользуемый младший номер и соответственно модифицирует данные *snode* (с помощью указателя на *vnode*, передаваемые ей `spec_open()`). Поскольку доступ процесса к драйверу осуществляется через *vnode* файловой системы `specfs`, все последующие операции будут использовать новый младший номер. Таким образом, процесс получит доступ к новому логическому устройству. Эта схема приведена на рис. 5.6.

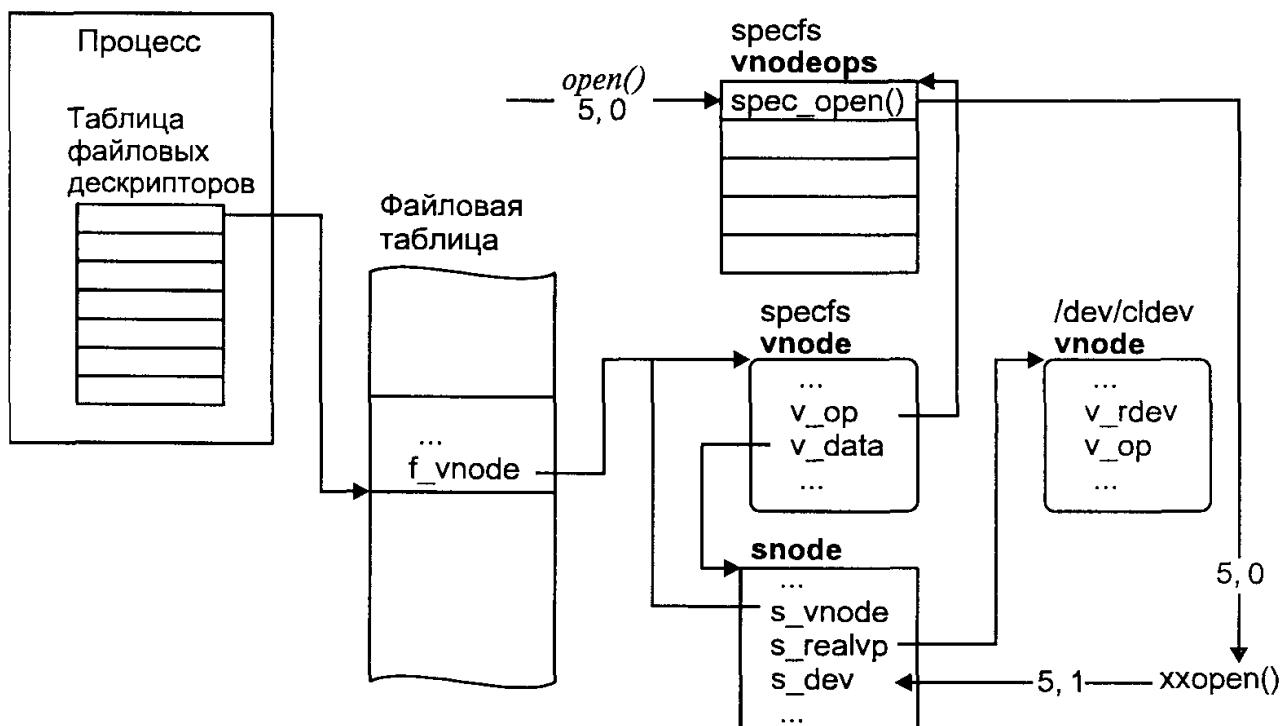


Рис. 5.6. Создание клонов с помощью зарезервированного младшего номера

<sup>2</sup> Clone (англ.) — размножаться.

Другой подход заключается в использовании специального драйвера, обеспечивающего создание клонов, — *драйвера клонов* (clone driver). При этом все драйверы, чье "размножение" обеспечивается таким образом, имеют один и тот же старший номер, адресующий драйвер клонов. Младший номер адресует собственно драйвер, т. е. представляет собой старший номер реального устройства, для которого создается клон. Примеры использования такой схемы можно обнаружить для драйверов системы STREAMS, с помощью которых часто реализуются сетевые протоколы и терминальный доступ, включая псевдотерминалы. Это можно заметить, рассмотрев подробный список файлов, отвечающих за эти устройства:

```
$ ls -1
```

crw-rw-rw-	1	root	sys	11, 44	Oct 31	16:36	arp
crw----	1	root	sys	11, 5	Oct 31	16:36	icmp
crw-rw --	1	root	sys	11, 3	Nov 3	1995	ip
crw -----	1	root	sys	11, 40	Nov 3	1995	le
crw-rw-rw-	1	root	sys	11, 42	Oct 31	16:36	tcp
crw-rw-rw-	1	root	sys	11, 41	Nov 3	1995	udp

В данном случае старший номер всех драйверов равен 11 — это драйвер клонов. Если проанализировать информацию файла, скажем, **tcp**, То станет понятно, что старший номер драйвера этого протокола равен 42, для файла **tcp** он представлен младшим номером устройства. Когда процесс открывает этот файл, производится вызов функции **clopen()** драйвера клонов, которой передаются номера устройства. Функция **clopen()** использует младший номер для поиска требуемых точек входа драйвера TCP в коммутаторе устройств **cdevswf[]** После этого **clopen()** вызывает процедуру **xxopen()** драйвера, в данном случае **tcpopen()**, передавая ей указатель на номера устройства и флаг **CLONEOPEN**. В ответ на это **tcpopen()** генерирует неиспользуемый младший номер, создает отдельный логический драйвер (т. е. копирует необходимые структуры данных) и соответствующим образом модифицирует поле **s\_dev** индексного дескриптора файловой системы **specfs**. Таким образом, для получения уникального TCP-соединения процессу нет необходимости самостоятельно производить поиск неиспользуемого младшего номера.

## Встраивание драйверов в ядро

Драйвер устройства является частью кода ядра операционной системы и обеспечивает взаимодействие других подсистем UNIX с физическими или псевдоустройствами. Существует два основных метода встраивания кода и данных драйвера в ядро операционной системы: перекомпиляция ядра, позволяющая статически поместить драйвер, и динамическая загрузка драйвера в ядро в процессе работы системы.

Традиционно для встраивания драйвера в ядро UNIX требуется перекомпиляция ядра и перезапуск системы. Принципиально эта процедура не отличается от компиляции обычной программы, все компоненты ядра являются объектными модулями и редактор связей объединяет их с объектным модулем драйвера для получения исполняемого файла. В этом случае драйвер встраивается в ядро статически, т. е. независимо от фактического наличия устройства и ряда других причин, код и данные драйвера будут присутствовать в ядре UNIX до следующей перекомпиляции.

Однако тенденция развития современных версий операционной системы UNIX заключается в предоставлении возможности динамического расширения функциональности ядра. Это, в частности, относится к файловой системе, драйверам устройств и сетевым протоколам (точнее, драйверам подсистемы STREAMS). Возможность работы с новыми периферийными устройствами без необходимости перекомпиляции ядра обеспечивается загружаемыми драйверами. Вместо того чтобы встраивать модуль драйвера, основываясь на статических таблицах и интерфейсах, ядро содержит набор функций, позволяющих загрузить необходимые драйверы и, соответственно, выгрузить их, когда необходимость работы с данным устройством отпадает. При этом структуры данных для доступа к драйверам устройств также являются динамическими.

Динамическая установка драйвера в ядро операционной системы требует выполнения следующих операций:

- Размещение и динамическое связывание символов драйвера. Эта операция аналогична загрузке динамических библиотек, и выполняется специальным загрузчиком.
  - Инициализация драйвера и устройства.
  - Добавление точек входа драйвера в соответствующий коммутатор устройств.
- П Установка обработчика прерываний драйвера.

Естественно, код динамически загружаемых драйверов сложнее, и содержит, помимо стандартных точек входа, ряд функций, отвечающих за загрузку и выгрузку драйвера, а также ряд дополнительных структур. Пример дополнительных функций и структур данных, которые должны быть определены в динамически загружаемом драйвере операционной системы Solaris 2.5, приведен в табл. 5.2.

**Таблица 5.2.** Дополнительные функции и структуры данных для загружаемых драйверов

<code>_init()</code>	Функция инициализации и установки, вызываемая при загрузке драйвера
<code>_fini()</code>	Функция, вызываемая перед выгрузкой драйвера, удаляющая его из системы

**Таблица 5.2** (продолжение)

<code>_info()</code>	Функция, возвращающая информацию о драйвере по запросу ядра
<code>struct modlinkage</code>	Структура, используемая функциями <code>_init()</code> , <code>_fini()</code> и <code>_info()</code> при загрузке, выгрузке и получении информации о драйвере
<code>struct moddrv</code>	Структура, экспортруемая ядру при загрузке драйвера, в частности, содержит адреса точек входа в драйвер

Помимо этого Solaris 2.5 предоставляет ряд функций ядра для работы с динамически загружаемыми драйверами: `mod_install(9F)`, `mod_remove(9F)` и `mod_info(9F)`.

## Блочные устройства

Драйверы блочных устройств предназначены для обслуживания периферийного оборудования, обеспечивающего обмен данными с помощью фрагментов фиксированной длины, называемыми *блоками*, размер которых значительно превышает один байт. В основном эти драйверы используются файловой подсистемой и подсистемой управления памятью. Например, свопинг характеризуется обменом данными с устройством вторичной памяти, размер которых обычно равен размеру страницы, что составляет 4 или 8 Кбайт. Файловая подсистема производит чтение и запись данных фрагментами, размер которых равен одному или нескольким блокам устройства. Типичными представителями блочных устройств являются жесткий и гибкий диски.

Блочные устройства можно разделить на два типа в зависимости от того, используются ли они для хранения файловой системы или нет. Соответственно различается и схема доступа к этим устройствам. В последнем случае доступ к устройству осуществляется только через специальный файл устройства, представляющий интерфейс низкого уровня. Хотя обращение к устройствам, содержащим файловые системы, может также осуществляться через интерфейс низкого уровня, доступ к таким устройствам, как правило, осуществляется процессом косвенно, через запросы к файловой системе. Например, чтение или запись обычного файла вызывает операции с драйвером блочного устройства (жесткого диска), на котором расположена файловая система, хранящая данный файл. В этом случае обмен данными происходит при активном участии буферного кэша, позволяющего минимизировать число обращений непосредственно к физическому устройству.

Вообще говоря, операции ввода/вывода для блочного устройства могут быть вызваны рядом событий:

- Чтением или записью в обычный файл.
- Чтением или записью непосредственно в специальный файл устройства.
- Операциями подсистемы управления памятью: страничным замещением или свопингом.

Доступ к блочным устройствам осуществляется с помощью трех основных точек входа: `xxopen()`, `xxclose()` и `xxstrategy()`. При этом за фактическое выполнение ввода/вывода отвечает `xxstrategy()`. Единственным аргументом, передаваемым этой функции, является указатель на структуру `buf`, представляющую собой заголовок буфера обмена, с которой мы уже встречались в предыдущей главе при разговоре о буферном кэше. Структура `buf` содержит всю необходимую для операций ввода/вывода информацию. Основные поля структуры `buf`:

<code>b_flags</code>	Флаги. Определяют состояние буфера (например, <code>B_BUSY</code> или <code>B_DONE</code> ) и направление передачи данных ( <code>B_READ</code> , <code>B_WRITE</code> , <code>B_PHYS</code> )
<code>av_back</code> , <code>av_forw</code>	Указатели двухсвязного рабочего списка буферов, ожидающих обработки драйвером
<code>b_bufsize</code>	Размер буфера
<code>b_un.b_addr</code>	Виртуальный адрес буфера
<code>b_blkno</code>	Номер блока начала данных на устройстве
<code>b_bcount</code>	Число байтов, которые требуется передать
<code>b_dev</code>	Старший и младший номера устройства

Использование заголовка `buf` при передачи блока данных показано на рис. 5.7.

Ядро адресует дисковый блок, указывая `vnode` и смещение. Если доступ осуществляется к специальному файлу устройства, то смещение является физическим, отсчитываемым от начала устройства. Например, если специальный файл устройства `/dev/dsk/c0t0d0s1` обеспечивает доступ ко второму разделу жесткого диска, то смещение будет отсчитываться от начала этого раздела. Если `vnode` представляет обычный файл, то смещение является логическим, отсчитываемым от начала файла.

Таким образом, блок устройства, содержащего файловую систему, может быть адресован двумя способами — либо через обычный файл и логическое смещение, либо через специальный файл устройства и физическое смещение на этом устройстве. Это, в свою очередь, может привести к различной идентификации одного и того же блока и, как следствие, двум различным копиям блока в памяти. Результатом такого несоответствия может стать потеря или нарушение целостности данных. Поэтому непосредственный доступ к специальному файлу такого устройства возможен только при размонтированной файловой системе.

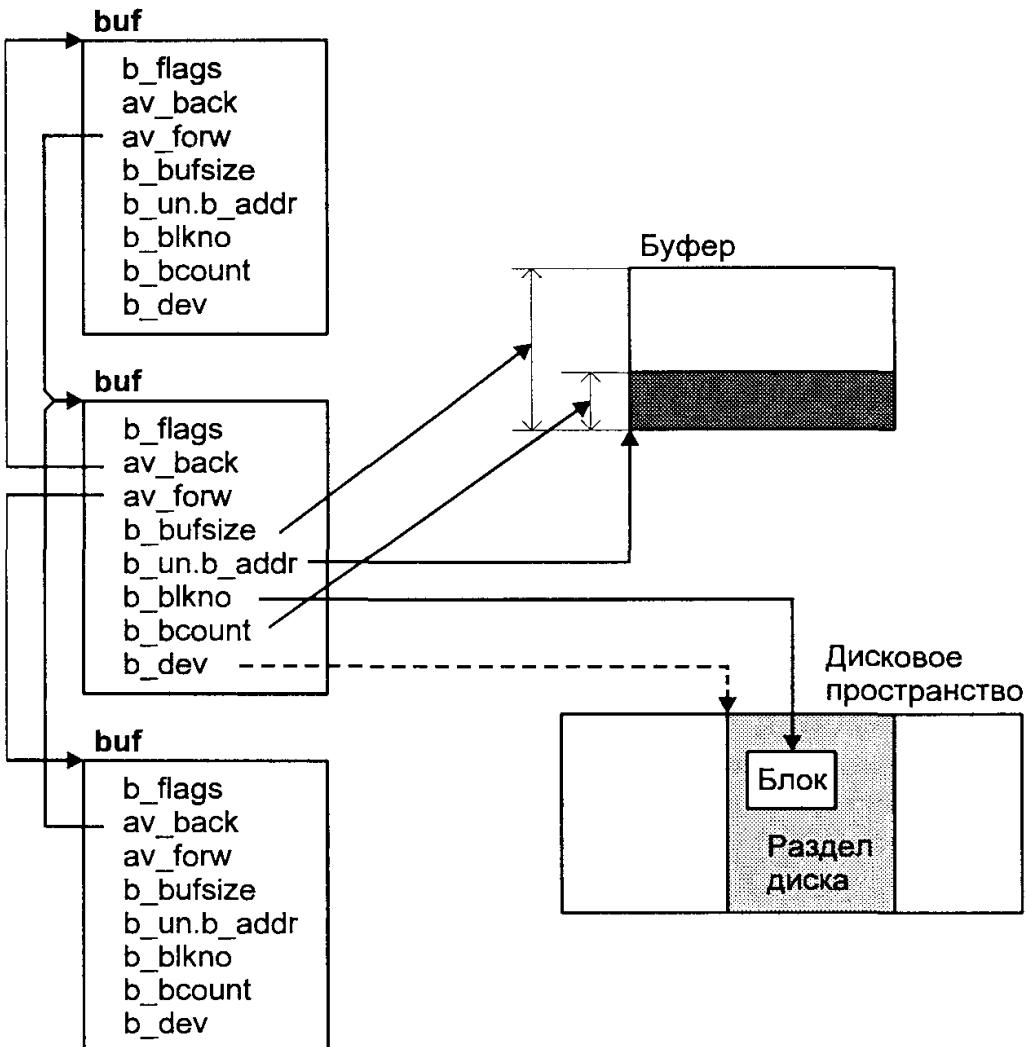


Рис. 5.7. Обмен данными с блочным устройством (диском)

Поскольку каждый дисковый блок связан с каким-либо файлом и соответственно с его vnode, а его образ в памяти — с физическими страницами, которые также связаны с vnode (через структуры описания физической памяти — page в SVR4, pfdat в SVR3), все операции ввода/вывода связаны с подкачкой и сохранением страниц и идентифицируются vnode.

## Символьные устройства

Символьные устройства представляют собой значительную часть периферийного оборудования системы, включая терминалы, манипуляторы (например, мышь), клавиатуру и локальные принтеры. Основное отличие этих устройств от блочных заключается в том, что они, как правило, передают небольшие объемы данных.

Обмен данными с символьными устройствами происходит непосредственно через драйвер, минуя буферный кэш. При этом данные обычно копи-

руются в драйвер из адресного пространства процесса, запросившего операцию ввода/вывода.

Если процесс сделал системный вызов ввода/вывода, например, `read(2)` или `write(2)` со специальным файлом символьного устройства, запрос направляется в файловую подсистему. Поскольку доступ к устройству обслуживается файловой системой `specfs`, рассмотренной ранее, в ответ на выполнение системного вызова процесса ядро выполняет вызов функции `spec_read()` или `spec_write()` соответственно для `read(2)` или `write(2)`. Действия функций `spec_read()` и `spec_write()` похожи. Обе проверяют тип `vnode` и определяют, что устройство является символьным. После этого с помощью коммутатора ядро выбирает соответствующую точку входа драйвера, используя старший номер, хранящийся в поле `v_rdev vnode`, и вызывает эту функцию (соответственно `xxread()` или `xxwrite()`), передавая ей в качестве параметров старший и младший номера, ряд дополнительных параметров, зависящих от конкретного вызова, а также явно или неявно адресует область копирования данных в адресном пространстве процесса<sup>3</sup>.

## Интерфейс доступа низкого уровня

Символьные драйверы обеспечивают доступ не только к символьным устройствам, например, к адаптеру последовательного или параллельного портов, манипулятору "мышь", монитору или терминалам. Часть символьных драйверов служит в качестве *интерфейса доступа низкого уровня* к блочным устройствам, таким как диски или накопители на магнитных лентах.

Большинство таких драйверов отличаются от соответствующих им драйверов блочных устройств характером выполнения операций ввода/вывода. В то время как драйверы блочных устройств производят обмен данными с буферным кэшем, драйверы доступа низкого уровня обеспечивают обмен данных непосредственно с адресным пространством процесса. Отсутствие посредника в виде буферного кэша устраняет необходимость в совершении дополнительных операций копирования (драйвер — буферный кэш — буфер процесса), но в то же время лишает процесс услуг кэширования данных, предоставляемых операционной системой.

Интерфейс доступа низкого уровня используется многими системными утилитами обслуживания файловой системы, например, `fsck(1M)`, а также рядом приложений, работающих с накопителями на магнитной ленте, например `tar(1)` или `cpio(1)`. Этот интерфейс используется некоторыми при-

<sup>3</sup> Несколько иная схема применяется для драйверов подсистемы STREAMS, которые также имеют символьный интерфейс доступа. Эти драйверы будут рассматриваться в данной главе в разделе "Подсистема STREAMS".

ложениями, например СУБД, которые самостоятельно обеспечивают оптимизированные механизмы кэширования данных на уровне задачи.

Поскольку драйверы низкого уровня не используют буферный кэш, они самостоятельно обеспечивают необходимые буфера для совершения операции ввода/вывода. На рис. 5.8 показаны отличия в характере выполнения операции ввода/вывода с блочными устройствами в случаях, когда запрос формируется при участии буферного кэша (драйверы блочных устройств), и когда манипуляция буфером производится драйвером самостоятельно (драйверы низкого уровня).

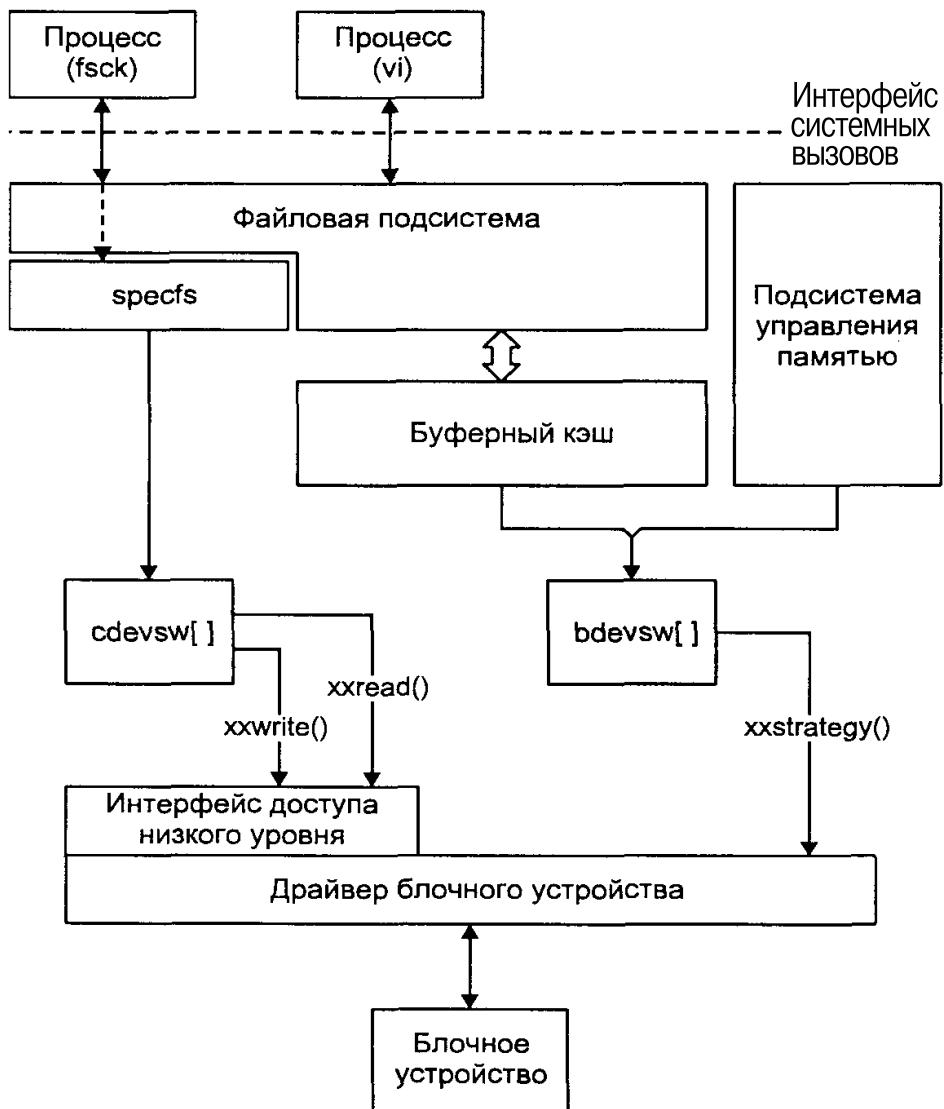


Рис. 5.8. Различные типы доступа к блочным устройствам

## Буферизация

Очевидно, что побайтная передача данных между драйвером символьного устройства и прикладным процессом весьма неэффективна. При таком режиме работы байт должен быть сначала скопирован в адресное про-

странство драйвера, затем некоторое время должно пройти, прежде чем драйвер сможет передать этот символ физическому устройству. Если при этом устройство оказывается занятым, процесс должен ожидать завершения предыдущей операции, что, скорее всего, вынудит его перейти в состояние сна и приведет к переключению контекста.

Существует несколько способов преодолеть данную ситуацию, но все они предполагают обеспечение некоторой буферизации данных драйвером устройства. Первый способ заключается в использовании прерываний, когда при поступлении на устройство следующего символа, генерируется аппаратное прерывание, которое обрабатывается функцией `xxintr()` драйвера независимо от функции `xxwrite()`. Функция обработки прерывания записывает данные в буфер, которые затемчитываются функцией `xxwrite()`.

Если устройство не поддерживает прерываний, их поступление можно симулировать с помощью функции `xxroll()` драйвера устройства, которая вызывается ядром через определенные промежутки времени (обычно каждый сигнал таймера). Обычно функция `xxroll()`, в свою очередь, вызывает функцию `xxintr()`, скажем, на каждый десятый сигнал таймера, обеспечивая тем самым независимое считывание и буферизацию данных.

Буферизация данных для символьных устройств осуществляется с помощью специальных структур данных, называемых `clist`. Каждая структура `clist` имеет следующие поля:

int	<code>c_cc</code>
<code>struct cblock</code>	<code>*c_cf</code>
<code>struct cblock</code>	<code>*c_cl</code>

Поле `c_cc` содержит число символов в буфере `cblock`. Поля `c_cf` и `c_cl` указывают, соответственно, на первый и последний элементы `cblock`, организованные в виде связанного списка и фактически обеспечивающие буфера хранения данных. Каждая структура `cblock` может хранить несколько символов. Когда буфер хранения заполняется, ядро автоматически выделяет новую структуру `cblock` и помещает ее в связанный список. Поля структуры `cblock` и их использование приведены на рис. 5.9.

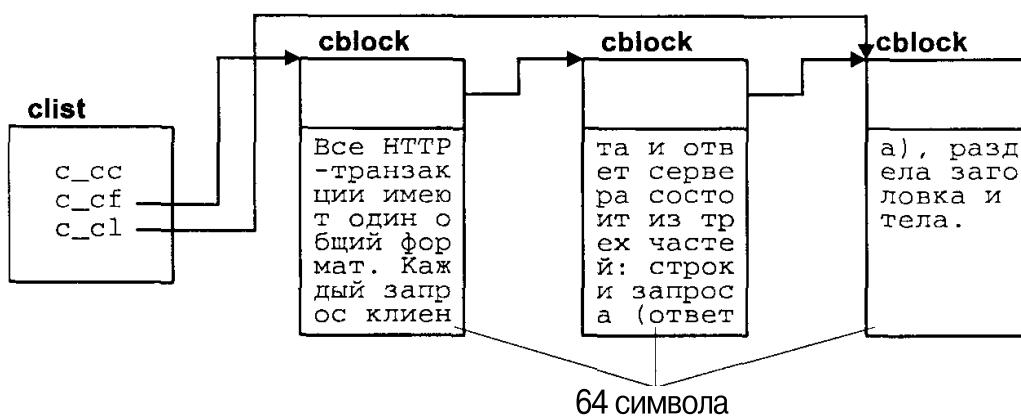


Рис. 5.9. Буферизация данных с помощью `clist`

Пример буферизации с использованием структуры `clist` в драйвере терминала показан на рис. 5.10.

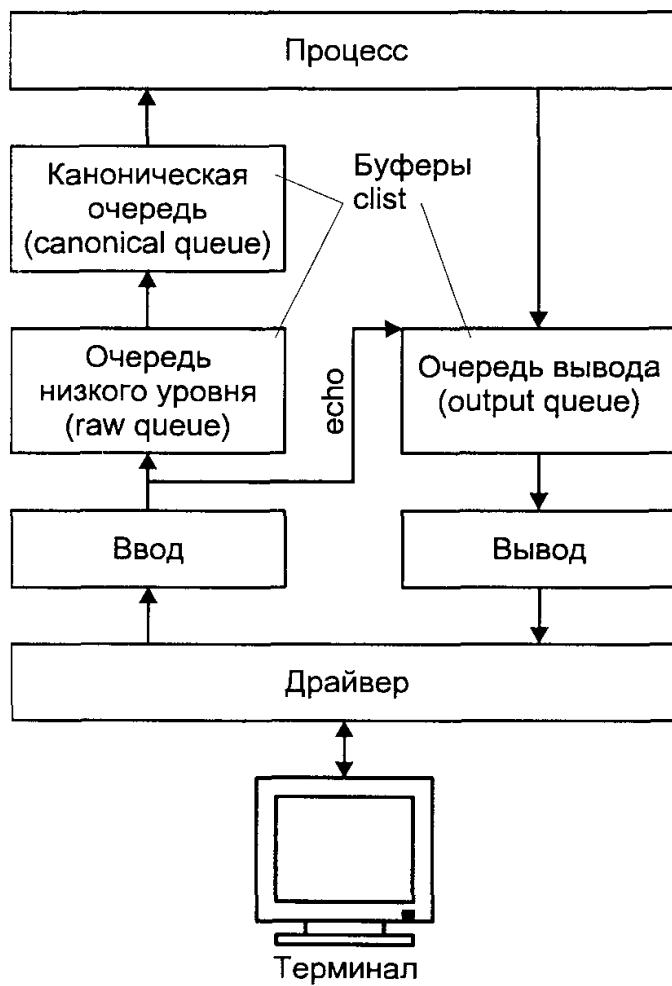


Рис. 5.10. Пример использования буферов `clist` в драйвере терминала

## Архитектура терминального доступа

Алфавитно-цифровой терминал — последовательное устройство, и операционная система производит обмен данными с терминалом через последовательный интерфейс, называемый *терминальной линией*. С каждой терминальной линией в UNIX ассоциирован специальный файл символьного устройства `/dev/ttymx4`.

Терминальные драйверы выполняют ту же функцию, что и остальные драйверы: управление передачей данных от/на терминалы. Однако терми-

В зависимости от версии UNIX вместо символов `xx` в имени файла терминала присутствует идентификатор, позволяющий поставить в соответствии специальному файлу конкретную терминальную линию. Например, в SCO UNIX виртуальные экраны системного монитора имеют имена `/dev/tty01`, `/dev/tty02` и т. д.

налы имеют одну особенность, связанную с тем, что они обеспечивают интерфейс пользователя с системой. Обеспечивая интерактивное использование системы UNIX, терминальные драйверы имеют свой внутренний интерфейс с модулями, интерпретирующими ввод и вывод строк. Модуль, отвечающий за такую обработку, называется *дисциплиной линии* (line discipline).

Существует два режима терминального ввода/вывода:

1. Канонический режим. В этом режиме ввод с терминала обрабатывается в виде законченных строк.
2. Неканонический режим, при котором ввод не интерпретируется.

В каноническом режиме интерпретаторы строк преобразуют неструктурированные последовательности данных, введенные с клавиатуры, в каноническую форму (то есть в форму, соответствующую тому, что пользователь имел в виду на самом деле) прежде, чем послать эти данные принимающему процессу. Например, программисты работают на клавиатуре терминала довольно быстро, но иногда допускают ошибки. На этот случай имеется клавиша стирания, и пользователь имеет возможность удалять часть введенной строки и вводить корректиды. Драйвер терминала получает всю введенную последовательность, включая и символы стирания. В каноническом режиме модуль дисциплины линии буферизует информацию в строку (набор символов, заканчивающийся символом возврата каретки) и стирает символы в буфере, прежде чем переслать исправленную последовательность считывающему процессу. В таком режиме, например, работает командный интерпретатор shell.

В режиме без обработки строковый интерфейс передает данные между процессами и терминалом без каких-либо преобразований. Например, текстовый редактор работает с драйвером в неканоническом режиме, благодаря чему любой символ, введенный пользователем интерпретируется самим процессом.

В функции модуля дисциплины линии входят:

1. Построчный разбор введенных последовательностей.
2. Обработка символов стирания.
3. Обработка символов удаления, отменяющих всех предыдущих символов.
4. Отображение символов, полученных терминалом.
5. Расширение выходных данных, например, преобразование символов табуляции в последовательности пробелов.
6. Предоставление возможности не обрабатывать специальные символы, такие как символы стирания, удаления и возврата каретки.

Существует дополнительная возможность обработки данных, получаемых и передаваемых устройству — отображение вводимых и выводимых символов

в символы, определенные *таблицей отображения*. Данную возможность поддерживает утилита *mapchan*.

## Псевдотерминалы

Псевдотерминалы являются специальным устройством, эмулирующим стандартную терминальную линию. Псевдотерминалы напоминают каналы как средство межпроцессного взаимодействия, позволяющее двум процессам обмениваться данными. Однако в отличие от каналов, псевдотерминалы обеспечивают дополнительную функциональность, специфичную для терминальных линий. Схематическая архитектура псевдотерминала представлена на рис. 5.11.

Ярким примером использования псевдотерминалов является регистрация в системе по сети с использованием серверов удаленного доступа *rlogin(1)* или *telnet(1)*, или использование графического эмулятора терминала *xterm* в системе X Window System. Когда пользователь регистрируется в системе подобным образом, псевдотерминал эмулирует обычную терминальную линию, поэтому пользователь не видит различия между удаленной и локальной работой с помощью терминала, подключенного по последовательной линии. Например, пользователь может установить различные режимы обработки и использовать соответствующие комбинации клавиш для генерации сигналов, как он это делает в случае обычного терминала.

Псевдотерминал по существу представляет собой два отдельных драйвера. Один из них выглядит как обычный терминальный драйвер и носит название *подчиненного устройства* (slave). Второй драйвер называется *основным* (master).

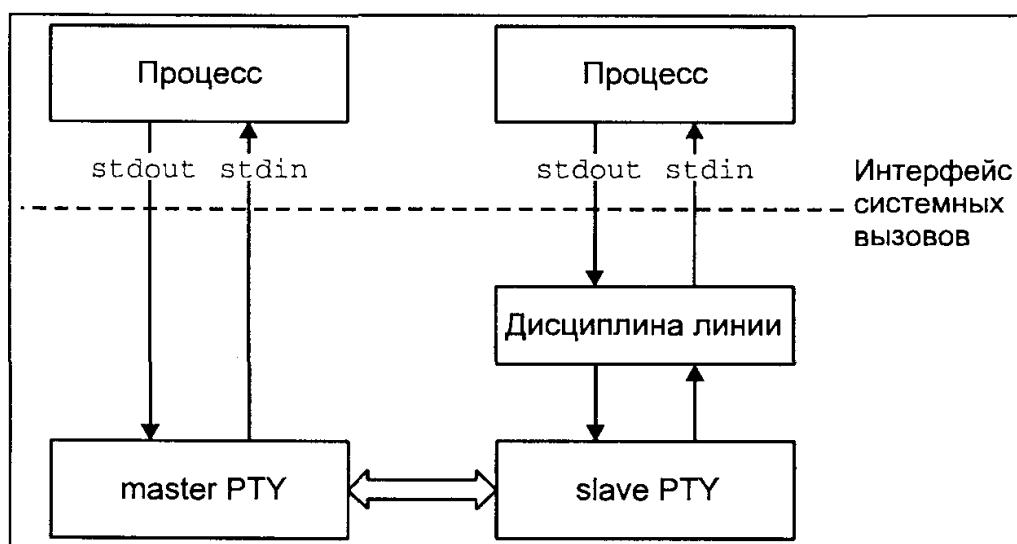


Рис. 5.11. Взаимодействие процессов с помощью псевдотерминала

Поскольку подчиненное устройство имеет все характеристики терминала, процесс может связать свои стандартные потоки ввода, вывода и вывода ошибок с этим устройством. Однако в отличие от обычного терминала, в случае которого запись процесса приводит к отображению данных на физическом устройстве, а ввод данных пользователем с клавиатуры может быть получен чтением терминальной линии, все данные, записанные в подчиненное устройство, передаются основному и наоборот — почти так, как работает канал. Однако модуль дисциплины линии позволяет обеспечить дополнительные возможности этого канала, которые могут потребоваться некоторым приложениям, например, командному интерпретатору *shell*.

В качестве иллюстрации использования псевдотерминала, рассмотрим схему работы в режиме командной строки пользователя, находящегося на некоторой удаленной системе в сети.

Пользователь удаленной системы запускает программу удаленного доступа *rlogin(1)*, которая формирует запрос и передает его по сети на требуемый компьютер. Там этот запрос доставляется серверу удаленного доступа *rlogind(1)*, который (после надлежащей проверки) запускает программу *login(1)*. При этом стандартные потоки ввода, вывода и вывода ошибок программы *login(1)* связываются не с терминальным файлом, как в случае входа в систему с помощью сервера *getty(1M)*, а с подчиненным устройством псевдотерминала. Основное же устройство оказывается связанным с сервером *rlogind(1)*. Программа *login(1)* запрашивает имя пользователя и его пароль точно так же, как она это делает при входе через *getty(1M)*. Более того, *login(1)* и "не представляет", что на самом деле работает с эмулятором терминала, а не с традиционной терминальной линией. Весь ввод *login(1)* поступает серверу *rlogind(1)* и затем передается по сети клиентской части *rlogin(1)* на удаленном компьютере. Далее работа ничем не отличается от работы локального пользователя, подключенного к системе с помощью обычновенного терминала или консоли. Если имя пользователя и пароль были введены правильно, программа *login(1)* запустит требуемый командный интерпретатор (*login shell*), который также не заметит подмены. Действительно, по всем характеристикам терминал будет неотличим от традиционной последовательной линии, включая различные установки и генерацию сигналов при нажатии определенных клавиш клавиатуры. Следует, правда, оговориться, что поскольку псевдотерминал не является "полноценным" терминальным устройством, часть установок для него не имеют смысла (например, скорость передачи, четность и т. д.) и просто игнорируются.

На рис. 5.12 приведена схема работы удаленного пользователя в системе с использованием псевдотерминала.

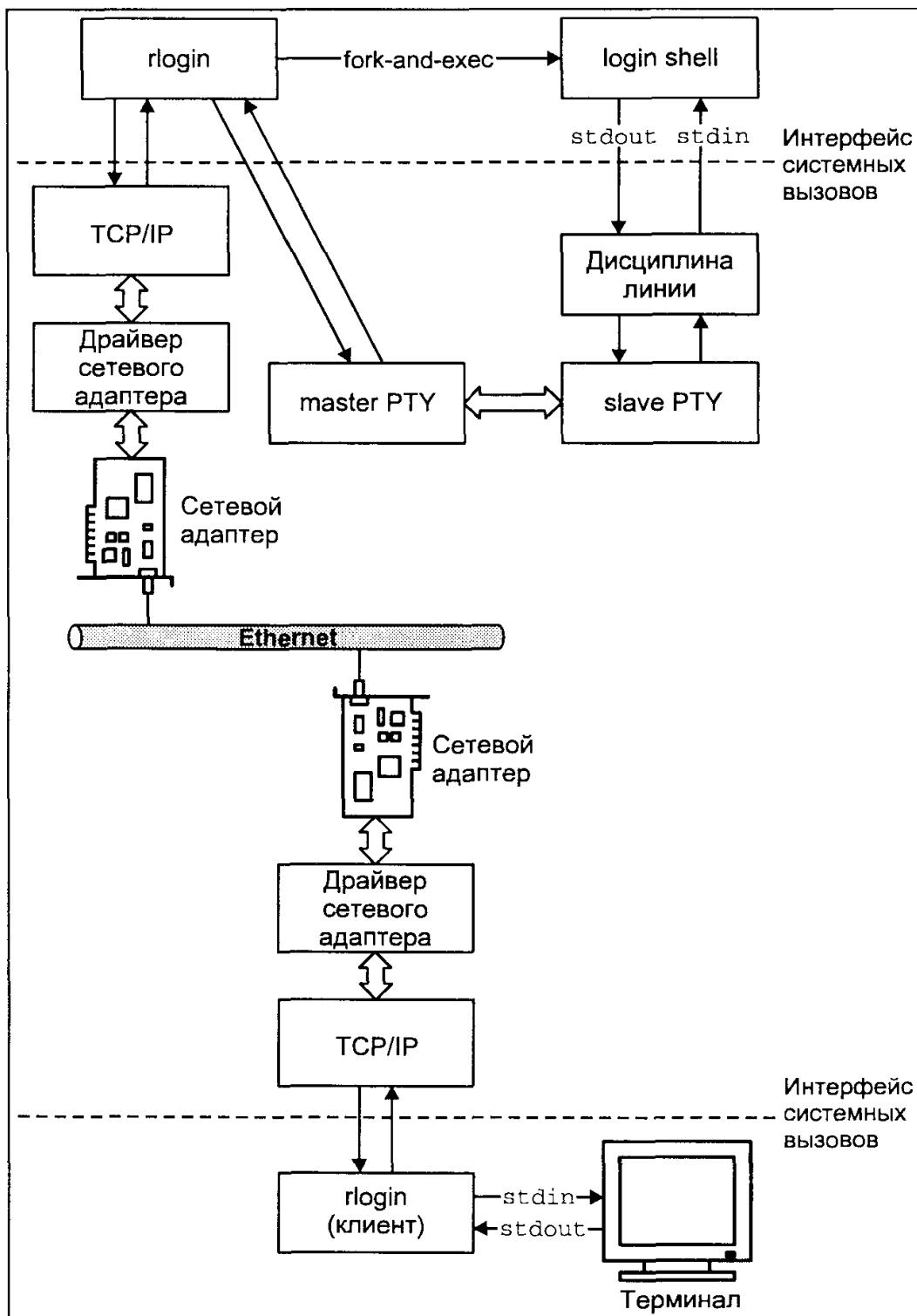


Рис. 5.12. Архитектура удаленного доступа с использованием псевдотерминала

## Подсистема STREAMS

Архитектура подсистемы потокового ввода/вывода STREAMS впервые была описана в статье "Потоковая система ввода/вывода" (Ritchie, D. M.,

"A Stream Input-Output System", AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Oct. 1984) в 1984 году. Двумя годами позднее эта система была реализована в коммерческой версии UNIX SVR3.

Поводом для создания новой архитектуры ввода/вывода послужили несколько обстоятельств.

Традиционная система ввода/вывода, ориентированная на посимвольную передачу данных и рассмотренная ранее в этой главе, была изначально предназначена для работы с ограниченным числом низкоскоростных асинхронных терминальных устройств. Операционная система взаимодействует с такими устройствами (через точки входа в драйвер) на достаточно высоком уровне, возлагая основную обработку данных на драйвер. При этом только часть кода драйвера аппаратно зависима. Остальная обработка может являться однотипной для широкого спектра периферийного оборудования. По мере роста числа поддерживаемых операционной системой устройств использование стандартной архитектуры подсистемы ввода/вывода приводило к существенным накладным расходам, в частности, к неоправданному дублированию кода в ядре UNIX.

Другой побудительной причиной для разработки новой подсистемы ввода/вывода явилось отсутствие стандартного механизма буферизации данных для символьных устройств. По мере увеличения скоростей передачи, посимвольная обработка и передача стала неэффективной. Поэтому был разработан ряд подходов для обеспечения буферизации, например использование механизма, основанного на структуре `clist`, рассмотренного нами ранее. Однако такие схемы, по-прежнему обладая невысокой производительностью, по существу возлагают буферизацию данных на драйвер, что приводит к неэффективному распределению памяти.

Наконец, необходимость поддержки сетевых протоколов, большинство из которых имеют уровневую организацию, требует соответствующей архитектуры подсистемы ввода/вывода. Передача сетевых данных производится в виде пакетов или сообщений, при этом каждый уровень сетевого протокола производит определенную обработку и передает их другому уровню. Каждый уровень имеет стандартные интерфейсы взаимодействия с другими (верхним и нижним уровнями) и при этом может работать с различными протоколами верхнего и нижнего уровней. Например, протокол IP (уровень 3 модели OSI<sup>5</sup>) может поддерживать работу нескольких протоколов верхнего уровня: TCP и UDP. На нижнем уровне протокол IP также взаимодействует с несколькими протоколами, обеспечивая передачу данных через различные сетевые интерфейсы (например, Ethernet, Token Ring

Модель OSI иерархии сетевых протоколов, предложенная Международной организацией по стандартам (ISO), включает определение функциональности для 7 уровней. Различные семейства протоколов, например TCP/IP или SNA, имеют то или иное отображение на эту модель. Эти вопросы рассмотрены в главе 6.

или последовательный канал). Такая организация сетевых протоколов предполагает иерархическую структуру подсистемы ввода/вывода, когда драйверы являются объединением независимых модулей.

Подсистема STREAMS в большой степени призвана решить эти задачи. Она предоставляет интерфейс обмена данными, основанный на сообщениях, и обеспечивает стандартные механизмы буферизации, управления потоком данных и различную приоритетность обработки. В STREAMS дублирование кода сводится к минимуму, поскольку однотипные функции обработки реализованы в независимых модулях, которые могут быть использованы различными драйверами. Сам драйвер обеспечивает требуемую функциональность, связывая в цепочку один или несколько модулей, подобно тому как программный канал позволяет получить новое качество обработки, связав несколько независимых утилит.

Сегодня подсистема STREAMS поддерживается большинством производителей операционных систем UNIX и является основным способом реализации сетевых драйверов и модулей протоколов. Использование STREAMS охватывает и другие устройства, например терминальные драйверы в UNIX SVR4.

## Архитектура STREAMS

Подсистема STREAMS обеспечивает создание *потоков* — полнодуплексных каналов между прикладным процессом и драйвером устройства<sup>6</sup>. С другой стороны, архитектура STREAMS определяет интерфейсы и набор правил, необходимых для взаимодействия различных частей этой системы и для разработки модульных драйверов, обеспечивающих такое взаимодействие и обработку.

На рис. 5.13 показана общая архитектура коммуникационного канала между процессом и драйвером STREAMS. Сам поток полностью располагается в пространстве ядра, соответственно и все функции обработки данных выполняются в системном контексте. Типичный поток состоит из головного модуля, драйвера и, возможно, одного или более модулей. Головной модуль взаимодействует с прикладными процессами через интерфейс системных вызовов. Драйвер, замыкающий поток, взаимодействует непосредственно с физическим устройством или псевдоустройством, в качестве которого может выступать другой поток. Модули выполняют промежуточную обработку данных.

Процесс взаимодействует с потоком, используя стандартные системные вызовы *open(2)*, *close(2)*, *read(2)*, *write(2)* и *ioctl(2)*. Дополнительные функ-

<sup>6</sup> Потоковый драйвер (драйвер STREAMS) имеет архитектуру, отличную от архитектуры драйверов символьных устройств, рассмотренных ранее.

ции работы с потоками включают *poll(2)*, *putmsg(2)* и *getmsg(2)*. Передача данных по потоку осуществляется в виде *сообщений*, содержащих данные, тип сообщения и управляющую информацию. Для передачи данных каждый модуль, включая головной модуль и сам драйвер, имеет две очереди — *очередь чтения* (read queue) и *очередь записи* (write queue). Каждый модуль обеспечивает необходимую обработку данных и передает их в очередь следующего модуля. При этом передача в очередь записи осуществляется вниз по потоку (downstream), а в очередь чтения — вверх по потоку (upstream). Например, на рис. 5.13 из очереди записи модуля 2 сообщение может быть передано в очередь записи модуля 1, но не наоборот. В свою очередь сообщение из очереди чтения модуля 2 передается в очередь чтения головного модуля, который далее передает данные процессу в ответ на системный вызов *read(2)*. Когда процесс выполняет системный вызов *write(2)*, данные передаются головному модулю и далее вниз по потоку.

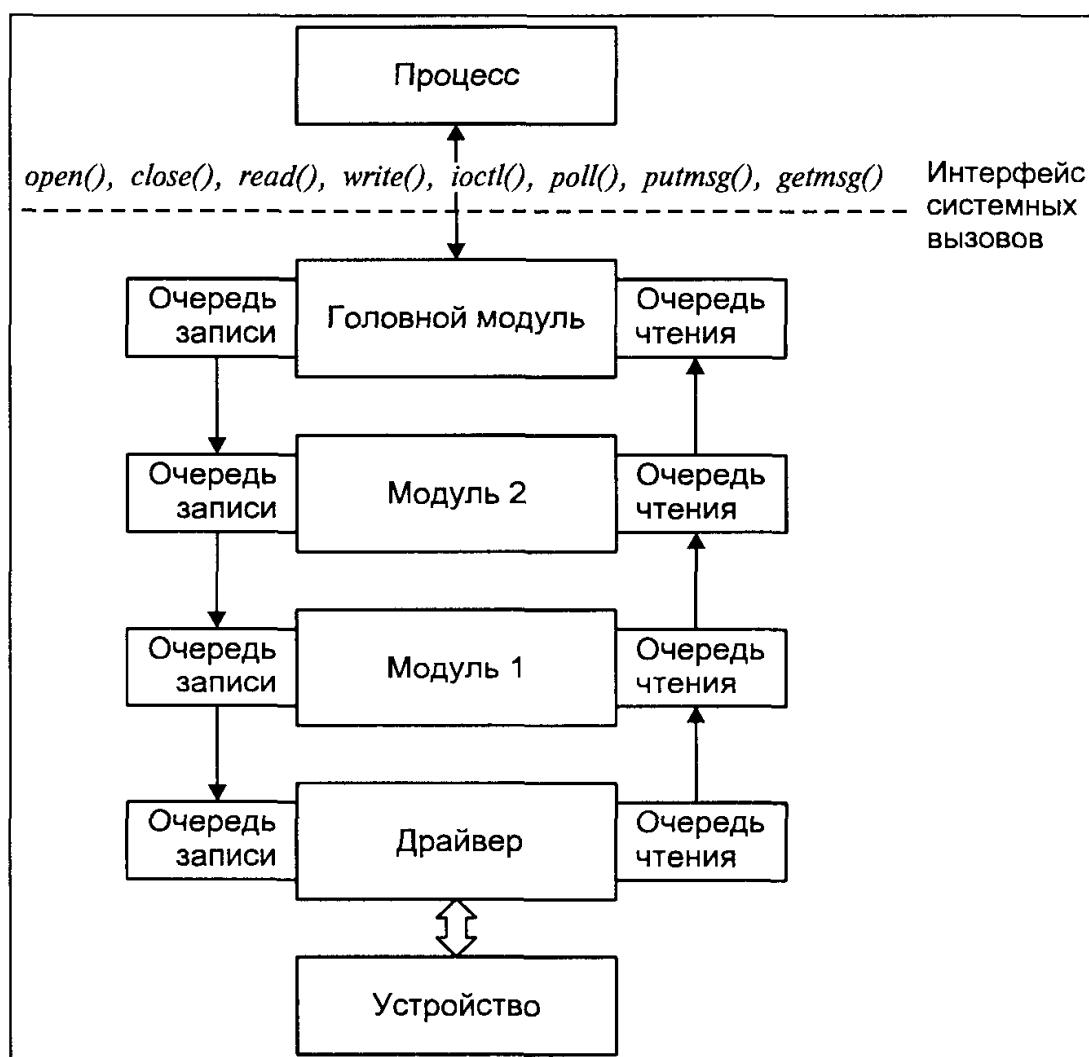


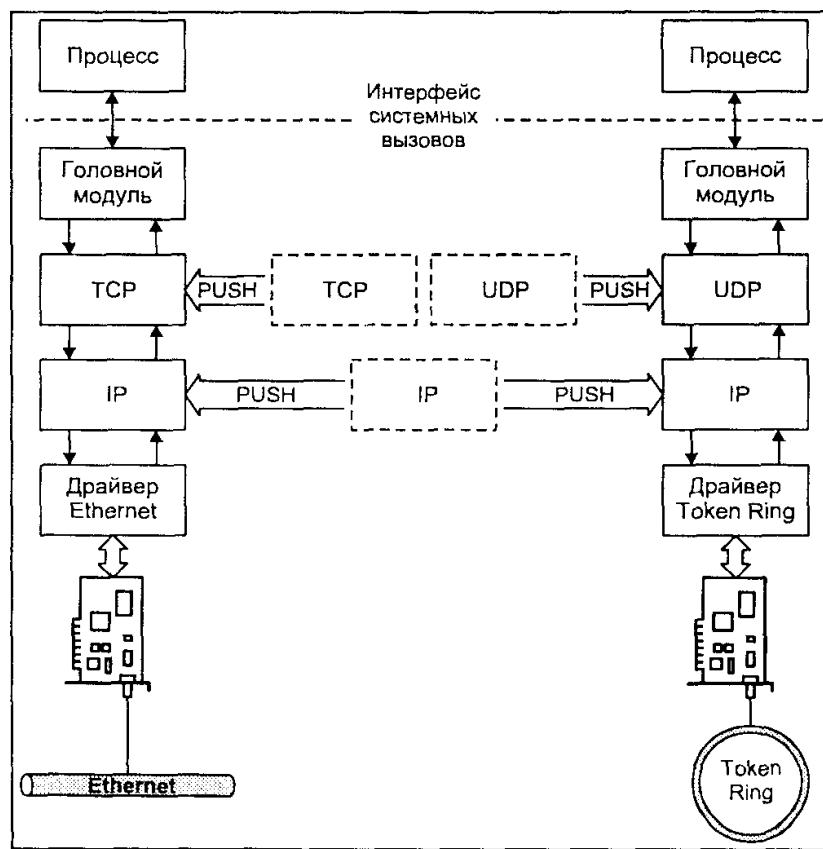
Рис. 5.13. Базовая архитектура потока

Сообщения также могут передаваться в *парную очередь*. Другими словами, из очереди записи модуля 1 сообщение может быть направлено в очередь

чтения того же модуля, а затем, при необходимости, передано вверх по потоку. При этом модулю нет необходимости знать, какой части потока принадлежит следующая очередь — головному или промежуточному модулю, или драйверу. Такой подход позволяет производить разработку модулей независимо друг от друга и использовать их затем в различных комбинациях и в различных потоках.

Подсистема STREAMS обеспечивает возможность такой комбинации благодаря механизму *динамического встраивания* (push) модуля в поток. Встраивание модуля возможно непосредственно после головного модуля. При этом будут установлены связи между соответствующими очередями встраиваемого модуля, головного модуля и модулей вниз по потоку. После этого встроенный модуль будет производить определенную обработку проходящих данных, тем самым изменяя изначальную функциональность потока. При необходимости модуль может быть *извлечен* (pop) из потока.

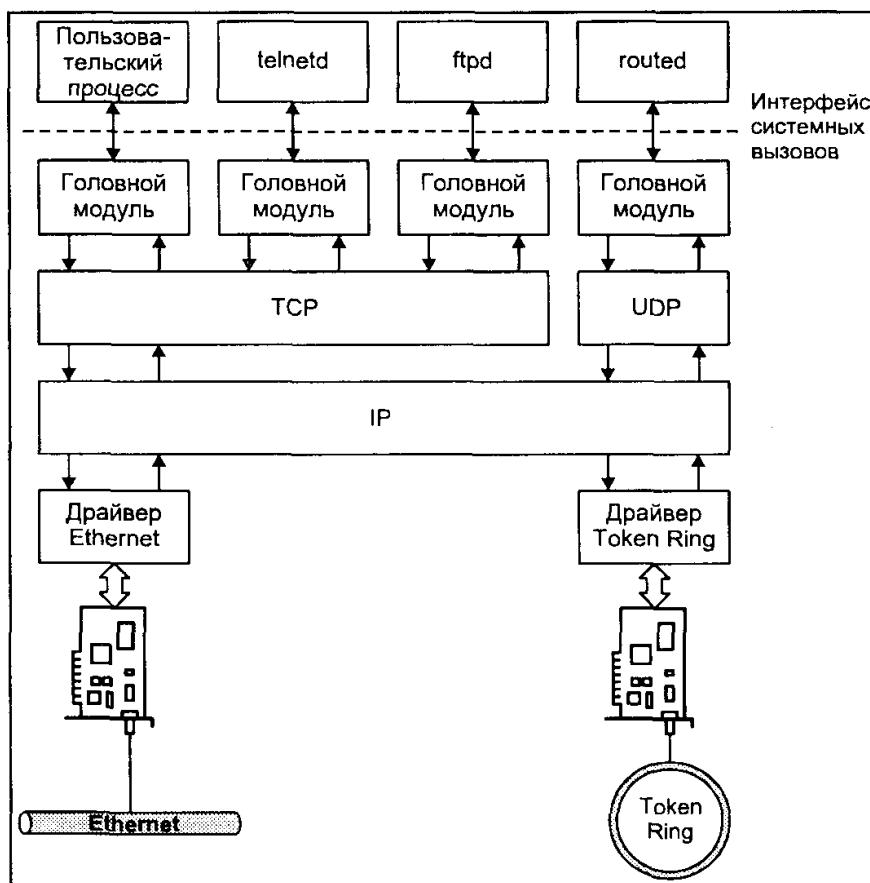
На рис. 5.14 показаны различные потоки, созданные из нескольких стандартных компонентов, для поддержки сетевых протоколов семейства TCP/IP. Причем модули IP, TCP и UDP могут поставляться одним производителем, а драйверы Ethernet или Token Ring — соответствующими производителями сетевых адаптеров. В результате встраивания необходимых модулей первый поток будет обеспечивать передачу трафика TCP через адаптер Ethernet, в то время как второй — передачу трафика UDP через адаптер Token Ring.



**Рис. 5.14.** Использование одних и тех же модулей для создания различных потоков

Подсистема STREAMS также обеспечивает возможность *мультиплексирования потоков*. Мультиплексирующий драйвер может быть подключен к нескольким модулям как вверх, так и вниз по потоку. Различают три типа мультиплексоров — *верхний*, обеспечивающий мультиплексирование вверх по потоку, *нижний*, обеспечивающий мультиплексирование вниз по потоку, и *гибридный*, поддерживающий несколько потоков выше и ниже мультиплексора.

С помощью мультиплексирующих драйверов потоки, представленные на рис. 5.14, могут быть объединены в единый *драйвер протоколов*, поддерживающий несколько каналов передачи данных. Именно таким образом реализована поддержка сети во многих версиях операционной системы UNIX. Возможная организация компонентов STREAMS приведена на рис. 5.15.



**Рис. 5.15.** Конфигурация сетевого доступа с использованием подсистемы STREAMS

В этом случае модули TCP и UDP являются верхними мультиплексорами, а модуль IP реализован в виде гибридного мультиплексора<sup>7</sup>. Такая организация позволяет приложениям создавать потоки, используя различные комбинации сетевых протоколов и драйверов сетевых устройств. Задача

<sup>7</sup> На самом деле мультиплексором может являться только драйвер STREAMS. Объединение драйверов в единый объект отлично от встраивания модулей и носит название *связывания*. Более подробно связывание и различия между модулями и драйверами STREAMS мы рассмотрим несколько позже в этой главе.

мультиплексирующего драйвера помимо обработки данных заключается в хранении состояния всех потоков и правильной маршрутизации данных между ними, т. е. передаче данных в очередь требуемого модуля.

## Модули

Модули являются основными компонентами потока. Каждый модуль состоит из пары очередей — очереди чтения и записи, а также набора функций, осуществляющих обработку данных и их передачу вверх или вниз по потоку. Архитектура модуля представлена на рис. 5.16.

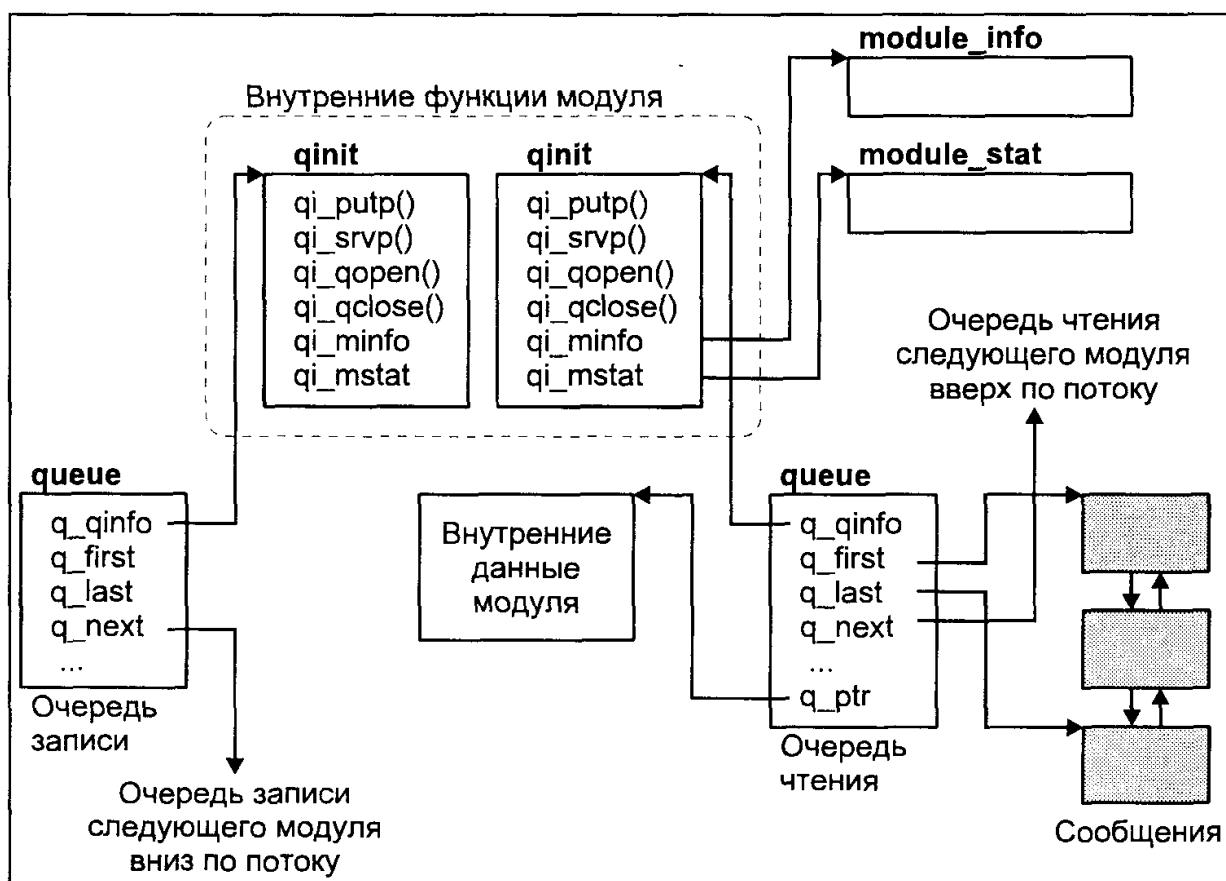


Рис. 5.16. Модуль STREAMS

Каждая очередь представлена структурой данных `queue`. Наиболее важными полями `queue` являются:

- |  |   |
|--|---|
| <code>q_qinfo</code>                       | Указатель на структуру <code>qinit</code> , описывающую функции обработки сообщений данной очереди. |
| <code>q_first</code> , <code>q_last</code> | Указатели на связанный список сообщений, ожидающих передачи вверх или вниз по потоку.               |
| <code>q_next</code>                        | Указатель на очередь следующего модуля вверх или вниз по потоку.                                    |
| <code>q_ptr</code>                         | Указатель на внутренние данные модуля (очереди).  |

Помимо указанных полей, структура `queue` содержит параметры для обеспечения управления потоком данных — *верхнюю* и *нижнюю* *ватерлини* очереди.

Передача данных вверх или вниз по потоку осуществляется с помощью функций модуля, указатели на которые хранятся в структуре `qinit`. Модуль должен определить четыре процедуры для обработки каждой из очередей: `xxrput()`, `xxservice()`, `xxopen()` и `xxclose()`, где `xx`, как и прежде, обозначает уникальный префикс драйвера. Эти функции адресуются указателями `(*qi_rputp)()`, `(*qi_srvp)()`, `(*qi_qopen)()`, `(*qi_qclose)()`. Этих четырех функций достаточно для взаимодействия с соседними модулями, обработки и передачи данных. Функция `xxopen()` вызывается каждый раз, когда процесс открывает поток или при встраивании модуля. Соответственно функция `xxclose()` вызывается при закрытии потока или извлечении модуля. Функция `xxrput()` осуществляет обработку сообщений, проходящих через модуль. Если `xxrput()` не может передать сообщение следующему модулю (например, в случае, если очередь следующего модуля переполнена), она помещает сообщение в собственную очередь. Периодически ядро вызывает процедуру `xxservice()` каждого модуля для передачи отложенных сообщений.

Модуль должен иметь функцию `xxrput()` для каждой очереди. Функция `xxservice()` может не существовать, в этом случае `xxrput()` не имеет возможности отложить передачу сообщения и должна передать его немедленно, даже если очередь следующего модуля переполнена. Таким образом модули, не имеющие процедуры `xxservice()`, не обладают возможностью управления потоком данных. Эти аспекты мы подробнее рассмотрим в следующих разделах.

Оставшиеся поля структуры `qinit`:

<code>module_info</code>	В этой структуре хранятся базовые значения таких параметров, как ватерлини, размер сообщений и т. д. Некоторые из этих параметров также находятся в структуре <code>queue</code> . Это дает возможность динамически изменять их, сохраняя при этом базовые значения.
<code>module_stat</code>	Эта структура непосредственно не используется подсистемой STREAMS. Однако модуль имеет возможность осуществлять сбор разнообразной статистики своего участка потока с помощью полей этой структуры.

## Сообщения

В подсистеме STREAMS все данные передаются в виде *сообщений*. С помощью сообщений передаются данные от приложений к драйверу и обратно. Сообщения используются для взаимодействия модулей между собой. Модули могут также генерировать сообщения для уведомления при-

кладного процесса или друг друга о возникновении ошибок или непредвиденных ситуаций. Таким образом, сообщения являются единственным способом передачи информации между различными компонентами потока и потому занимают ключевое место в подсистеме STREAMS.

Сообщение описывается двумя структурами данных: *заголовком сообщения* msgb (message block) и *заголовком блока данных* datab (data block). Обе эти структуры адресуют буфер данных, где находятся фактические данные сообщения.

Заголовок сообщения msgb имеет следующие поля:

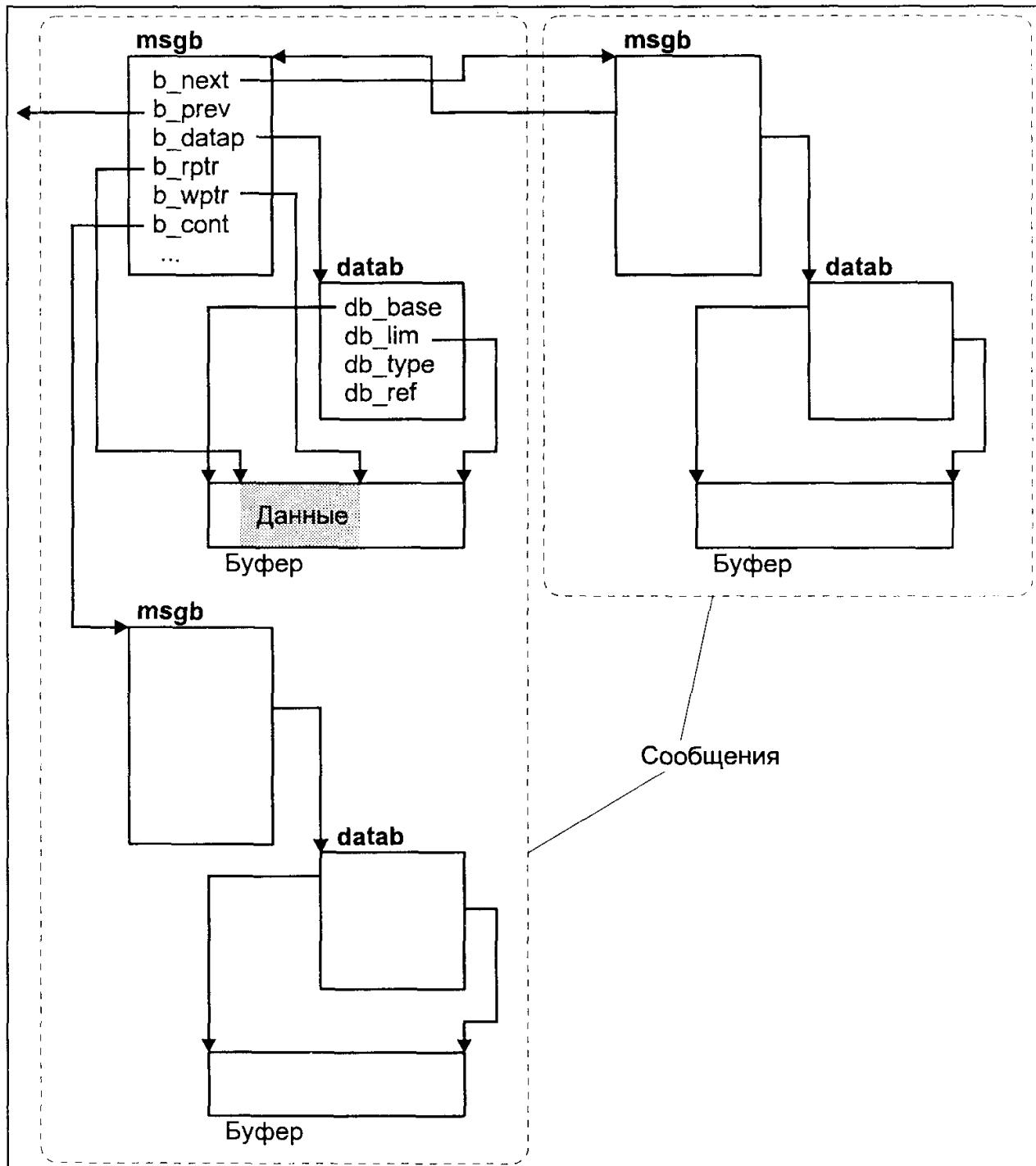
<code>b_next, b_prev</code>	Используются для формирования связанного списка сообщений и соответственно адресуют следующее и предыдущее сообщение очереди
<code>b_cont</code>	Указывает на продолжение сообщения и используется для связывания различных частей одного сообщения
<code>b_datap</code>	Указатель на заголовок блока данных
<code>b_rptr, b_wptr</code>	Указатели, определяющие расположение (начало и конец) данных в буфере данных
	Содержит ссылку на следующую структуру msgb

Заголовок блока данных datab используется для описания буфера и имеет следующие поля:

<code>db_base</code>	Адрес начала буфера
<code>db_lim</code>	Адрес ячейки памяти, следующей непосредственно за буфером. Таким образом, размер буфера равен <code>db_lim - db_base</code>
<code>db_type</code>	Тип сообщения
<code>db_ref</code>	Число заголовков сообщения, адресующих этот блок

Использование этих структур данных для формирования очереди сообщений и сообщений, состоящих из нескольких частей, показано на рис. 5.17.

Поле `b_cont` заголовка сообщения позволяет объединять несколько блоков данных в одно сообщение. Эта возможность особенно полезна при использовании подсистемы STREAMS для реализации сетевых протоколов. Сетевые протоколы имеют уровневую организацию. По мере передачи данных вниз по потоку, каждый последующий модуль (реализующий протокол определенного уровня) добавляет собственную управляющую информацию. Поскольку протоколы верхнего уровня не имеют представления об архитектуре нижних, невозможно заранее зарезервировать необходимую память под сообщение. Вместо того чтобы изменять размер буфера данных сообщения, модуль может добавлять управляющую информацию в виде отдельных частей, связывая их с помощью указателя `b_cont`. Этот процесс, получивший название *инкапсуляции данных*, графически представлен на рис. 5.18.



**Рис. 5.17. Сообщения STREAMS**

Поле `db_ref` заголовка блока данных позволяет нескольким заголовкам сообщения совместно использовать один и тот же буфер. При этом происходит виртуальное копирование сообщения, каждая копия которого может обрабатываться отдельно. Как правило, такой буфер используется совместно только для чтения, хотя сама подсистема STREAMS не накладывает никаких ограничений, возлагая всю ответственность за обработку данных на модули ПОТОКА.

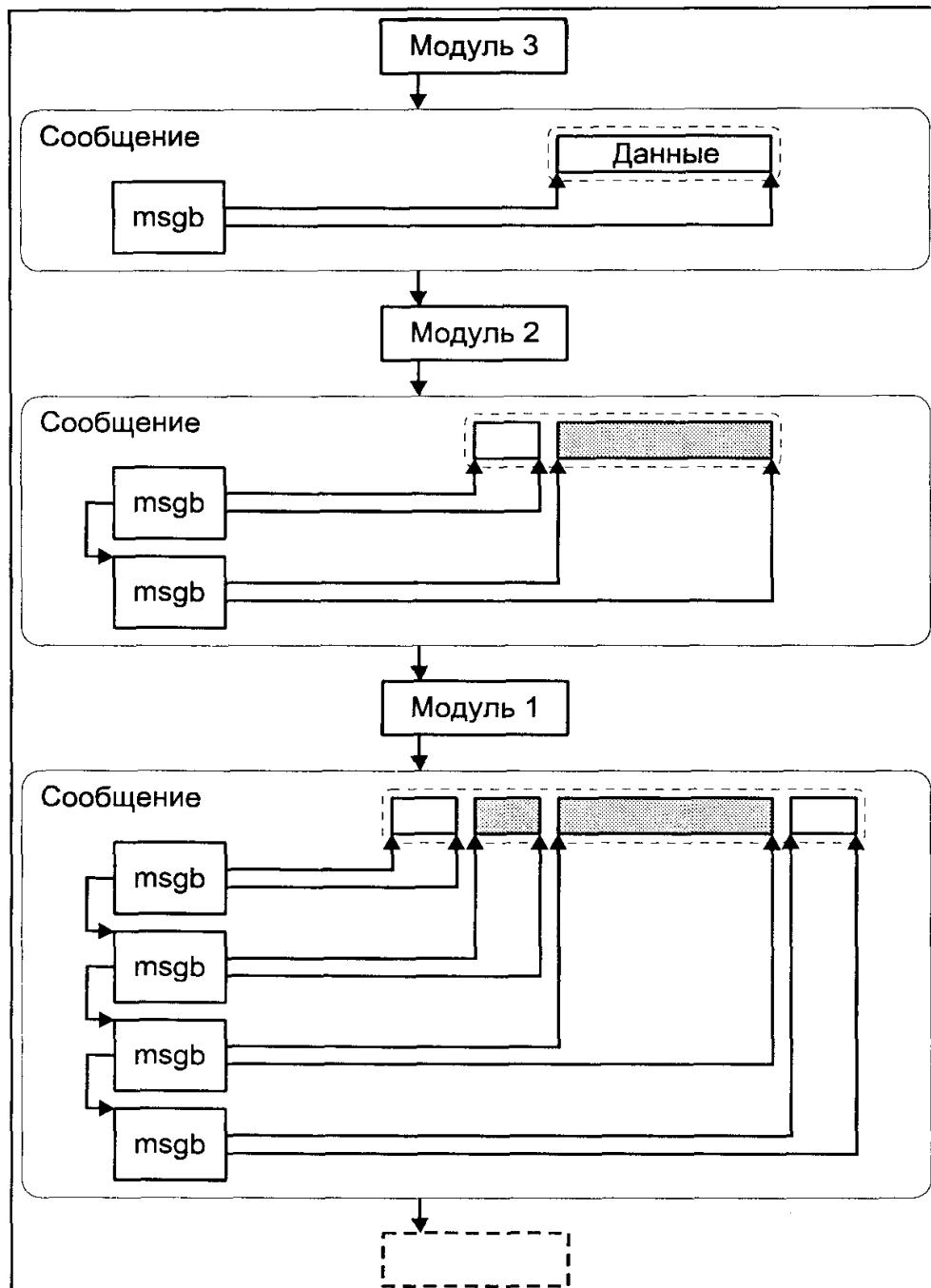


Рис. 5.18. Инкапсуляция данных с использованием составных сообщений

В качестве примера виртуального копирования можно привести реализацию протокола TCP. Протокол TCP является надежным, т. е. данные считаются доставленными только после того, как от получателя поступит подтверждение. Это означает, что протокол должен хранить копии всех отправленных, но не подтвержденных сообщений. Вместо неэффективного физического копирования, производится виртуальное дублирование сообщения, одна копия которого затем передается вниз по потоку (модулю IP), а вторая сохраняется до получения подтверждения. После отправления сообщения драйвером сетевого адаптера, одна из копий будет уничтожена,

что выразится в уменьшении поля `db_ref` заголовка блока данных, но сам блок данных сохранится, поскольку значение счетчика по-прежнему будет превышать 0. И только после получения подтверждения `db_ref` станет равным 0, и соответствующий буфер будет освобожден.

## Типы сообщений

Каждое сообщение принадлежит определенному типу, определяющему назначение сообщения и его приоритет. В зависимости от типа сообщения попадают в одну из двух категорий: *обычные сообщения* и *приоритетные сообщения*. Категория определяет порядок, в котором сообщения будут обрабатываться соответствующей процедурой `xxservice()`. Приоритетные сообщения всегда помещаются перед обычными сообщениями и потому обрабатываются в первую очередь.

В подсистеме STREAMS определены следующие типы обычных сообщений:

<code>M_DATA</code>	Содержит обычные данные. Например, системные вызовы <code>read(2)</code> и <code>write(2)</code> осуществляют передачу данных в виде сообщений этого типа.
<code>M_PROTO</code>	Содержит управляющую информацию. Обычно сообщение этого типа содержит также несколько блоков типа <code>M_DATA</code> . С помощью системных вызовов <code>putmsg(2)</code> и <code>getmsg(2)</code> процесс имеет возможность отправлять и получать как управляющую часть сообщения (блок <code>M_PROTO</code> ), так и данные (блоки <code>M_DATA</code> ).
<code>M_BREAK</code>	Посыпается драйверу устройства для генерации команды <code>break</code> .
<code>M_PASSFP</code>	Используется в каналах STREAMS (STREAMS pipe) для передачи файлового указателя от одного конца канала к другому.
<code>M_SIG</code>	Генерируется модулями или драйверами и передается вверх по потоку головному модулю для отправления процессу сигнала.
<code>M_DELAY</code>	Передается драйверу устройства и указывает задержку между последовательно передаваемыми символами. Как правило, используется при работе с медленными устройствами во избежание переполнения их буферов.
<code>M_CTL</code>	Используется для взаимодействия модулей потока друг с другом. Все сообщения этого типа уничтожаются головным модулем и, таким образом, не могут распространяться за пределы потока.
<code>M_IOCTL</code>	Формируется головным модулем в ответ на управляющие команды, переданные процессом с помощью системного вызова <code>ioctl(2)</code> : <code>I_LINK</code> , <code>I_UNLINK</code> , <code>I_PLINK</code> , <code>I_PUNLINK</code> и <code>I_STR</code> . Эти команды используются для создания мультиплексированных потоков. Последняя команда используется для управления модулями потока.
<code>M_SETOPTS</code>	Используется для задания различных характеристик головного модуля.
<code>M_RSE</code>	Зарезервировано для внутреннего использования. Модули и драйверы должны передавать его без изменений.

Как мы увидим далее, на передачу обычных сообщений влияет механизм управления потоком данных, который может быть реализован модулями потока. Этот механизм не оказывает влияния на передачу приоритетных сообщений. Сообщения этой категории будут переданы следующему модулю, независимо от того, насколько заполнена его очередь. Эти сообщения обеспечивают основное взаимодействие между компонентами потока. Перечисленные ниже сообщения являются высокоприоритетными:

M_COPYIN	Передается вверх по потоку головному модулю и указывает ему скопировать данные от процесса для команды <i>ioctl(2)</i> . Сообщение допустимо в интервале между получением сообщения M_IOCTL и сообщения M_IOCACK или M_IOCNAK.
M_COPYOUT	Передается вверх по потоку головному модулю и указывает ему передать данные, связанные с вызовом <i>ioctl(2)</i> , процессу. Сообщение допустимо в интервале между получением сообщения M_IOCTL и сообщений M_IOCACK или M_IOCNAK.
M_ERROR	Передается вверх по потоку головному модулю и указывает на возникновение ошибки вниз по потоку. Последующие операции с потоком будут заканчиваться ошибкой, за исключением системных вызовов <i>close(2)</i> и <i>poll(2)</i> .
M_FLUSH	При получении этого сообщения модуль должен очистить очередь (чтения, записи или обе) от сообщений.
M_HANGUP	Передается вверх по потоку головному модулю и указывает, что драйвер не может передавать данные, обычно из-за обрыва линии (связи с удаленным объектом).
M_IOCACK	Подтверждение предыдущего сообщения M_IOCTL. В ответ головной модуль возвратит необходимые данные процессу, сделавшему системный вызов <i>ioctl(2)</i> .
M_IOCNAK	Если выполнение команды <i>ioctl(2)</i> закончилось неудачей, это сообщение передается вверх по потоку головному модулю, в ответ на это последний возвратит процессу ошибку.
M_PCPROTO	Высокоприоритетная версия сообщения M_PROTO.
M_PCSIG	Высокоприоритетная версия сообщения M_SIG.
M_PCRSE	Зарезервировано для внутреннего использования в подсистеме.
M_READ	Сообщение передается вниз по потоку, когда от процесса поступает запрос на чтение, но в головном модуле отсутствуют данные.
M_STOP	Предписывает немедленно прекратить передачу.
M_START	Предписывает продолжить передачу после останова, вызванного сообщением M_STOP.

## Передача данных

Как уже обсуждалось, передача данных в потоке происходит в виде сообщений. Процесс инициирует передачу данных с помощью системных вы-

зовов *write(2)* и *putmsg(2)*, которые непосредственно взаимодействуют с головным модулем. Головной модуль формирует сообщение, копируя в него прикладные данные, и передает его следующему модулю вниз по потоку. В конечном итоге сообщение принимается драйвером, который выполняет необходимые операции с конкретным устройством. В случае, когда драйвер получает данные от устройства, он также передает их в виде сообщений вверх по потоку. Процесс имеет возможность получить данные с помощью системных вызовов *read(2)* или *getmsg(2)*. Если в головном модуле данные отсутствуют, процесс блокируется и переходит в состояние сна.

Сообщения передаются модулями с помощью системной функции *putnext(9F)*:

```
#include <sys/stream.h>
#include <sys/ddi.h>

int putnext(queue_t *q, mblk_t *mp);
```

Эта функция адресует очередь следующего модуля параметром *q* и вызывает процедуру *xxrhit()* этой очереди, передавая ей сообщение *tr*. Не поощряется непосредственный вызов функции *xxrhit()* следующего модуля, поскольку это может вызвать определенные проблемы переносимости.

Передача данных внутри потока осуществляется асинхронно и не может блокировать процесс. Блокирование процесса возможно только при передаче данных между процессом и головным модулем. Таким образом, функции обработки данных потока — *xxrhit()* и *xxservice()* не могут блокироваться. Если процедура *xxrhit()* не может передать данные следующему модулю, она помещает сообщение в собственную очередь, откуда оно может быть передано позже процедурой *xxservice()*. Если и процедура *xxservice()* не может осуществить передачу сообщения, например, из-за переполнения очереди следующего модуля, она не будет ожидать изменения ситуации, а вернет сообщение обратно в собственную очередь и завершит выполнение. Попытка передачи повторится, когда ядро через некоторое время опять запустит *xxservice()*.

Процедура *xxservice()* вызывается в системном контексте, а не в контексте процесса, который инициировал передачу данных. Таким образом, блокирование процедуры *xxservice()* может заблокировать (перевести в состояние сна) независимый процесс, что может привести к непредсказуемым результатам и потому недопустимо. Решение этой проблемы заключается в запрещении процедурам *xxrhit()* и *xxservice()* блокирования своего выполнения.

Блокирование недопустимо и для драйвера. Обычно прием данных драйвером осуществляется с использованием прерываний. Таким образом процедура *xxrhit()* вызывается в контексте прерывания и не может блокировать свое выполнение.

Когда процедура `xxput()` не может передать сообщение следующему модулю, она вызывает функцию `putq(9F)`, имеющую следующий вид:

```
#include <sys/stream.h>
int putq(queue_t *q, mblk_t *mp);
```

Функция `putq(9F)` помещает сообщение `mp` в очередь `q`, где сообщение ожидает последующей передачи, и заносит очередь в список очередей, нуждающихся в обработке. Для таких очередей ядро автоматически вызывает процедуру `xxservice()`. Планирование вызова процедур `xxservice()` производится функцией ядра `runqueues()`<sup>8</sup>. Функция `runqueues()` вызывается ядром в двух случаях:

- Когда какой-либо процесс выполняет операцию ввода/вывода над потоком.
- Непосредственно перед переходом какого-либо процесса из режима ядра в режим задачи.

Заметим, что планирование обслуживания очередей не связано с конкретным процессом и производится для всей подсистемы STREAMS в целом.

Функция `runqueues()` производит поиск всех потоков, нуждающихся в обработке очередей. При наличии таких просматривается список очередей, ожидающих обработки, и для каждой из них вызывается соответствующая функция `xxservice()`. Каждая процедура `xxservice()`, в свою очередь, пытается передать все сообщения очереди следующему модулю. Если для каких-либо сообщений это не удается, они остаются в очереди, ожидая следующего вызова `runqueues()`, после чего процесс повторяется.

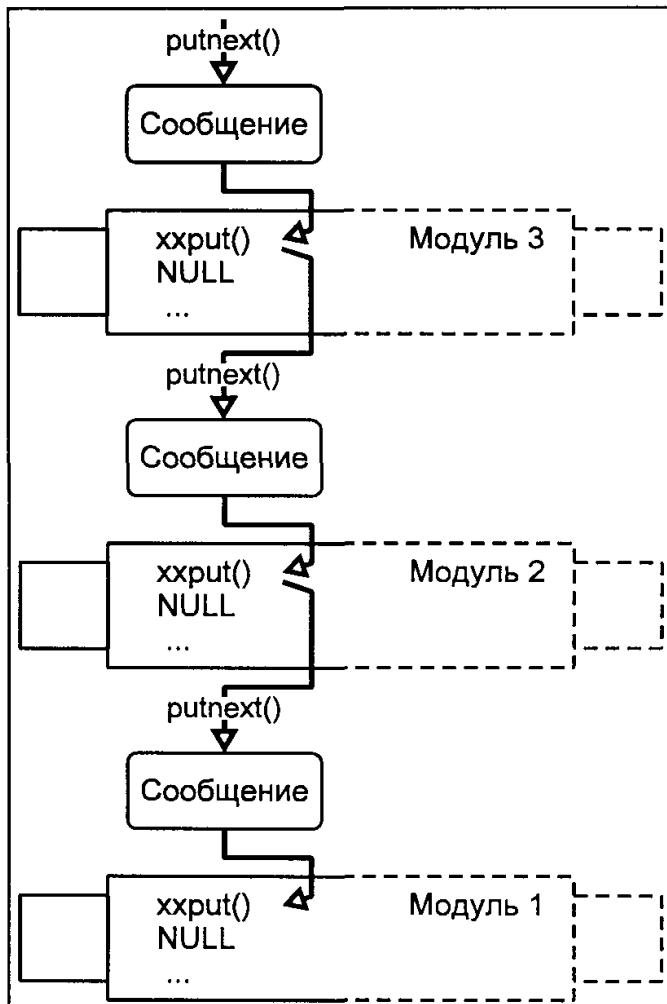
## Управление передачей данных

Деление процесса передачи данных на два этапа, выполняемых, соответственно, функциями `xxput()` и `xxservice()`, позволяет реализовать механизм управления передачей данных.

Как уже упоминалось, обязательной для модуля является лишь функция `xxput()`. Рассмотрим ситуацию, когда модули потока не содержат процедур `xxservice()`. В этом случае, проиллюстрированном на рис. 5.19, каждый предыдущий модуль вызывает функцию `xxput()` следующего, передавая ему сообщение, с помощью функции ядра `putnext(9F)`. Функция `xxput()` немедленно вызывает `putnext(9F)` и т. д.:

```
xxput(queue_t *q, mblk_t *mp)
{
    putnext(q, mp);
```

<sup>8</sup> Система планирования STREAMS использует собственные функции и ИС имет отношение к планированию процессов в UNIX.



**Рис. 5.19.** Передача данных без управления потоком

Когда данные достигают драйвера, он передает их непосредственно устройству. Если устройство занято, или драйвер не может немедленно обработать данные, сообщение уничтожается. В данном примере никакого управления потоком не происходит, и очереди сообщений не используются.

Хотя такой вариант может применяться для некоторых драйверов (как правило, для псевдоустройств, например, `/dev/null`), в общем случае устройство не может быть все время готово к обработке данных, а потеря данных из-за занятости устройства недопустима. Таким образом, в потоке может происходить блокирование передачи данных<sup>9</sup>, и эта ситуация не должна приводить к потере сообщений, во избежание которой необходим согласованный между модулями механизм управления потоком. Для этого сообщения обрабатываются и буферизуются в соответствующей очереди модуля, а их передача возлагается на функцию `xxservice()`, вызываемую ядром автоматически. Для каждой очереди определены две ватерлинии —

<sup>9</sup> Блокирование передачи может происходить не только в драйвере (оконечном модуле) ПОТОКА из-за занятости устройства. Возможна ситуация, когда отдельный модуль вынужден отложить обработку сообщений до наступления некоторого события.

верхняя и нижняя, которые используются для контроля заполненности очереди. Если число сообщений превышает верхнюю ватерлинию, очередь считается переполненной, и передача сообщений блокируется, пока их число не станет меньше нижней ватерлинии.

Рассмотрим пример потока, модули 1 и 3 которого поддерживают управление потоком данных, а модуль 2 — нет. Другими словами, модуль 2 не имеет процедуры `xxservice()`. Когда сообщение достигает модуля 3, вызывается его функция `xxput()`. После необходимой обработки сообщения, оно помещается в очередь модуля 3 с помощью функции `putq(9F)`. Если при этом число сообщений в очереди превышает верхнюю ватерлинию, `putq(9F)` устанавливает специальный флаг, сигнализирующий о том, что очередь переполнена:

```
mod1put(queue_t *q, mblk_t *mp)
{
    /* Необходимая обработка сообщения*/

    putq(q, mp);
}
```

Через некоторое время ядро автоматически запускает процедуру `xxservice()` модуля 3. Для каждого сообщения очереди `xxput()` вызывает функцию `canput(9F)`, которая проверяет заполненность очереди следующего по потоку модуля. Функция `canput(9F)` имеет вид:

```
#include <sys/stream.h>
int canput(queue_t *q);
```

Заметим, что `canput(9F)` проверяет заполненность очереди следующего модуля, реализующего механизм управления передачей данных, т. е. производящего обработку очереди с помощью процедуры `xxservice()`. В противном случае, как уже говорилось, очередь модуля не принимает участия в передаче данных. В нашем примере, `canput(9F)` проверит заполненность очереди записи модуля 1. Функция возвращает истинное значение, если очередь может принять сообщение, и ложное — в противном случае. В зависимости от результата проверки процедура `xxservice()` либо передаст сообщение следующему модулю (в нашем примере — модулю 2, который после необходимой обработки сразу же передаст его модулю 1), либо вернет сообщение обратно в очередь, если следующая очередь переполнена.

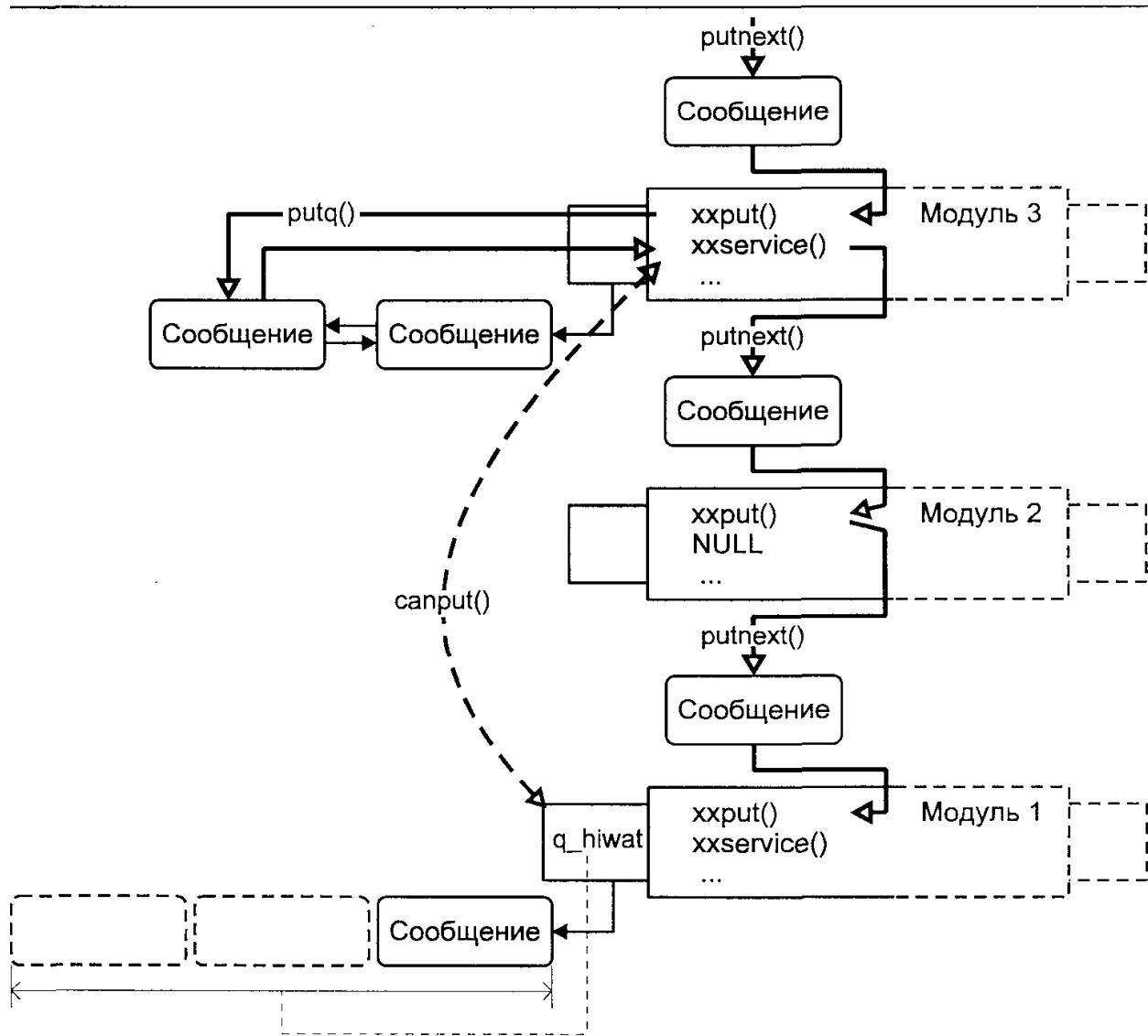
Описанная схема показана на рис. 5.20. Ниже приведен скелет процедуры `xxservice()` модуля 3, иллюстрирующий описанный алгоритм передачи сообщений с использованием механизма управления передачей данных.

```
mod3service(queue_t *q)
{
    mblk_t *mp;
    while((mp = getq(q)) !=NULL) {
        if (canput(q->q_next))
```

```

    putnext(q, mp)
else
    putbq(q, mp);
    break;
}

```



**Рис. 5.20.** Управление потоком данных

В этом примере функция *getq(9F)* используется для извлечения следующего сообщения из очереди, а функция *putbq(9F)* — для помещения сообщения в начало очереди. Если модуль 1 блокирует передачу, т. е. *canput(9F)* вернет "ложно", процедура *xxservice()* завершает свою работу, и сообщения начинают буферизироваться в очереди модуля 3. При этом очередь временно исключается из списка очередей, ожидающих обработки, и процедура *xxservice()* для нее вызываться не будет. Данная ситуация про-

длится до тех пор, пока число сообщений очереди записи модуля 1 не станет меньше нижней ватерлинии.

Пока существует возникшая блокировка передачи, затор будет постепенно распространяться вверх по потоку, последовательно заполняя очереди модулей, пока, в конечном итоге, не достигнет головного модуля. Поскольку передачу данных в головной модуль (вниз по потоку) инициирует приложение, попытка передать данные в переполненный головной модуль вызовет блокирование процесса<sup>10</sup> и переход его в состояние сна.

В конечном итоге, модуль 1 обработает сообщения своей очереди, и их число станет меньше нижней ватерлинии. Как только очередь модуля 1 станет готовой к приему новых сообщений, планировщик STREAMS автоматически вызовет процедуры `xxservice()` для модулей, ожидавших освобождения очереди модуля 1, в нашем примере — для модуля 3.

Управление передачей данных в потоке требует согласованной работы всех модулей. Например, если процедура `xxrput()` буферизирует сообщения для последующей обработки `xxservice()`, такой алгоритм должен выполняться для всех сообщений". В противном случае, это может привести к нарушению порядка сообщений, и как следствие, к потере данных.

Когда запускается процедура `xxservice()`, она должна обработать все сообщения очереди. "Уважительной" причиной прекращения обработки является переполнение очереди следующего по потоку модуля. В противном случае нарушается механизм управления передачей, и очередь может навсегда лишиться обработки.

## Драйвер

Драйверы и модули очень похожи, они используют одинаковые структуры данных (`streamtab`, `qinit`, `module_info`) и одинаковый интерфейс (`xxopen()`, `xxrput()`, `xxservice()` и `xxclose()`). Однако между драйверами и модулями существуют различия.

Во-первых, только драйверы могут непосредственно взаимодействовать с аппаратурой и отвечать за обработку аппаратных прерываний. Поэтому драйвер должен зарегистрировать в ядре соответствующий обработчик прерываний. Аппаратура обычно генерирует прерывания при получении данных. В ответ на это драйвер копирует данные от устройства, формирует сообщение и передает его вверх по потоку.

Во-вторых, к драйверу может быть подключено несколько потоков. Как уже обсуждалось, на мультиплексировании потоков построены многие

<sup>10</sup> Это единственная ситуация, в которой возможно блокирование процесса.

<sup>11</sup> Более точно — для всех сообщений с данным приоритетом.

подсистемы ядра, например, поддержка сетевых протоколов. В качестве мультиплексора может выступать только драйвер. Несмотря на то что драйвер в этом случае не является оконечным модулем (см., например, рис. 5.15), размещение драйверов существенным образом отличается от встраивания модулей.

Наконец, процесс инициализации драйверов и модулей различен. Функция `ххореп()` драйвера вызывается при открытии потока, в то время как инициализация модуля происходит при встраивании.

## Головной модуль

Обработку системных вызовов процессов осуществляет головной модуль. Головной модуль потока является единственным местом, где возможно блокирование обработки и, соответственно, процесса, в контексте которого осуществляется операция ввода/вывода. Головной модуль является внешним интерфейсом потока, и хотя его структура похожа на структуру обычного модуля, функции обработки здесь обеспечиваются самой подсистемой STREAMS. В отличие от точек входа в модуль или драйвер потока, реализующих специфическую для данного устройства обработку, функции головного модуля выполняют ряд общих для всех потоков задач, включающих:

- Трансляцию данных, передаваемых процессом с помощью системных вызовов, в сообщения и передачу их вниз по потоку.
- Сообщение об ошибках и отправление сигналов процессам, связанным с потоком.
- Распаковку сообщений, переданных вверх по потоку, и копирование данных в пространство ядра или задачи.

Процесс передает данные потоку с помощью системных вызовов `write(2)` и `putmsg(2)`. Системный вызов `write(2)`, представляющий собой унифицированный интерфейс передачи данных любым устройствам, позволяет производить передачу простых данных в виде потока байтов, не сохраняя границы логических записей. Системный вызов `putmsg(2)`, предназначенный специально для работы с потоками, позволяет процессу за один вызов передать управляющее сообщение и данные. Головной модуль преобразует эту информацию в единое сообщение с сохранением границ записи.

Системный вызов `putmsg(2)` имеет вид:

```
#include <stropts.h>
int putmsg(int fildes, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flags);
```

С помощью этого вызова головной модуль формирует сообщение, состоящее из управляющей части м PROTO и данных, передаваемых в блоках

м `M_DATA`. Содержимое сообщения передается с помощью указателей на структуру `strbuf` — `ctlptr` для управляющего блока и `dataptr` для блоков данных.

Структура `strbuf` имеет следующий формат:

```
struct strbuf {
    int maxlen;
    int len;
    void *buf;
}
```

где `maxlen` не используется, `len` — размер передаваемых данных, `buf` — указатель на буфер.

С помощью аргумента `flags` процесс может передавать экстренные сообщения, установив флаг `RS_HIPRI`.

В обоих случаях головной модуль формирует сообщение и с помощью функции `capriit(9F)` проверяет, способен ли следующий вниз по потоку модуль, обеспечивающий механизм управления передачей, принять его. Если `capriit(9F)` возвращает истинный ответ, сообщение передается вниз по потоку с помощью функции `putnexti(9F)`, а управление возвращается процессу. Если `capriit(9F)` возвращает ложный ответ, выполнение процесса блокируется, и он переходит в состояние сна, пока не рассосется образовавшийся затор. Заметим, что возврат системного вызова еще не гарантирует, что данные получены устройством. Возврат из `write(2)` или `putmsg(2)` свидетельствует лишь о том, что данные были успешно скопированы в адресное пространство ядра, и в виде сообщения направлены вниз по потоку.

Процесс может получить данные из потока с помощью системных вызовов `read(2)` и `getmsg(2)`. Стандартный вызов `read(2)` позволяет получать только обычные данные без сохранения границ сообщений<sup>12</sup>. В отличие от этого вызова `getmsg(2)` позволяет получать данные сообщений типов `M_DATA` и `M_PROTO`, при этом сохраняются границы сообщений. Например, если полученное сообщение состоит из блока `M_PROTO` и нескольких блоков `M_DATA`, вызов `getmsg(2)` корректно разделит сообщение на две части: управляющую информацию и собственно данные.

Вызов `getmsg(2)` имеет вид:

```
#include <stropts.h>
int getmsg(int fildes, struct strbuf *ctlptr,
           struct strbuf *dataptr, int *flagsp);
```

<sup>12</sup> С помощью сообщения `M_SETOPTS` можно дать указания головному модулю обрабатывать сообщения `M_PROTO` как обычные данные. В этом случае вызов `read(2)` будет возвращать содержимое как сообщений `M_DATA`, так и `M_PROTO`. Однако информация о типе сообщения (данных) и границы сообщений сохранены не будут.

С помощью вызова `getmsg(2)` прикладной процесс может получить сообщение, причем его управляющие и прикладные данные будут помещены в буферы, адресуемые `ctlptr` и `dataptr` соответственно. Так же как и в случае `putmsg(2)` эти указатели адресуют структуру `strbuf`, которая отличается только тем, что поле `maxlen` определяет максимальный размер буфера, а `len` устанавливается равным фактическому числу полученных байтов. По умолчанию `getmsg(2)` получает первое полученное сообщение, однако с помощью флага `RS_HIPRI`, установленного в переменной, адресуемой аргументом `flagsp`, процесс может потребовать получение только экстренных сообщений.

В обоих случаях, если данные находятся в головном модуле, ядро извлекает их из сообщения, копирует в адресное пространство процесса и возвращает управление последнему. Если же в головном модуле отсутствуют сообщения, ожидающие получения, выполнение процесса блокируется, и он переходит в состояние сна до прихода сообщения.

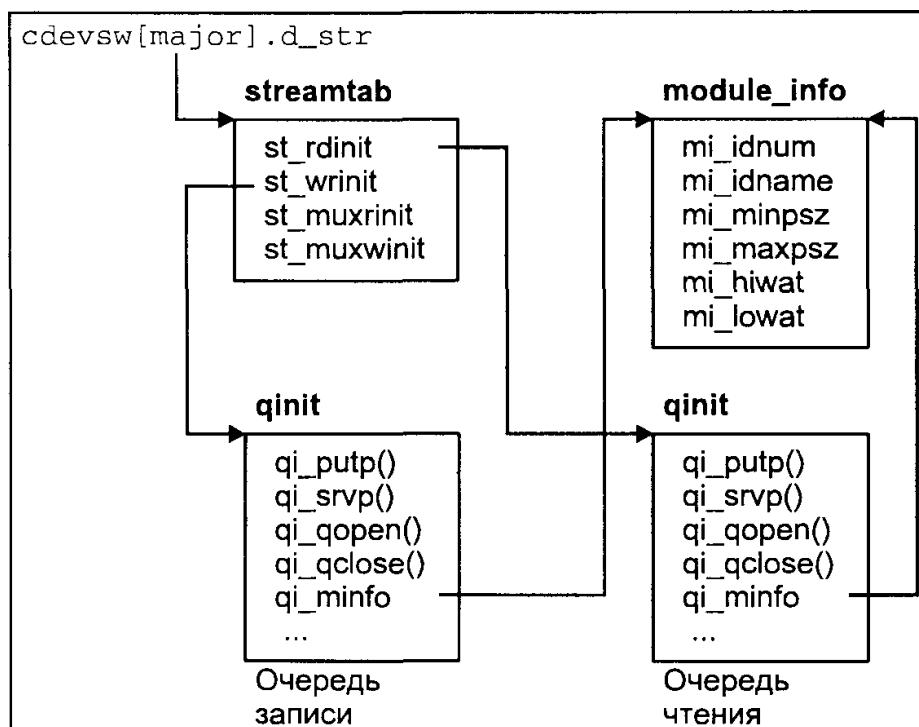
Когда головной модуль получает сообщение, ядро проверяет, ожидает ли его какой-либо процесс. Если такой процесс имеется, ядро пробуждает процесс, копирует данные в пространство задачи и производит возврат из системного вызова. Если ни один из процессов не ожидает получения сообщения, оно буферизуется в очереди чтения головного модуля.

## Доступ к потоку

Как и для обычных драйверов устройств, рассмотренных ранее, прежде чем процесс сможет получить доступ к драйверу STREAMS, необходимо встроить драйвер в ядро системы и создать специальный файл устройства — файловый интерфейс доступа. Независимо от того, как именно осуществляется встраивание (статически с перекомпиляцией ядра, или динамически), для этого используются три структуры данных, определенных для любого драйвера или модуля STREAMS: `module_info`, `qinit` и `streamtab`. Связь между ними представлена на рис. 5.21.

Структура `streamtab` используется ядром для доступа к точкам входа драйвера или модуля — к процедурам его очередей `xxopen()`, `xxclose()`, `xxrput()` и `xxservice()`. Для этого `streamtab` содержит два указателя на структуры `qinit`, соответственно, для обработки сообщений очереди чтения и записи. Два других указателя, также на структуры `qinit`, используются только для мультиплексоров для обработки команды `I_LINK`, используемой при конфигурации мультиплексированного потока. Каждая структура `qinit` определяет процедуры, необходимые для обработки сообщений вверх и вниз по потоку (очередей чтения и записи). Функции `xxopen()` и `xxclose()` являются общими для всего модуля и определены только для очереди чтения. Все очереди модуля имеют ассоциированную с ними процедуру `xxrput()`, в то время как процедура `xxservice()` определяется

только для очередей, реализующих управление передачей. Каждая структура `qinit` также имеет указатель на структуру `module_info`, которая обычно определяется для всего модуля и хранит базовые значения таких параметров, как максимальный и минимальный размеры передаваемых пакетов данных (`mi_maxpsz`, `mi_minpsz`), значения ватерлиний (`mi_hiwat`, `mi_lowat`), а также идентификатор и имя драйвера (модуля) (`mi_idnum`, `mi_idname`).



**Рис. 5.21.** Конфигурационные данные драйвера (модуля) STREAMS

Доступ к драйверам STREAMS осуществляется с помощью коммутатора символьных устройств — таблицы `cdevsw[]`. Каждая запись этой таблицы имеет поле `d_str`, которое равно NULL для обычных символьных устройств. Для драйверов STREAMS это поле хранит указатель на структуру `streamtab` драйвера. Таким образом, через коммутатор устройств ядро имеет доступ к структуре `streamtab` драйвера, а значит и к его точкам входа. Для обеспечения доступа к драйверу из прикладного процесса необходимо создать файловый интерфейс — т. е. специальный файл символьного устройства, старший номер которого был бы равен номеру элемента `cdevsw[]`, адресующего точки входа драйвера.

## Создание потока

Поток создается при первом открытии с помощью системного вызова `open(2)` специального файла устройства, ассоциированного с драйвером STREAMS. Как правило, процесс создает поток в два этапа: сначала создается элементарный поток, состоящий из нужного драйвера и головного модуля

(являющегося обязательным приложением), а затем производится встраивание дополнительных модулей для получения требуемой функциональности.

Процесс открывает поток с помощью системного вызова `open(2)`, передавая ему в качестве аргумента имя специального файла устройства. При этом ядро производит трансляцию имени и обнаруживает, что адресуемый файл принадлежит файловой системе `specfs`, через которую в дальнейшем производятся все операции работы с файлом. В памяти размещается `vnode` этого файла и вызывается функция открытия файла для файловой системы `specfs` — `spec_open()`. В свою очередь `spec_open()` находит требуемый элемент коммутатора `cdevsw[]` и обнаруживает, что поле `d_str` ненулевое. Тогда она вызывает процедуру подсистемы STREAMS `stropen()`, которая отвечает за размещение головного модуля и подключение драйвера. После выполнения необходимых операций поток приобретает вид, изображенный на рис. 5.22.

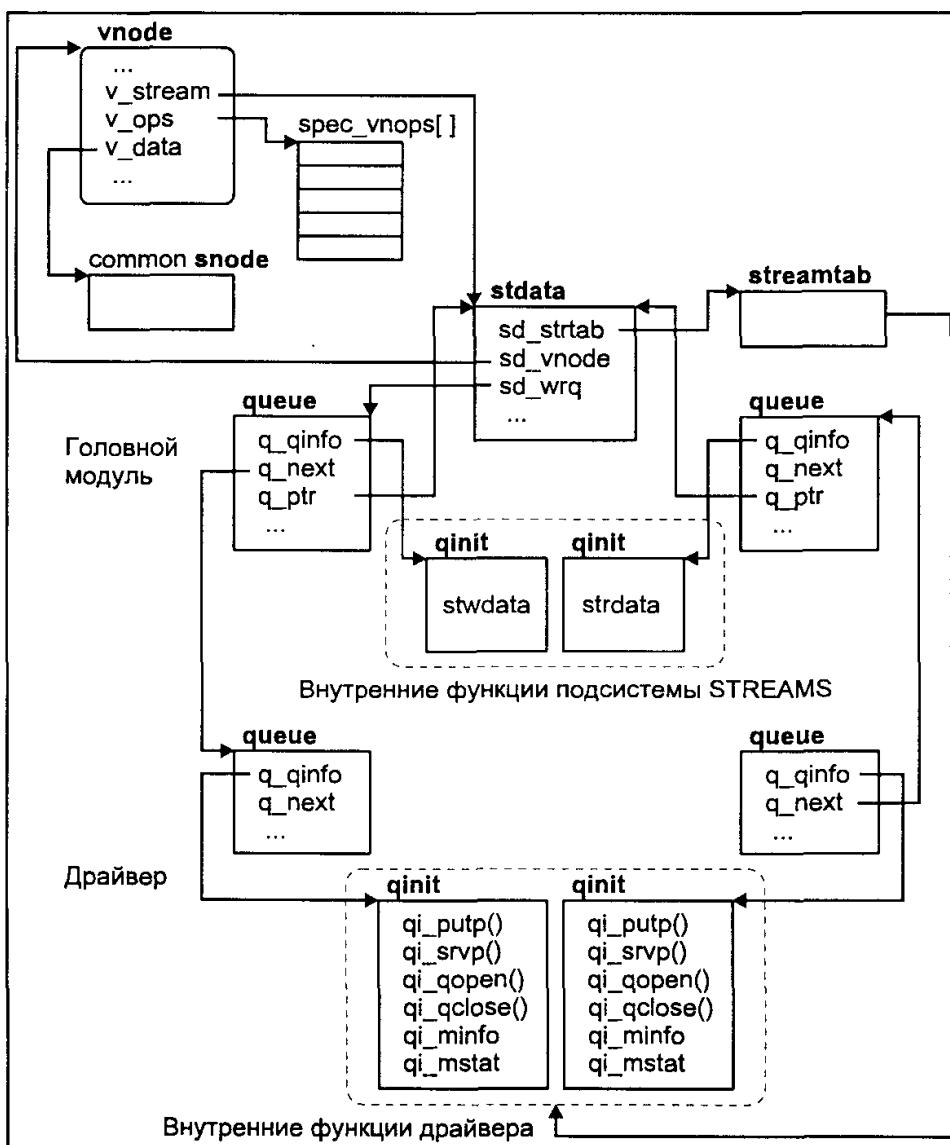


Рис. 5.22. Структура потока после открытия

Головной модуль представлен структурой `stdata`, которая выполняет роль интерфейса между потоком и ядром системы при выполнении операций чтения, записи и управления. Индексный дескриптор `vnode` содержит указатель на эту структуру. Поля `q_ptr` структур `queue` головного модуля также указывают на `stdata`. Поля `q_qinfo` очередей `queue` указывают на структуры `qinit`, адресующие общие для всех головных модулей функции, реализованные самой подсистемой STREAMS.

Очереди чтения и записи драйвера связываются с соответствующими очередями головного модуля. Информация, хранящаяся в структуре `streamtab` используется для заполнения полей `q_qinfo` соответствующих структур `queue` драйвера указателями на процедурные интерфейсы очередей чтения и записи.

В завершение вызывается функция `xxopen()` драйвера. При последующих операциях открытия потока функция `stropen()` последовательно вызовет функции `xxopen()` каждого модуля и драйвера, тем самым информируя их, что другой процесс открыл тот же поток, и позволяя разместить соответствующие структуры данных для обработки нескольких каналов одновременно. Обычно открытие потоков производится через драйвер клонов.

После открытия потока процесс может произвести встраивание необходимых модулей. Для этого используется системный вызов `ioctl(2)`. Команда `I_PUSH` этой функции служит для встраивания модулей, а команда `I_POP` — для извлечения модулей из потока. Приведем типичный сценарий конструирования потока:

```
fd = open("/dev/stream", O_RDWR);
ioctl(fd, I_PUSH, "module1");
ioctl(fd, I_PUSH, "module2");

ioctl(fd, I_POP, (char *)0);
ioctl(fd, I_POP, (char *)0);
close(fd);
```

В этом примере процесс открыл поток `/dev/stream`, а затем последовательно встроил модули `module1` и `module2`. Заметим, что команда `I_PUSH` системного вызова `ioctl(2)` встраивает модуль непосредственно после головного модуля. После выполнения операций ввода/вывода, процесс извлек модули и закрыл поток<sup>13</sup>.

Поскольку модули описываются такими же структурами данных, что и драйверы, схемы их встраивания похожи. Как и в случае драйверов, для заполнения полей `q_qinfo` структур `queue` используются данные из структуры `streamtab` модуля. Для хранения информации, необходимой для инициализации модуля, во многих версиях UNIX используется табли-

При закрытии потока все встроенные модули извлекаются автоматически.

ца `fmodsw[]`, каждый элемент которой хранит имя модуля и указатель на структуру `streamtab`. После установления всех связей вызывается функция `xxordep()` модуля.

## Управление потоком

Управление потоком осуществляется прикладным процессом с помощью команд системного вызова `ioctl(2)`:

```
ttinclude<sys/types.h>
#include <stropts.h>
#include <sys/conf.h>

int ioctl(int fildes, int command, ... /* arg */);
```

Хотя часть команд обрабатывается исключительно головным модулем потока, другие предназначены промежуточным модулям или драйверу. Для этого головной модуль преобразует команды `ioctl(2)` в сообщения и направляет их вниз по потоку. При этом возникают две потенциальные проблемы: синхронизация процесса с системным вызовом (поскольку передача сообщения и реакция модуля имеют асинхронный характер) и передача данных между процессом и модулем.

Синхронизацию осуществляет головной модуль. Когда процесс выполняет системный вызов `ioctl(2)`, который может быть обработан самим головным модулем, последний выполняет все операции в контексте процесса, и никаких проблем синхронизации и копирования данных не возникает. Именно так происходит обработка `ioctl(2)` для обычных драйверов устройств. Если же головной модуль не может обработать команду, он блокирует выполнение процесса и формирует сообщение `M_IOCCTL`, содержащее команду и ее параметры, и отправляет его вниз по потоку. Если какой-либо модуль вниз по потоку может выполнить указанную команду, в ответ он направляет подтверждение в виде сообщения `M_IOCACK`. Если ни один из модулей и сам драйвер не смогли обработать команду, драйвер направляет вверх по потоку сообщение `M_IOCNAK`. При получении одного из этих сообщений головной модуль пробуждает процесс и передает ему результаты выполнения команды.

При обработке сообщения промежуточным модулем или драйвером возникает проблема передачи данных. Как правило, команда `ioctl(2)` содержит ассоциированные с ней параметры, число и размер которых зависят от команды. При обработке команды `ioctl(2)` обычным драйвером последний имеет возможность копировать параметры из пространства задачи и подобным образом возвращать результаты, поскольку вся обработка команды происходит в контексте процесса.

Эта схема неприменима для подсистемы STREAMS. Обработка сообщений модулем или драйвером выполняется в системном контексте и не имеет

отношения к адресному пространству текущего процесса. Поэтому модуль не имеет возможности копировать параметры команды и возвращать результаты обработки, используя адресное пространство задачи.

Для преодоления этой проблемы в подсистеме STREAMS предлагаются два подхода.

Первый из них основан на использовании специальной команды *ioctl(2)* *I\_STR*. При этом в качестве параметра передается указатель на структуру *strioctl*:

```
ioctl(fd, I_STR, (struct strioctl *)arg);
struct strioctl {
    int ic_cmd;
    int ic_timeout;
    int ic_len;
    char*ic_dp;
}
```

где *ic\_cmd* — фактическая команда,  
*ic\_timeout* — число секунд, которое головной модуль будет ожидать подтверждения запроса, после чего он вернет процессу ошибку тайм-аута ETIME,  
*ic\_len* — размер блока параметров команды,  
*ic\_dp* — указатель на блок параметров.

Если головной модуль не может обработать команду, он формирует сообщение *M\_IOCTL* и копирует в него команду (*ic\_cmd*) и блок параметров (*ic\_len*, *ic\_dp*). После этого сообщение направляется вниз по потоку. Когда модуль получает сообщение, оно содержит все необходимые данные для обработки команды. Если команда предполагает передачу информации процессу, модуль записывает необходимые данные в то же сообщение, изменяет его тип на *M\_IOCACK* и отправляет его вверх по потоку. В свою очередь головной модуль получает сообщение и производит передачу параметров процессу.

Другой подход получил название *прозрачных команд ioctl(2)* (transparent *ioctl*). Он позволяет использовать стандартные команды *ioctl(2)*, решая при этом проблему копирования данных. Когда процесс выполняет вызов *ioctl(2)*, головной модуль формирует сообщение *M\_IOCTL* и копирует в него параметры вызова — *command* и *arg*. Обычно параметр *arg* является указателем на блок параметров, размер и содержимое которого известны только модулю (или драйверу), отвечающему за обработку данной команды. Поэтому головной модуль просто копирует этот указатель, не интерпретируя его и тем более не копируя в сообщение сам блок параметров. Сообщение передается вниз по потоку.

Когда модуль получает сообщение, в ответ он отправляет сообщение `M_COPYIN`, содержащее размер и расположение данных<sup>14</sup>, необходимых для выполнения команды. Головной модуль пробуждает процесс, вызвавший `ioctl(2)`, для копирования параметров. Поскольку последующие операции выполняются в контексте процесса, никаких проблем доступа к его адресному пространству не возникает. Головной модуль создает сообщение `M_IOCARGS`, копирует в него параметры команды и направляет сообщение вниз по потоку. После этого процесс опять переходит в состояние сна.

Когда модуль получает сообщение `M_IOCARGS`, он интерпретирует содержащиеся в нем параметры и выполняет команду. В некоторых случаях для получения всех параметров, необходимых для выполнения команды, может потребоваться дополнительный обмен сообщениями `M_COPYIN` и `M_IOCARGS`. Такая ситуация может возникнуть, например, если один из параметров являлся указателем на структуру данных. Для получения копии структуры модулю потребуется дополнительная итерация.

После получения всех необходимых данных и выполнения команды в случае, если результат должен быть передан процессу, модуль формирует одно или несколько сообщений `M_COPYOUT`, помещая в них требуемые данные, и направляет их вверх по потоку. Головной модуль пробуждает процесс, передавая ему результаты выполнения команды. Когда все результаты переданы процессу, модуль посыпает подтверждение `M_IOCACK`, в результате которого головной модуль пробуждает процесс в последний раз, завершая тем самым выполнение вызова `ioctl(2)`.

## Мультиплексирование

Подсистема STREAMS обеспечивает возможность мультиплексирования потоков с помощью *мультиплексора*, который может быть реализован только драйвером STREAMS. Различают три типа мультиплексоров — верхний, нижний и гибридный. *Верхний мультиплексор*, называемый также мультиплексором  $N:1$ , обеспечивает подключение нескольких каналов вверх по потоку к одному каналу вниз по потоку. *Нижний мультиплексор*, называемый также мультиплексором  $1:M$ , обеспечивает подключение нескольких каналов вниз по потоку к одному каналу вверх по потоку. *Гибридный мультиплексор*, как следует из названия, позволяет мультиплексировать несколько каналов вверх по потоку с несколькими каналами вниз по потоку.

<sup>14</sup> Расположение данных уже содержится в параметре `arg`, который передается обратно в сообщении `M_COPYIN`.

Заметим, что подсистема STREAMS обеспечивает возможность мультиплексирования, но за идентификацию различных каналов и маршрутизацию данных между ними отвечает сам мультиплексор.

Мультиплексирование каналов вверх по потоку осуществляется в результате открытия одного и того же драйвера с различными младшими номерами. Верхний мультиплексор должен обеспечить возможность одновременной работы с устройством с использованием различных младших номеров. Если два процесса открывают поток, используя различные младшие номера, ядро создаст отдельный канал для каждого из них, каждый из них будет адресоваться отдельным vnode, и процедура `xxopen()` драйвера будет вызвана дважды. Драйвер при этом будет обрабатывать две пары очередей, каждая из которых отвечает за отдельный поток. Когда данные поступают от устройства, драйвер должен принять решение, в какую очередь чтения их направить. Обычно такое решение делается на основании управляющей информации, содержащейся в полученных данных. На рис. 5.23 представлен вид верхнего мультиплексора с двумя подключенными потоками.

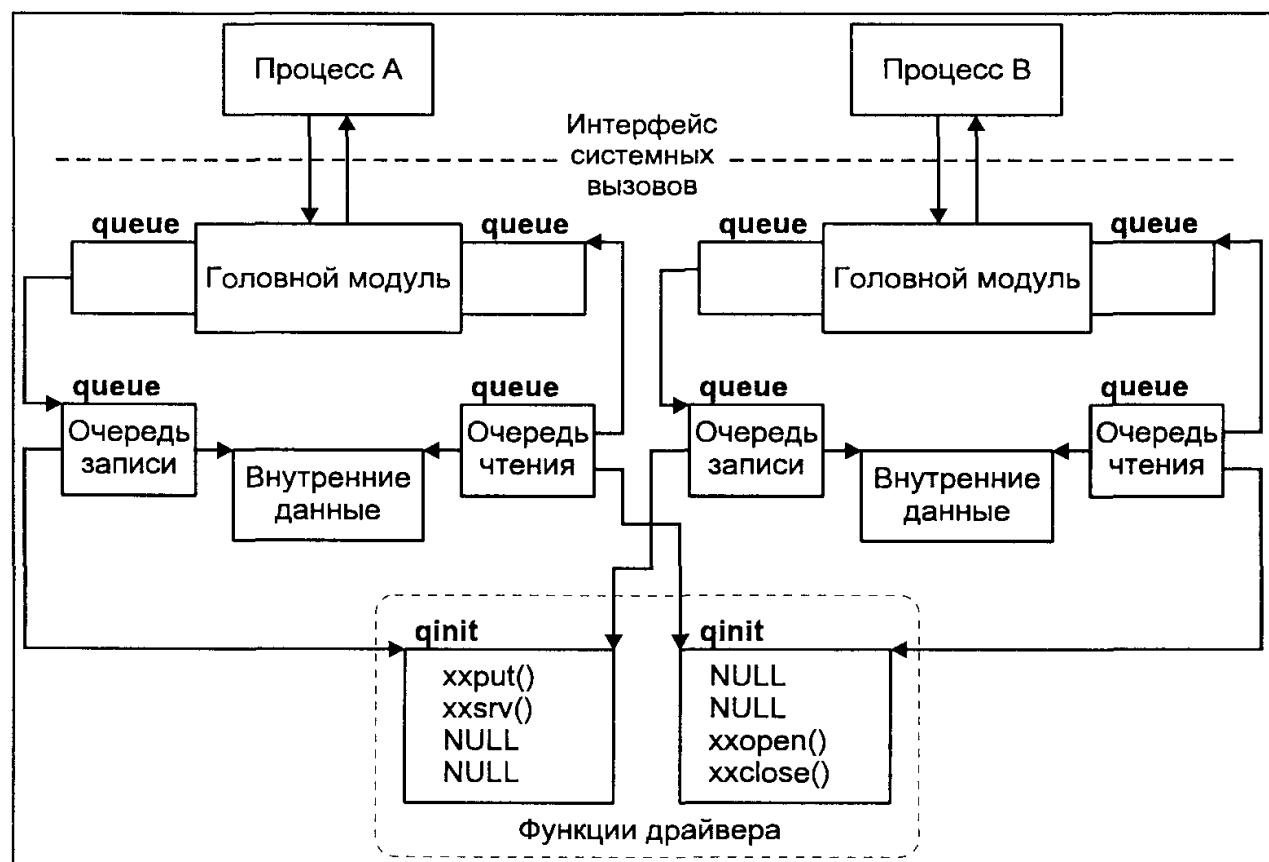


Рис. 5.23. Верхний мультиплексор

Нижний мультиплексор представляет собой драйвер псевдоустройства. Вместо работы с физическим устройством он взаимодействует с несколькими каналами вниз по потоку. Для этого нижний мультиплексор обеспе-

чиает работу с еще одной парой очередей — нижними очередями чтения и записи. Структура `streamtab` нижнего мультиплексора адресует процедурный интерфейс работы с нижними очередями соответственно полями `st_muxrinit` и `st_muxwinit`.

Для работы с мультиплексированными потоками подсистема STREAMS поддерживает четыре команды `ioctl(2)`:

<code>I_LINK</code>	Используется для <b>объединения</b> потоков. При этом файловый дескриптор указывает на поток, подключенный к мультиплексору. Второй файловый дескриптор, передаваемый в качестве аргумента команды, указывает на поток, который необходимо подключить ниже мультиплексора.
<code>I_PLINK</code>	Используется для <b>объединения</b> потоков, которое сохраняется при закрытии файлового дескриптора. В остальном аналогично команде <code>I_LINK</code> .
<code>I_UNLINK</code> , <code>I_PUNLINK</code>	Используются для <b>разъединения</b> потоков, созданных командами <code>I_LINK</code> и <code>I_PLINK</code> .

Создание мультиплексированного потока происходит в два этапа. Поясним этот процесс на примере создания стека протокола IP, поддерживающего работу как с адаптером Ethernet, так и с адаптером FDDI. Для этого необходимо объединить драйвер адаптера Ethernet, драйвер адаптера FDDI и драйвер IP, который является нижним мультиплексором. Процесс должен выполнить следующие действия:

```
fdenet = open("/dev/le", O_RDWR);
fdfddi = open("/dev/fddi", O_RDWR);
fdip = open("/dev/ip", O_RDWR);

ioctl(fdip, I_LINK, fdenet);
ioctl(fdip, I_LINK, fdfddi);
```

Сначала процесс создает три независимых потока, адресуемых дескрипторами `fdenet`, `fdfddi` и `fdip` (рис. 5.24, а) Для объединения потоков используется команда `I_LINK` системного вызова `ioctl(2)`. В результате получается конфигурация, представленная на рис. 5.24, б.

В результате объединения потоков очереди и процедурный интерфейс головного модуля нижнего потока (в данном случае, потока, подключенного к драйверу Ethernet или FDDI), реализованный самой подсистемой STREAMS, заменяются на нижние очереди и соответствующий процедурный интерфейс мультиплексора. Более детально процесс объединения потока IP и потока Ethernet показан на рис. 5.25.

Задачей нижнего мультиплексора является хранение информации обо всех подключенных ниже потоках и обеспечение правильной маршрутизации между ними.

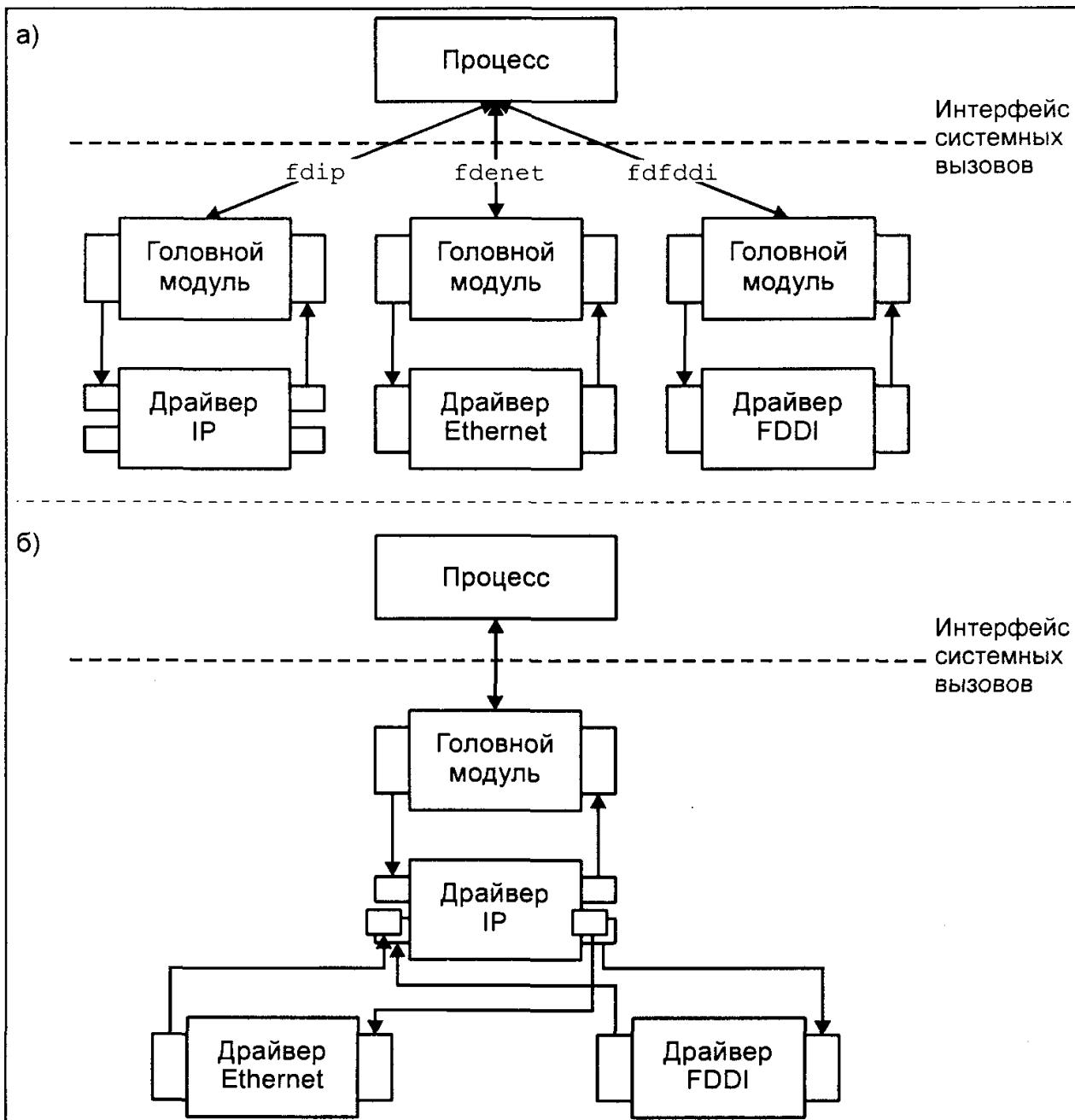


Рис. 5.24. Создание мультиплексированного потока

## Заключение

Эта глава посвящена внутренней архитектуре подсистемы ввода/вывода, движущей силой которой являются драйверы устройств. Были рассмотрены традиционные типы драйверов, присутствующих в операционной системе UNIX с ранних ее версий, — символьные и блочные драйверы. Важную роль в процессе обмена данными с драйвером играют файловый интерфейс и файловая система.

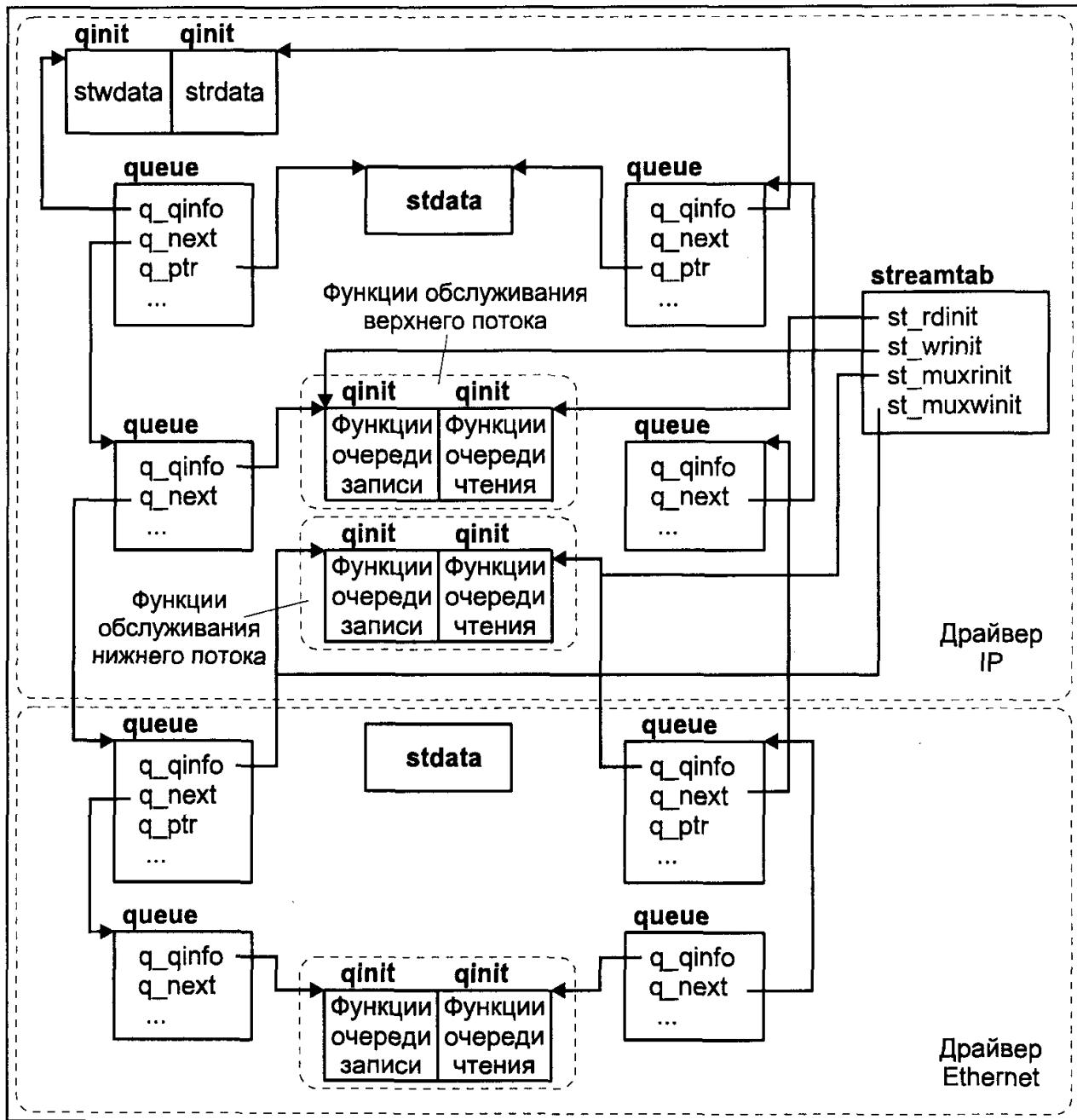


Рис. 5.25. Объединение верхнего и нижнего потоков

Во второй части главы была описана архитектура драйверов подсистемы STREAMS, имеющая модульную структуру и позволяющая более изящно осуществить буферизацию данных и управление их передачей. Вопросы, затронутые в этой части, будут также рассмотрены в следующей главе при обсуждении архитектуры сетевого доступа в операционной системы UNIX.

# 6

## **Поддержка сети в операционной системе UNIX**

Сегодня изолированный компьютер имеет весьма ограниченную функциональность. Дело даже не в том, что пользователи лишены возможности доступа к обширным информационным и вычислительным ресурсам, расположенным на удаленных системах. Изолированная система не имеет требуемой в настоящее время гибкости и масштабируемости. Возможность обмена данными между рассредоточенными системами открыла новые горизонты для построения распределенных ресурсов, их администрирования и наполнения, начиная от распределенного хранения информации (сетевые файловые системы, файловые архивы, информационные системы с удаленным доступом), и заканчивая сетевой вычислительной средой. UNIX — одна из первых операционных систем, которая обеспечила возможность работы в сети. И в этом одна из причин ее потрясающего успеха и долгожительства.

Хотя многие версии UNIX сегодня поддерживают несколько сетевых протоколов, в этой главе мы подробнее остановимся на наиболее известном и распространенном семействе под названием TCP/IP. Эти протоколы были разработаны, а затем прошли долгий путь усовершенствований для обеспечения требований феномена XX века — глобальной сети Internet. Протоколы TCP/IP используются практически в любой коммуникационной среде, от локальных сетей на базе технологии Ethernet или FDDI, до сверхскоростных сетей ATM, от телефонных каналов точка-точка до трансатлантических линий связи с пропускной способностью в сотни мегабит в секунду.

Глава начинается с описания наиболее важных протоколов семейства TCP/IP — Internet Protocol (IP), User Datagram Protocol (UDP) и Transmission Control Protocol (TCP). Здесь описываются стандартная спецификация этих протоколов и особенности реализации их алгоритмов, не определенные стандартами, но позволяющие значительно повысить производительность работы в сети.

Далее обсуждается программный интерфейс доступа к протоколам TCP/IP. При этом рассматриваются два основных интерфейса — традиционный интерфейс работы с протоколами TCP/IP — интерфейс сокетов, изначаль-

но разработанный для системы BSD UNIX, и интерфейс ТЫ, позволяющий унифицированно работать с любыми сетевыми протоколами, соответствующими модели OSI. В конце раздела описан программный интерфейс более высокого уровня, позволяющий отвлечься от особенностей сетевых протоколов и полностью сосредоточиться на определении интерфейса и функциональности предоставляемых прикладных услуг. Эта система, которая называется **RPC** (Remote Procedure Call — удаленный вызов процедур), явилась предтечей современных систем разработки распределенных приложений, таких как **CORBA** (Common Object Request Broker), Java и т. д.

В последних разделах главы рассматривается архитектура сетевого доступа в двух основных ветвях операционной системы — BSD UNIX и UNIX System V.

## **Семейство протоколов TCP/IP**

В названии семейства присутствуют имена двух протоколов — TCP и IP. Это, конечно, не означает, что данными двумя протоколами исчерпывается все семейство. Более того, как будет видно, названные протоколы выполняют различные функции и используются совместно.

В 1969 году Агентство Исследований Defence Advanced Research Projects Agency (DAPRA) Министерства Обороны США начало финансирование проекта по созданию экспериментальной компьютерной сети коммутации пакетов (packet switching network). Эта сеть, названная ARPANET, была построена для обеспечения надежной связи между компьютерным оборудованием различных производителей. По мере развития сети были разработаны коммуникационные протоколы — набор правил и форматов данных, необходимых для установления связи и передачи данных. Так появилось семейство протоколов TCP/IP. В 1983 году TCP/IP был стандартизирован (MIL STD), в то же время агентство DAPRA начало финансирование проекта Калифорнийского университета в Беркли по поддержке TCP/IP в операционной системе UNIX.

Основные достоинства TCP/IP:

- Семейство протоколов основано на открытых стандартах, свободно доступных и разработанных независимо от конкретного оборудования или операционной системы. Благодаря этому TCP/IP является наиболее распространенным средством объединения разнородного оборудования и программного обеспечения.
- Протоколы TCP/IP не зависят от конкретного сетевого оборудования физического уровня. Это позволяет использовать TCP/IP в физических сетях самого различного типа: Ethernet, Token-Ring, X.25, т. е. практически в любой среде передачи данных.

- Протоколы этого семейства имеют гибкую схему адресации, позволяющую любому устройству однозначно адресовать другое устройство сети. Одна и та же система адресации может использоваться как в локальных, так и в территориально распределенных сетях, включая Internet.
- В семейство TCP/IP входят стандартизованные протоколы высокого уровня для поддержки прикладных сетевых услуг, таких как передача файлов, удаленный терминальный доступ, обмен сообщениями электронной почты и т. д.

## Краткая история TCP/IP

История создания и развития протоколов TCP/IP неразрывно связана с Internet — интереснейшим достижением мирового сообщества в области коммуникационных технологий. Internet является глобальным объединением разнородных компьютерных сетей, использующих протоколы TCP/IP и имеющих общее адресное пространство. Явление Internet уникально еще и потому, что эта глобальная сеть построена на принципах самоуправления (хотя ситуация отчасти начинает меняться). Однако вернемся к истории.

Сегодняшняя сеть Internet "родилась" в 1969 году, когда агентство DARPA получило заказ на разработку сети, получившей название ARPANET. Целью создания этой сети было определение возможностей использования коммуникационной технологии пакетной коммутации. В свою очередь, агентство DARPA заключило контракт с фирмой Bolt, Beranek and Newman (BBN). В сентябре 1969 года произошел запуск сети, соединивший четыре узла: Станфордский исследовательский институт (Stanford Research Institute), Калифорнийский университет в Санта-Барбаре (University of California at Santa Barbara), Калифорнийский университет в Лос-Анжелесе (University of California at Los Angeles) и Университет Юты (University of Utah). Роль коммуникационных узлов выполняли мини-компьютеры Honeywell 316, известные как Interface Message Processor (IMP).

Запуск и работа сети были успешными, что определило быстрый рост ARPANET. В то же время использованием сети в своих целях заинтересовались исследователи, далекие от военных кругов. Стали поступать многочисленные запросы от руководителей университетов США в Национальный научный фонд (National Science Foundation, NSF) с предложениями создания научно-образовательной компьютерной сети. В результате в 1981 году NSF одобрил и финансировал создание сети CSNET (Computer Science Network).

В 1984 году ARPANET разделилась на две различные сети: MILNET, предназначенную исключительно для военных приложений, и ARPANET для использования в "мирных" целях.

В 1986 году фонд NSF финансировал создание опорной сети, соединившей каналами с пропускной способностью 56 Кбит/с шесть суперкомпьютерных центров США. Сеть получила название NSFNET и просуществовала до 1995 года, являясь основной магистралью Internet. За это время пропускная способность опорной сети возросла до 45 Мбит/с, а число пользователей превысило 4 миллиона<sup>1</sup>.

Стремительное развитие NSFNET сделало бессмысленным дальнейшее существование ARPANET. В июне 1990 года Министерство обороны США приняло решение о прекращении работы сети. Однако уроки, полученные в процессе создания и эксплуатации ARPANET, оказали существенное влияние на развитие коммуникационных технологий, таких как локальные сети и сети пакетной коммутации.

При создании ARPANET был разработан и протокол сетевого взаимодействия коммуникационных узлов. Он получил название Network Control Program (NCP). Однако этот протокол строился на предположении, что сетевая среда взаимодействия является абсолютно надежной. Учитывая специфику ARPANET, такое предположение являлось, мягко говоря, маловероятным: качество коммуникационных каналов могло существенно изменяться в худшую сторону (особенно при предполагаемом использовании радио- и спутниковой связи), а отдельные сегменты сети могли быть разрушены<sup>2</sup>. Таким образом, подход к коммуникационной среде нуждался в пересмотре, и, как следствие, возникла необходимость разработки новых протоколов. Еще одной задачей, стоявшей перед разработчиками, являлось обеспечение согласованной работы связанных сетей (internet), использующих различные коммуникационные технологии (например, пакетное радио, спутниковые сети и локальные сети). Результатом исследований в этой области явилось рождение нового семейства протоколов — Internet Protocol (IP), с помощью которого осуществлялась базовая доставка данных в гетерогенной коммуникационной среде, и Transmission Control Protocol (TCP), который обеспечивал надежную передачу данных между пользователями в ненадежной сетевой инфраструктуре. Спецификации этих протоколов в 1973 году получили статус стандартов Министерства обороны MIL-STD-1777 и MIL-STD-1778 соответственно.

Общее число пользователей Internet на начало 1995 года составило 4852000, из них в США — более 3 миллионов. Уже к середине 1996 года сеть Internet имела следующие показатели: почти 13 миллионов хостов, 134 365 сетей, почти полмиллиона зарегистрированных доменов. На начало 1997 года население Internet по сведениям компании Network Wizards (<http://www.nw.com>) составляло 16 146 000 хостов (число записей в системе DNS), расположенных в 828 000 доменах. Правда, на запрос "откликнулось" в среднем около 3 миллионов хостов.

Принимая во внимание существовавшие в то время отношения между СССР и США, приходится констатировать, что такое вполне могло произойти. Сегодня предположение о надежности сети также не всегда справедливо, только роль бомб и ракет исполняют ковши экскаваторов.

## Архитектура TCP/IP

Архитектура семейства протоколов TCP/IP основана на представлении, что коммуникационная инфраструктура включает три объекта: процессы, хосты, и сети. Процессы являются основными коммуникационными объектами, поскольку между процессами, в конечном итоге, осуществляется передача информации. Выполнение процессов происходит на различных хостах (или компьютерах). Передача информации между процессами проходит через сети, к которым подключены хосты.

Подобный взгляд на вещи позволяет сделать основной вывод: чтобы доставить данные процессу, их необходимо сначала передать нужному хосту, а затем определенному процессу, который выполняется на этом хосте. Более того — эти две фазы могут выполняться независимо. Таким образом, от коммуникационной инфраструктуры требуется маршрутизация и доставка данных между хостами, а хосты, в свою очередь, обязаны обеспечить доставку нужным процессам.

Основываясь на этом простом соображении, при разработке семейства протоколов взаимодействия логичным было четкое распределение обязанностей между отдельными протоколами, представив их в виде нескольких уровней. Разработчиками было выбрано четыре уровня:

- Уровень приложений/процессов (Application/process layer)
- Транспортный уровень (Host-to-host layer)
- Уровень Internet (Internet layer)
- Уровень сетевого интерфейса (Network interface layer)

Уровень сетевого интерфейса составляют протоколы, обеспечивающие доступ к физической сети. С помощью этих протоколов осуществляется передача данных между коммуникационными узлами, подключенными к одному и тому же сетевому сегменту (например, сегменту Ethernet или каналу точка-точка). Протоколы этого уровня должны поддерживаться всеми активными устройствами, подключенными к сети (например, мостами). К этому уровню относятся протоколы Ethernet, IEEE802.X, SLIP, PPP и т. д. Протоколы уровня сетевого интерфейса формально не являются частью семейства TCP/IP, однако стандарты Internet определяют, каким образом должна осуществляться передача данных TCP/IP с использованием вышеуказанных протоколов.

Уровень Internet составляют протоколы, обеспечивающие передачу данных между хостами, подключенными к различным сетям. Одной из функций, которая должна быть реализована протоколами этого уровня, является выбор маршрута следования данных, или *маршрутизация*. Сетевые элементы, осуществляющие передачу данных из одной сети в другую, получили

название *шлюзов* (gateway)<sup>3</sup>. Шлюз имеет несколько сетевых интерфейсов, подключенных к различным физическим сетям, и его основной задачей является выбор маршрута передачи данных из одного сетевого интерфейса в другой. Основной представитель уровня Internet — протокол IP.

Протоколы транспортного уровня обеспечивают передачу данных между процессами, выполняющими на разных хостах. Помимо этого транспортные протоколы могут реализовывать дополнительные функции, например, гарантированную доставку, создание виртуального канала и т. д. К транспортному уровню относятся протоколы TCP и UDP.

Наконец, протоколы уровня приложений обеспечивают функционирование прикладных услуг, таких как удаленный терминальный доступ, копирование удаленных файлов, передача почтовых сообщений и т. д. Работу этих приложений обеспечивают протоколы Telnet, File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP) и т. д.

На рис. 6.1 показана иерархическая четырехуровневая модель семейства протоколов TCP/IP. Заметим, что протоколы уровня сетевого интерфейса, фактически не являются частью семейства, поскольку не определены ни стандартами Министерства обороны США, ни стандартами Internet. Вместо этого используются существующие протоколы сети и определяются методы передачи трафика TCP/IP с помощью данной коммуникационной технологии. Например, RFC894 (A Standard for the Transmission of IP Datagrams over Ethernet Networks) определяет формат и процедуру передачи IP-пакетов в сетях Ethernet, а RFC 1577(Classical IP and ARP over ATM) — в сетях ATM.

На рис. 6.2 показана базовая коммуникационная схема протоколов TCP/IP. Коммуникационная инфраструктура может состоять из нескольких физических сетей. Для передачи данных в физической сети между подключенными хостами используется некоторый протокол уровня сетевого интерфейса, определенный для данной технологии передачи данных (Ethernet, FDDI, ATM и т. д.). Отдельные сети связаны между собой шлюзами, — устройствами, подключенными одновременно к нескольким сетям и служащими для передачи пакетов данных из одного интерфейса в другой. Выполнение этой функции обеспечивается протоколом IP. Как видно из рисунка, протокол IP выполняется на хостах и шлюзах и в конечном итоге обеспечивает доставку данных от хоста-отправителя к хосту-получателю. За обмен данными между процессами отвечают протоколы транспортного уровня — TCP или UDP. Поскольку работа транспортных

Более точным названием этих устройств является "маршрутизатор" (router). С формальной точки зрения термин "шлюз", применительно к данным устройствам, не совсем верен. Модель OSI определяет шлюз, как устройство, которое может осуществлять функции передачи на всех семи уровнях (подробнее о модели OSI будет рассказано в следующем разделе). Однако в мире UNIX маршрутизаторы почему-то называют шлюзами, и мы будем придерживаться этой терминологии.

протоколов обеспечивает передачу данных между удаленными процессами, протоколы этого уровня должны быть реализованы на хостах. При этом шлюзов для TCP или UDP как бы не существует, поскольку их присутствие и работу полностью скрывает протокол IP. Наконец, процессы также используют некоторый протокол для обмена данными, например Telnet или FTP.

Передача файлов	Электронная почта	Эмуляция терминала	Сетевая файловая система	Менеджмент сети	Уровень приложений процессов				
FTP RFC 959	SMTP RFC 821	Telnet RFC 854	NFS	SNMP RFC 1157					
Transmission Control Protocol TCP RFC 793			User Datagram Protocol UDP RFC 768		Транспортный уровень				
Address Resolution ARP RFC 826 RARP RFC 903	Internet Protocol IP RFC 791		Internet Control Message Protocol ICMP RFC 792		Уровень Internet				
Ethernet, Token Ring, FDDI, serial, ATM					Уровень сетевого интерфейса				
Витая пара, коаксиальный кабель, волоконно-оптический кабель, радио-релейная линия, спутниковый канал									

Рис. 6.1. Архитектура протоколов TCP/IP

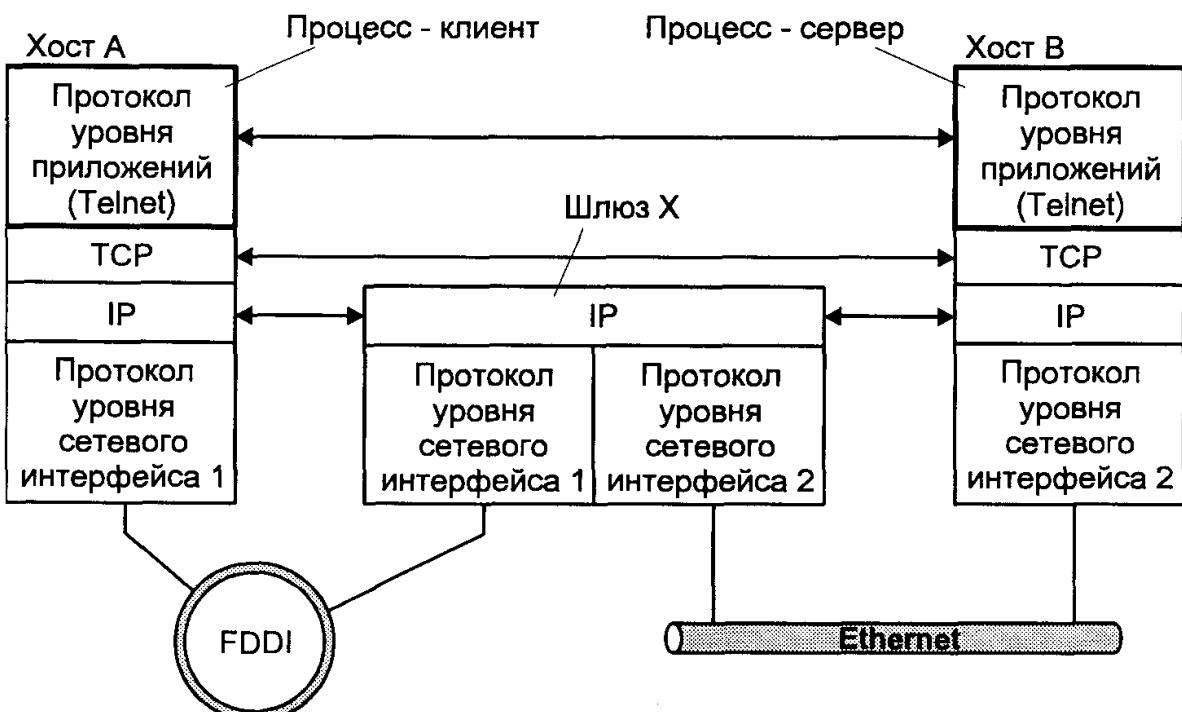


Рис. 6.2. Коммуникационная схема TCP/IP

Для правильного обмена данными каждый коммуникационный узел должен иметь уникальный адрес. На самом деле, как правило, существует несколько уровней адресации. Например, в локальной сети, каждый сетевой интерфейс (первый уровень модели) имеет т. н. *MAC-адрес*. С помощью этого адреса обеспечивается доставка данных требуемому получателю в физической сети. Для доставки данных IP необходимо адресовать хост-получатель. Для этого используется т. н. *IP-* или *Internet-адрес*. Наконец, хост, получивший данные, должен доставить их требуемому процессу. Таким образом, каждый процесс хоста, участвующий в коммуникационном взаимодействии также имеет адрес. Этот адрес получил название *номера порта*.

Таким образом, для того чтобы однозначно адресовать принимающую сторону, отправитель данных должен указать адреса хоста (IP-адрес) и процесса на этом хосте (номер порта). Он также должен указать, какой протокол транспортного уровня будет использован при обмене данными (номер протокола). Поскольку путь данных может проходить по нескольким физическим сегментам, физический адрес, или MAC-адрес, сетевого интерфейса не имеет смысла и определяется автоматически на каждом этапе пересылки (hop) между шлюзами.

Попробуем вкратце рассмотреть процесс передачи данных от процесса 2000 (номер порта), выполняющегося на хосте А, к процессу 23, выполняющемуся на хосте В. Согласно рис. 6.2 хосты расположены в разных физических сегментах, соединенных шлюзом X. Для этого процесс 2000 передает некоторые данные модулю протокола TCP (допустим, что приложение использует этот транспортный протокол), указывая, что данные необходимо передать процессу 23 хоста В. Модуль TCP, в свою очередь, передает данные модулю IP, указывая при этом только адрес хоста В. Модуль IP выбирает маршрут и соответствующий ему сетевой интерфейс (если их несколько) и передает последнему данные, указывая шлюз X в качестве промежуточного получателя.

Можно заметить, что наряду с передачей данных, каждый уровень обработки передает последующему некоторую управляющую информацию (IP-адрес, номер порта и т. д.). Эта информация необходима для правильной доставки данных адресату. Поэтому каждый протокол формирует пакет (Protocol Data Unit, PDU), состоящий из данных, переданных модулем верхнего уровня, и заголовка, содержащего управляющую информацию. Эта управляющая информация распознается модулем того же уровня (peer module) удаленного узла и используется для правильной обработки данных и передачи их соответствующему протоколу верхнего уровня.

На рис. 6.3 схематически показан процесс обработки данных при их передаче между хостами сети с использованием протоколов TCP/IP. С точки зрения процессов 23 и 2000 между ними существует коммуникационный

канал, обеспечивающий надежную и достоверную передачу потока данных, внутреннюю структуру которого определяют сами процессы по предварительной договоренности (например, в соответствии с протоколом Telnet). Модуль TCP хоста А обменивается сегментами данных с парным ему модулем TCP хоста В, не задумываясь о топологии сети или физических интерфейсах. Задача модулей TCP заключается в обеспечении достоверной и последовательной передачи данных между модулями приложений (процессов). TCP не интерпретирует прикладные данные и ему безразлично, передается ли в сегменте фрагмент почтового сообщения, файл или регистрационное имя пользователя. В свою очередь модуль IP хоста А передает данные, полученные от транспортных протоколов, модулю IP хоста В, не заботясь о надежности и последовательности передачи. Он не интерпретирует данные TCP, поскольку его задача — правильно адресовать отправляемую датаграмму. Поэтому модулю IP все равно, передает ли он данные TCP или UDP, управляющие сегменты или инкапсулированные прикладные данные.

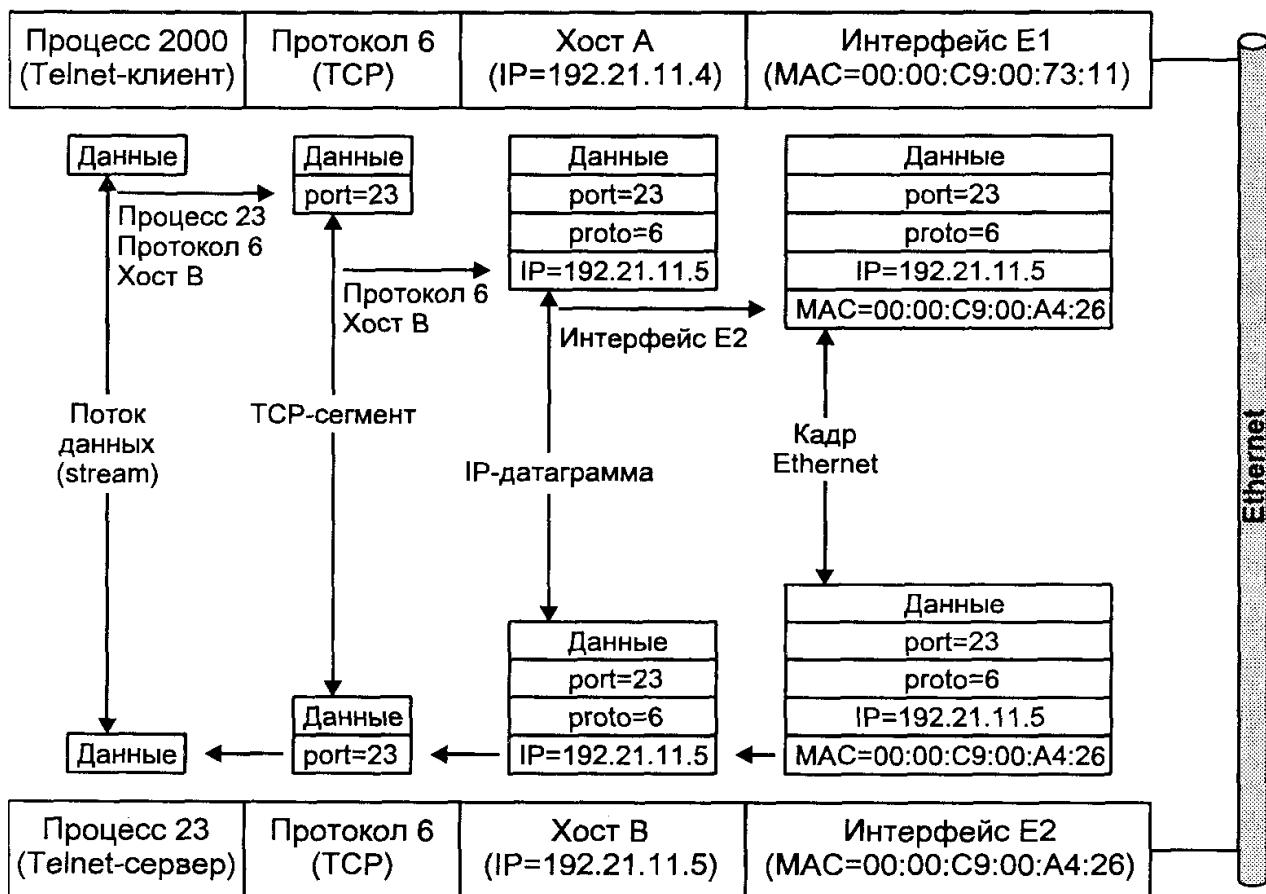


Рис. 6.3. Обработка данных в соответствии с протоколами TCP/IP

Работу модулей TCP/IP можно сравнить со сборочным конвейером: каждый участок выполняет определенную для него задачу, полагаясь на качество работы, выполненной на предыдущем этапе.

## Общая модель сетевого взаимодействия OSI

При знакомстве с семейством протоколов TCP/IP мы отметили уровневую структуру этих протоколов. Каждый из уровней выполняет строго определенную функцию, изолируя в то же время особенности этой обработки и связанные с ней данные от протоколов верхнего уровня. Четкое определение интерфейсов между протоколами соседних уровней позволяет выполнять разработку и реализацию протоколов независимо, не внося изменений в другие модули системы. Характерным примером является интерфейс между протоколом IP и протоколами транспортного уровня TCP и UDP. Хотя последние выполняют различную обработку, их взаимодействие с IP идентично.

Развитие сетевых технологий и связанных с ними протоколов обмена данными наглядно показало необходимость стандартизации этого процесса. Вместе с тем было очевидно, что единый стандарт на все случаи жизни не может решить подобную задачу. Очевидно также, что коммуникационная архитектура должна иметь модульную структуру, в которой модули обладают стандартными интерфейсами взаимодействия и могут подключаться в соответствии с этими интерфейсами, образуя "конвейер" обработки данных. Все это позволяет считать наиболее жизнеспособным подход, когда в рамках общей модели или архитектуры сетевого взаимодействия стандартизируются интерфейсы и функциональность отдельных модулей.

Такая общая модель была принята в 1983 году Международной организацией по стандартизации (International Organization for Standardization, ISO), и получила название модели взаимодействия открытых систем (Open Systems Interconnection, OSI). Эта модель является основой для объединения разнородных компьютеров в гетерогенную сетевую инфраструктуру. Данная архитектура определяет возможность установления соединения между любыми двумя системами, удовлетворяющими модели и поддерживающими соответствующие стандарты.

В модели OSI, как и в TCP/IP, общая функциональность системы разделена на несколько уровней, каждый из которых выполняет свою часть функций, необходимых для установления соединения с парным ему уровнем удаленной системы. В то же время каждый из уровней выполняет определенную обработку данных, реализуя набор услуг для уровня выше. Описание услуг и формат их предоставления определяются внутренним протоколом взаимодействия соседних уровней и определяют межуровневый интерфейс.

Модель OSI состоит из семи уровней, краткое описание которых приведено в табл. 6.1.

**Таблица 6.1.** Семь уровней модели OSI

Название уровня	Описание
Уровень приложений (Application layer)	Обеспечивает пользовательский интерфейс доступа к распределенным ресурсам
Уровень представления (Presentation layer)	Обеспечивает независимость приложений от различий в способах представления данных
Уровень сеанса (Session layer)	Обеспечивает взаимодействие прикладных программ в сети
Транспортный уровень (Transport layer)	Обеспечивает прозрачную передачу данных между конечными точками сетевых коммуникаций. Отвечает за восстановление ошибок и контроль за потоком данных
Сетевой уровень (Network layer)	Обеспечивает независимость верхних уровней от конкретной реализации способа передачи данных по физической среде. Отвечает за установление, поддержку и завершение сетевого соединения
Уровень канала данных (Data link layer)	Обеспечивает надежную передачу данных по физической сети. Отвечает за передачу пакетов данных — кадров и обеспечивает необходимую синхронизацию, обработку ошибок и управление потоком данных
Физический уровень (Physical layer)	Отвечает за передачу неструктурированного потока данных по физической среде. Определяет физические характеристики среды передачи данных

Рассмотрим процесс передачи данных между удаленными системами в рамках модели OSI. Пусть пользователю А системы С1 необходимо передать данные приложению В системы С2. Обработка прикладных данных начинается на уровне приложения. Уровень приложения передает обработанные данные и управляющую информацию на следующий уровень — уровень представления и т. д., пока данные наконец не достигнут физического уровня и не будут переданы по физической сети. Система С2 принимает эти данные и обрабатывает их в обратном порядке, начиная с физического уровня и заканчивая уровнем приложения, после чего исходные прикладные данные будут получены пользователем В.

Для того чтобы каждый уровень мог правильно обработать полученные данные, последние содержат также управляющую информацию. Эта управляющая информация интерпретируется только тем уровнем, для которого она предназначена, в соответствии с его протоколом, и невидима для других уровней: для верхних, потому что после обработки она удаляется, а для нижних — потому, что представляется им как обычные данные. Благодаря этому каждый уровень по существу общается с расположенным на удаленной системе равным (peer) ему уровнем. Таким образом, взаимодействие между удаленными системами можно представить состоящим из нескольких логических каналов, соответствующих уровням модели, передача данных в каждом из которых определяется протоколом своего уровня.

Так физический уровень и уровень канала данных обеспечивают коммуникационный канал сетевому уровню, который, в свою очередь, предоставляет связность объектам транспортного уровня и т. д.

Нетрудно заметить, что модель TCP/IP отличается от модели OSI. На рис. 6.4 показана схема отображения архитектуры TCP/IP на модель OSI. Видно, что соответствие существует для уровня Internet (сетевой уровень) и транспортного уровня. Уровни сеанса, представления и приложений OSI в TCP/IP представлены одним уровнем приложений. Обсуждение соответствия двух моделей носит весьма теоретический характер, поэтому мы перейдем к болеециальному для практики обсуждению прекрасно зарекомендовавших себя протоколов Internet.

TCP/IP		OSI
		Уровень приложений (Application layer)
Уровень приложений (Application layer)		Уровень представления (Presentation layer)
		Уровень сеанса (Session layer)
Транспортный уровень (Host-to-Host layer)		Транспортный уровень (Transport layer)
Уровень Internet (Internet layer)		Сетевой уровень (Network layer)
Уровень сетевого (Network interface layer)		Уровень канала данных (Data link layer)
		Физический уровень (Physical layer)
Рис. 6.4. Соответствие между моделями TCP/IP и OSI		

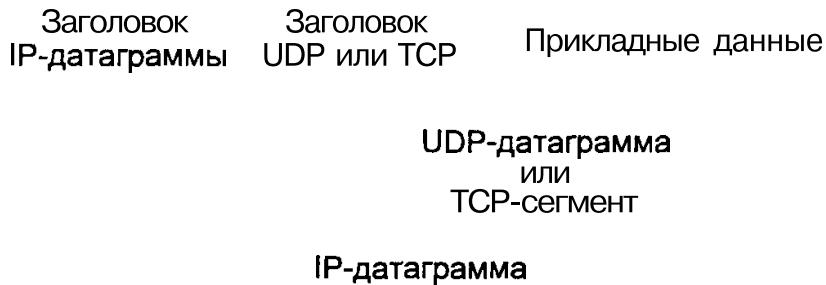
## Протокол IP

Межсетевой протокол (Internet Protocol, IP) обеспечивает доставку фрагмента данных (датаграммы) от источника к получателю через систему связанных между собой сетей. В протоколе IP отсутствуют функции подтверждения, контроля передачи, сохранения последовательности передаваемых датаграмм и т. д. В этом смысле протокол IP обеспечивает потенциально ненадежную передачу. Надежность и прочие функции, отсутствующие у IP, при необходимости реализуются протоколами верхнего уровня. Например, протокол TCP дополняет IP функциями подтверждения и управления передачей, позволяя приложениям (или протоколам более высокого

уровня) рассчитывать на получение упорядоченного потока данных, свободных от ошибок. Эта функциональность может быть реализована и протоколами более высокого уровня, как например это сделано в реализации распределенной файловой системы NFS, традиционно работающей на базе "ненадежного" транспортного протокола UDP. При этом работа NFS в целом является надежной.

В рамках модели OSI протокол IP занимает 3-й уровень и, таким образом, взаимодействует с протоколами управления передачей снизу и транспортными протоколами сверху. В рамках этой модели IP выполняет три основные функции: адресацию, фрагментацию и маршрутизацию данных.

Данные, формат которых понятен протоколу IP, носят название *датаграммы* (datagram), вид которой приведен на рис. 6.5. Датаграмма состоит из заголовка, содержащего необходимую управляющую информацию для модуля IP, и данных, которые передаются от протоколов верхних уровней и формат которых неизвестен IP. Вообще говоря, термин "датаграмма" обычно используется для описания пакета данных, передаваемого по сети без установления предварительной связи (connectionless).



**Рис. 6.5. IP-датаграмма**

Протокол IP обрабатывает каждую датаграмму как самостоятельный объект, не зависящий от других передаваемых датаграмм. Для датаграмм не применимы виртуальные каналы или другие логические тракты передачи.

Модули IP производят передачу датаграммы по направлению к получателю на основании адреса, расположенного в заголовке IP-датаграммы. Выбор пути передачи датаграммы называется *маршрутизацией*.

В процессе обработки датаграммы протокол IP иногда вынужден выполнять ее *фрагментацию*. Фрагментация бывает необходима, поскольку путь датаграммы от источника к получателю может пролегать через локальные и территориально-распределенные физические сети различной топологии и архитектуры, использующие различные размеры кадра. Например, кадр FDDI позволяет передавать датаграммы размером до 4470 октетов, в то время как сети Ethernet накладывают ограничение в 1500 октетов.

Заголовок IP-датаграммы, позволяющий модулю протокола выполнить необходимую обработку данных, приведен на рис. 6.6.

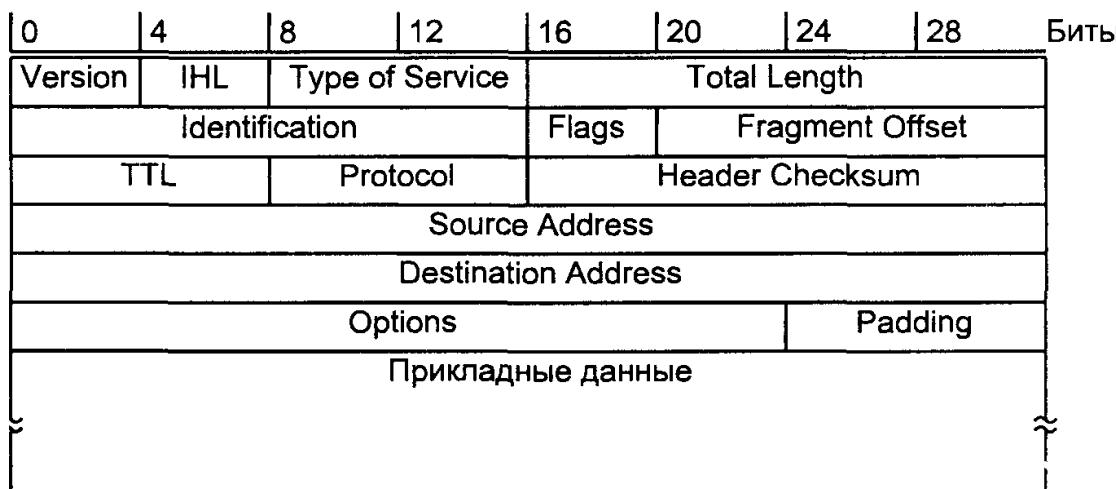


Рис. 6.6. Заголовок IP-датаграммы

Заголовок занимает как минимум 20 октетов управляющих данных. Поле Version определяет версию протокола и ее значение равно 4 (для IPv4). Поле IHL (Internet Header Length) указывает длину заголовка в 32-битных словах. При минимальной длине заголовка в 20 октетов значение IHL будет равно 5. Это поле также используется для определения смещения, начиная с которого размещаются управляющие данные протоколов верхнего уровня (например, заголовок TCP). Поле Type of Service определяет требуемые характеристики обработки датаграммы и может принимать следующие значения:

Биты 0–2      **Precedence.** Относительная значимость датаграммы. Это поле может использоваться рядом сетей, при этом большее значение поля Precedence соответствует более приоритетному трафику (например, при перегрузке сети модуль передает только трафик со значением Precedence выше определенного порогового значения).

Бит 3      **Delay.** Задержка. Значение 0 соответствует нормальной задержке при обработке, значение 1 — низкому значению задержки.

Бит 4      **Throughput.** Скорость передачи. Значение 0 соответствует нормальной скорости передачи, значение 1 — высокой скорости.

Бит 5      **Reliability.** Надежность. Значение 0 соответствует нормальной надежности, значение 1 — высокой надежности.

Биты 6–7      Зарезервированы для последующего использования.

Поле Type of Service определяет обработку датаграммы при передаче через различные сети от источника к получателю. В большинстве случаев может оказаться невозможным удовлетворение сразу всех требований по

обработке, предусмотренных полем Type of Service. Например, удовлетворение требования низкого значения задержки, может сделать невозможным повышение надежности передачи. Фактическое отображение параметров Type of Service на процедуры обработки конкретной сети зависит от архитектуры этой сети. Примеры возможных отображений можно найти в RFC 795 "Service mappings".

Поле Total Length содержит общий размер датаграммы в октетах. Размер поля (16 бит) ограничивает максимальный размер IP-датаграммы 65535 октетами.

Следующее 32-битное слово используется при фрагментации и последующем реассемблировании датаграммы. Фрагментация необходима, например, когда датаграмма отправляется из сети, позволяющей передачу пакетов, размер которых превышает максимальный размер пакета какой-либо из сетей по пути следования датаграммы к получателю. В этом случае IP-модуль, вынужденный передать "большую" датаграмму в сеть с малым размером кадра, должен разбить ее на несколько датаграмм меньшего размера. Вообще говоря, модуль протокола должен обеспечивать возможность фрагментации исходной датаграммы на произвольное число частей (фрагментов), которые впоследствии могут быть реассемблированы получателем. Получатель фрагментов отличает фрагменты одной датаграммы от другой по полю Identification. Это поле устанавливается при формировании исходной датаграммы и должно быть уникальным для каждой пары источник-получатель на протяжении жизни датаграммы в сети. Поле Fragment Offset указывает получателю на положение данного фрагмента в исходной датаграмме.

Поле Flags содержит следующие флаги:

- |       |  |
|-------|--|
| Бит 0 | Зарезервирован   |
| Бит 1 | DF. Значение 0 позволяет фрагментировать датаграмму. Значение 1 запрещает фрагментацию. Если в последнем случае передача исходной датаграммы невозможна, модуль протокола просто уничтожает исходную датаграмму без уведомления              |
| Бит 2 | MF. Значение 0 указывает, что данный фрагмент является последним в исходной датаграмме (в исходной датаграмме значение MF равно 0). Значение 1 сообщает реассемблирующему модулю о том, что данный фрагмент исходной датаграммы не последний |

Для фрагментации датаграммы большого размера модуль протокола формирует две или более новых датаграмм и копирует содержимое заголовка исходной датаграммы в заголовки вновь созданных. Флаг MF устанавливается равным 1 для всех датаграмм, кроме последней, для которой значение этого флага копируется из исходной датаграммы. Данные разбиваются на

необходимое число частей с сохранением 64-битной границы. Соответствующим образом устанавливаются значения полей Total Length и Fragment Offset.

Получатель фрагментов, например хост, производит реассемблирование, объединяя датаграммы с равными значениями четырех полей: Identification, адрес источника (Source Address), адрес получателя (Destination Address) и Protocol. При этом положение фрагмента в объединенной датаграмме определяется полем Fragment Offset.

Следующее поле заголовка называется TTL (Time-to-Live) и определяет "время жизни" датаграммы в сети. Если значение этого поля становится равным 0, датаграмма уничтожается. Каждый модуль протокола, обрабатывающий датаграмму, уменьшает значение этого поля на число секунд, затраченных на обработку. Однако поскольку обработка датаграммы в большинстве случаев занимает гораздо меньшее время, а TTL все равно уменьшается на 1, то фактически это поле определяет максимальное количество хопов (число промежуточных передач через шлюзы), которое датаграмма может совершить. Смысл этой функции — исключить возможность засорения сети "заблудившимися" датаграммами.

Поле Protocol определяет номер протокола верхнего уровня, которому предназначена датаграмма. Значения этого поля для различных протоколов приведены в RFC 1700 "Assigned numbers", некоторые из них показаны в табл. 6.2.

**Таблица 6.2.** Некоторые номера протоколов

Номер	Протокол
1	Internet Control Message Protocol, ICMP
2	Internet Group Management Protocol, IGMP
4	Инкапсуляция IP в IP
6	Transmission Control Protocol, TCP
17	User Datagram Protocol, UDP
46	Resource Reservation Protocol, RSVP
75	Packet Video Protocol, PVP

Завершает третье 32-битное слово заголовка его 16-битная контрольная сумма/поле Header Checksum.

Поля Source Address и Destination Address содержат соответственно адреса источника датаграммы и ее получателя. Это адреса сетевого уровня, или IP-адреса, размер которых составляет 32 бита каждый.

Поле Options содержит различные опции протокола, а поле Padding служит для выравнивания заголовка до границы 32-битного слова.

## Адресация

Каждый IP-адрес можно представить состоящим из двух частей: адреса (или идентификатора) сети и адреса хоста в этой сети. Существует пять возможных форматов IP-адреса, отличающихся по числу бит, которые отводятся на адрес сети и адрес хоста. Эти форматы определяют *классы адресов*, получивших названия от А до D. Определить используемый формат адреса позволяют первые три бита, как это показано на рис. 6.7.

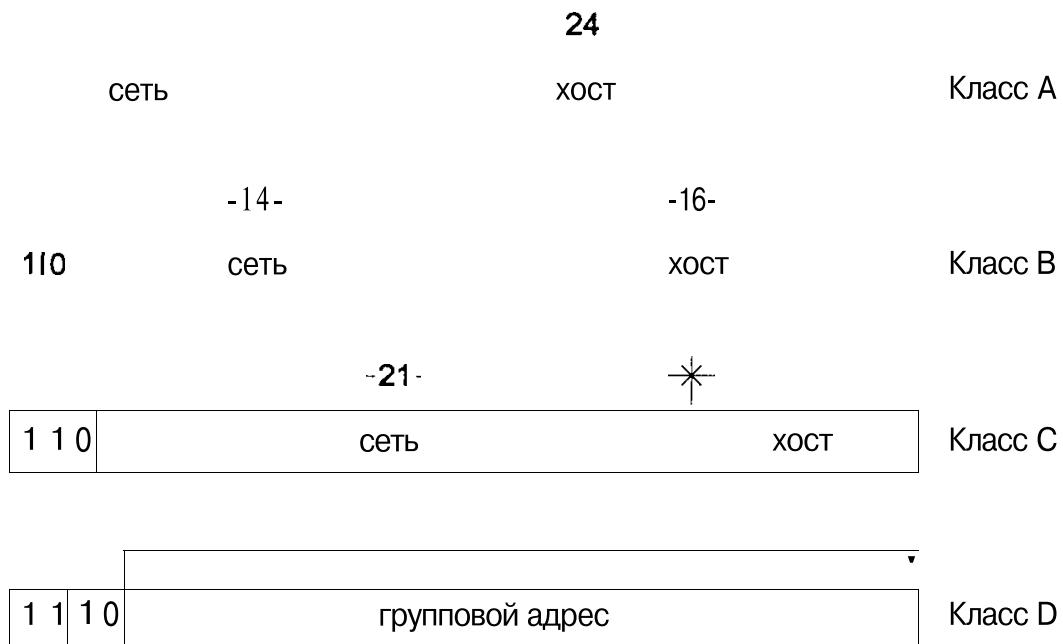


Рис. 6.7. Форматы IP-адресов

Взаимосвязанные сети (internet), должны обеспечивать общее адресное пространство. IP-адрес каждого хоста этих сетей должен быть уникальным. На практике это достигается с использованием иерархии, заложенной в базовый формат адреса. Некий центральный орган отвечает за назначение номеров сетей, следя за их уникальностью, в то время как администраторы отдельных сетей могут назначать номера хостов, также следя за уникальностью этих номеров в рамках собственной сети. В итоге — каждый хост получит уникальный адрес. В случае глобальной сети Internet уникальность адресов также должна выполняться глобально. За назначение адресов сетей отвечает центральная организация IANA, имеющая региональные и национальные представительства. При предоставлении зарегистрированного адреса сети вам гарантируется его уникальность.

Адреса класса А позволяют использовать 7 бит для адресации сети, ограничивая таким образом количество сетей этого класса числом  $126^4$ . Этот фор-

<sup>4</sup> Вообще-то 7 бит позволяют адресовать 128 сетей, но адреса сетей 0 и 127 являются зарезервированными. Это же правило для адреса сети, состоящего из всех нулей или всех единиц (в двоичном виде), справедливо и для остальных классов.

мат адреса напоминает формат, используемый в предтече современной глобальной сети Internet — сети ARPANET. В те времена мало кто мог предвидеть столь бурное развитие этих технологий и число 126 не казалось малым.

Число уникальных сетей класса В значительно больше — 16 382, поскольку адрес сети состоит из 14 бит. Однако сегодня и этого недостаточно — поэтому адреса сетей этого класса больше не предоставляются<sup>5</sup>.

В настоящее время выделяются сети класса С. Сетей такого класса в Internet может быть не более 2 097 150. Но и это число сегодня нельзя назвать большим. При этом в каждой сети класса С может находиться не более 254 хостов.

Популярность локальных сетей в середине 80-х годов и стремительный рост числа пользователей Internet в последнее десятилетие привели к значительному "истощению" адресного пространства. Дело в том, что если ваша организация использует только четыре адреса сети класса С, то остальные 250 адресов "потеряны" для сообщества Internet и использоваться не могут. Для более эффективного распределения адресного пространства была предложена дополнительная иерархия IP-адреса. Теперь адрес хоста может в свою очередь быть разделен на две части — адрес подсети (subnet) и адрес хоста в подсети.

Заметим, что подсети по-прежнему являются отдельными сетями для протокола IP, требующими наличия маршрутизатора для передачи датаграмм из одной подсети в другую.

Для определения фактической границы между адресом подсети и хоста используется *маска сети*, представляющая собой 32-битное число, маскирующее единицами (в двоичном виде) номера сети и подсети и содержащее нули в позициях номера хоста. Модуль протокола IP производит логическую операцию "И" между маской и конкретным адресом, и таким образом определяет, предназначена ли эта датаграмма данному хосту (для модуля протокола хоста), или датаграмма адресована непосредственно подключенной подсети, или ее необходимо передать другому шлюзу для последующей доставки. Использование маски сети показано на рис. 6.8.

Если хост или шлюз "не знает", какую маску использовать, он формирует сообщение ADDRESS MASK REQUEST (запрос маски адреса) протокола ICMP и направляет его в сеть, ожидая сообщения ICMP ADDRESS MASK REPLY от соседнего шлюза.

Ряд IP-адресов имеют специальное значение и не могут присваиваться сетевым элементам (хостам, шлюзам и т. д.). Эти значения приведены в табл. 6.3.

<sup>5</sup> Конечно, в изолированной сети (или сетях), не имеющей выхода в глобальную сеть Internet, вы вольны использовать адреса любого класса.

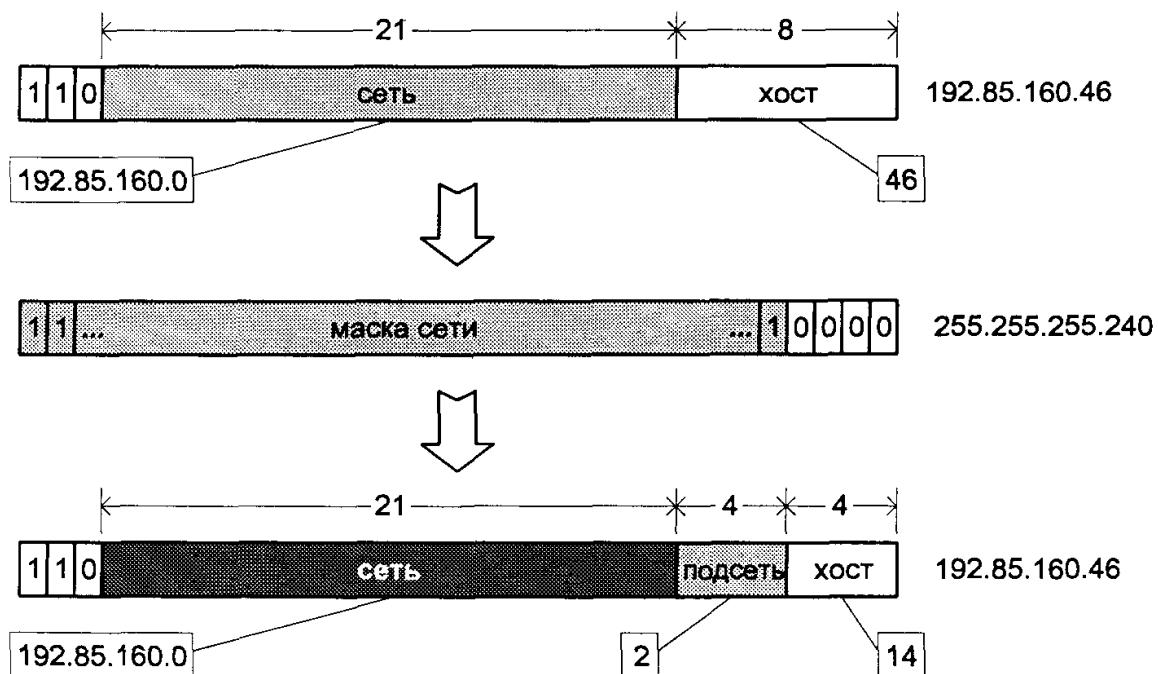


Рис. 6.8. Подсети

Таблица 6.3. Специальные IP-адреса

Адрес	Пример	Интерпретация
	Адрес: 192.85.160.46 Маска: 255.255.255.240	Адрес сети: 192.85.160.0 Адрес подсети: 2 Адрес хоста: 14
Сеть:0, Хост:0	0.0.0.0	Данный хост в данной сети
Сеть:0, Хост:Н	0.0.0.5	Определенный хост в данной сети (только для адреса источника)
Сеть:1111...1 Подсеть:1111...1 Хост:1111..1	255.255.255.255	Групповой адрес всех хостов данной подсети
Сеть:Н Подсеть:1111...1 Хост:1111...1	192.85.160.255	Групповой адрес всех хостов всех подсетей сети N
Сеть:Н Подсеть:С Хост:1111...1	192.85.160.47	Групповой адрес всех хостов подсети S сети N
Сеть: 127 Хост:1	127.0.0.1	Адрес внутреннего логического хоста

### Протоколы транспортного уровня

В соответствии с моделью DARPA, рассмотренной нами ранее, протоколы транспортного уровня работают исключительно на хостах, являющихся

точками обмена информацией — источниках или получателях датаграмм. Поскольку основная функция шлюзов заключается в выборе пути и последующей передаче датаграммы, которые непосредственно шлюзу не адресованы, протоколы этого уровня обычно не задействованы в шлюзах.

Два протокола этого уровня — TCP и UDP обеспечивают транспорт данных с заданными характеристиками между источником и получателем. Поскольку на каждом хосте как правило существует несколько процессов-получателей данных, протоколы этого уровня должны располагать необходимой информацией для доставки данных требуемому протоколу уровня приложений.

Как было показано, каждый уровень протоколов DARPA имеет собственную систему адресации. Например, для уровня сетевого интерфейса (соответствующего физическому уровню и уровню канала данных модели OSI) в локальных сетях используется физический адрес интерфейса. Он представляет собой 48-битный адрес, как правило, записанный в память платы. Для отображения физического адреса в адрес протокола верхнего уровня (Internet) используется специальный протокол трансляции адреса Address Resolution Protocol (ARP).

Уровень Internet (или сетевой уровень модели OSI) в качестве адресов использует уже рассмотренные нами IP-адреса. Для адресации протокола верхнего уровня используется поле Protocol заголовка IP-датаграммы.

Протоколы транспортного уровня замыкают систему адресации DARPA. Адреса, которые используются протоколами этого уровня и называются *номерами портов* (port number), служат для определения процесса (приложения), выполняющегося на данном хосте, которому адресованы данные. Другими словами, для передачи сообщения от источника к получателю требуется шесть адресов — по три с каждой стороны (физический адрес адаптера, IP-адрес и номер порта) — для однозначного определения пути. Номер порта адресует конкретный процесс (приложение) и содержится в заголовке TCP- или UDP-пакета. IP-адрес определяет сеть и хост, на котором выполняется процесс, и содержится в заголовке IP-датаграммы. Адрес сетевого адаптера определяет расположение хоста в физической сети.

Номера портов занимают 16 бит и стандартизированы в соответствии с их назначением. Полный список стандартных номеров портов приведен в RFC 1700 "Assigned Numbers". Часть из них в качестве примера приведена в табл. 6.4.

**Таблица 6.4.** Некоторые стандартные номера портов

<b>Номер порта</b>	<b>Название</b>	<b>Назначение (протокол уровня приложений)</b>
7	echo	Echo
20	ftp-data	Передача данных по протоколу FTP
21	ftp	Управляющие команды протокола FTP

Таблица 6.4 (продолжение)

Номер порта	Название	Назначение (протокол уровня приложений)
23	telnet	Удаленный доступ (Telnet)
25	smtp	Электронная почта (Simple Mail Transfer Protocol)
53	domain	Сервер доменных имен (Domain Name Server)
67	bootps	Сервер загрузки Bootstrap Protocol
68	bootpc	Клиент загрузки Bootstrap Protocol
69	tftp	Передача файлов (Trivial File Transfer Protocol)
70	gopher	Информационная система Gopher
80	www-http	World Wide Web (HyperText Transfer Protocol)
110	pop3	Электронная почта (POP версии 3)
119	nntp	Телеконференции (Network News Transfer Protocol)
123	ntp	Синхронизация системных часов (Network Time Protocol)
161	snmp	Менеджмент/статистика (Simple Network Management Protocol)
179	bgp	Маршрутизационная информация (Border Gateway Protocol)

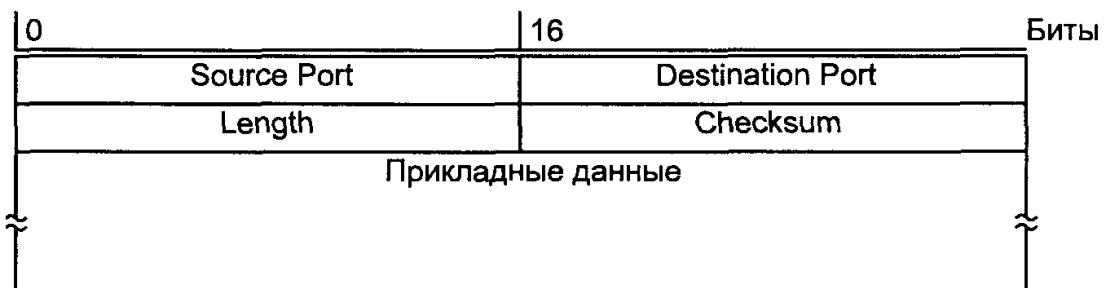
### User Datagram Protocol (UDP)

UDP является протоколом транспортного уровня и, как следует из названия, обеспечивает логический коммуникационный канал между источником и получателем данных без предварительного установления связи. Другими словами, сообщения, обрабатываемые протоколом не имеют друг к другу никакого отношения с точки зрения UDP. Для передачи датаграмм UDP использует протокол IP и так же, как и последний, не обеспечивает надежности передачи. Поэтому приложения, использующие этот транспортный протокол, должны при необходимости самостоятельно обеспечить надежность доставки, например, путем обмена подтверждениями и повторной передачей недоставленных сообщений.

Однако благодаря минимальной функциональности протокола UDP, передача данных с его использованием вносит гораздо меньшие накладные расходы по сравнению, скажем, с парным ему транспортным протоколом TCP. Размер заголовка UDP, показанного на рис. 6.9, составляет всего 8 октетов.

Первые два поля, каждое из которых занимает по 2 октета, адресуют соответственно порты источника и получателя. Указание порта источника яв-

ляется необязательным и это поле может быть заполнено нулями. Поле *Length* содержит длину датаграммы, которая не может быть меньше 8 октетов. Поле *Checksum* используется для хранения контрольной суммы и используется только если протокол верхнего уровня требует этого. Если контрольная сумма не используется, это поле заполняется нулями. В противном случае она вычисляется по *псевдозаголовку*, содержащему IP-адреса источника и получателя датаграммы и поле *Protocol* из IP-заголовка. Вид псевдозаголовка представлен на рис. 6.10. То, что вычисление контрольной суммы включает IP-адреса, гарантирует, что полученная датаграмма доставлена требуемому адресату. Заметим, что для протокола UDP значение поля *Protocol* равно 17.



**Рис. 6.9.** Заголовок UDP



**Рис. 6.10.** Псевдозаголовок UDP

В качестве примеров протоколов уровня приложений, которые используют в качестве транспортного протокола UDP, можно привести:

- Протокол взаимодействия с сервером доменных имен DNS, порт 53.
- Протокол синхронизации времени Network Time Protocol, порт 123.
- Протокол удаленной загрузки BOOTP, порты 67 и 68 для клиента и сервера соответственно.

- Протокол удаленного копирования Trivial FTP (TFTP), порт 69.
- Удаленный вызов процедур RPC, порт 111.

Для всех перечисленных протоколов и соответствующих им приложений предполагается, что в случае недоставки сообщения необходимые действия предпримет протокол верхнего уровня (приложение). Как правило, приложения, использующие протокол UDP в качестве транспорта, обмениваются данными, имеющими статистический повторяющийся характер, когда потеря одного сообщения не влияет на работу приложения в целом. Приложения, требующие гарантированной надежной доставки данных, используют более сложный протокол транспортного уровня, в значительной степени дополняющего функциональность протокола IP, — протокол TCP.

### **Transmission Control Protocol (TCP)**

TCP является протоколом транспортного уровня, поддерживающим надежную передачу потока данных с предварительным установлением связи между источником информации и ее получателем. На базе протокола TCP реализованы такие протоколы уровня приложений, как Telnet, FTP или HTTP.

Протокол TCP характеризуется следующими возможностями, делающими его привлекательным для приложений:

- Перед фактической передачей данных необходимо установление связи, т. е. запрос на начало сеанса передачи данных источником и подтверждение получателем. После обмена данными сеанс передачи должен быть явно завершен.
- Доставка информации является надежной, не допускающей дублирования или нарушения очередности получения данных.
- Возможность управления потоком данных для избежания переполнения и затора.
- Доставка экстренных данных.

Эти возможности протокола позволяют протоколам верхнего уровня и, соответственно, приложениям, их реализующим, не заботиться о надежности, последовательности доставки и т. д. Таким образом, протоколы приложений, использующие TCP, могут быть значительно упрощены. С другой стороны, это ведет к сложности самого транспортного протокола и, как следствие, к значительным накладным расходам при передаче данных.

TCP-канал представляет собой двунаправленный поток данных между соответствующими объектами обмена — источником и получателем. Данные могут передаваться в виде пакетов различной длины, называемых *сегментами*. Каждый TCP-сегмент предваряется заголовком, за которым следуют

данные, инкапсулирующие протоколы уровня приложения. Вид заголовка TCP-сегмента представлен на рис. 6.11.

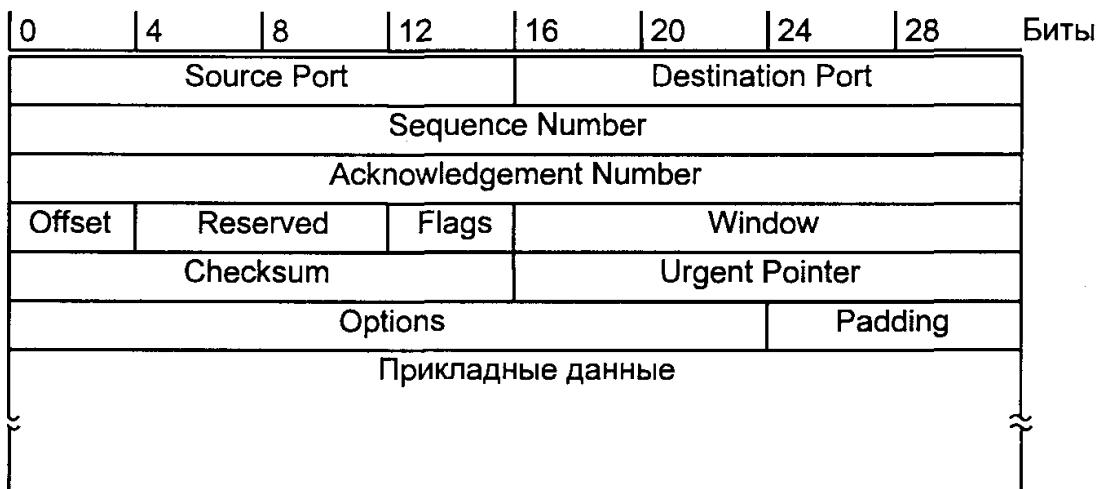


Рис. 6.11. Формат TCP-сегмента

Положение каждого сегмента в потоке фиксируется порядковым номером (Sequence Number), представленным соответствующим полем заголовка и обозначающим номер первого октета сегмента в потоке TCP. Порядковые номера также используются для подтверждения получения: каждый TCP-сегмент содержит номер подтверждения (Acknowledgement Number), сообщающий отправителю количество полученных от него последовательных данных. Номер подтверждения определяется как номер первого неподтвержденного октета в потоке.

И порядковый номер, и номер подтверждения занимают по 32 бита в заголовке TCP-сегмента, таким образом, их максимальное значение составляет  $(2^{32} - 1)$ , за которым следует 0. При установлении связи стороны договариваются о начальных значениях порядковых номеров (Initial Sequence Number, ISN) в каждом из направлений. Впоследствии первый октет переданных данных будет иметь номер (ISN+1).

Управление потоком данных осуществляется с помощью *метода скользящего окна* (sliding window). Каждый TCP-заголовок содержит также поле Window, которое указывает на количество данных, которое адресат готов принять, начиная с октета, указанного в поле Acknowledgement Number.

Заголовок TCP-сегмента занимает как минимум 20 октетов. Помимо рассмотренных нами порядковых номеров и анонсируемого окна, он содержит ряд других важных полей. Заголовок начинается с двух номеров портов, адресующих логические процессы на обоих концах виртуального канала. Далее следуют порядковый номер и номер подтверждения.

Поле смещения (Offset) указывает начало данных сегмента. Это поле необходимо, поскольку размер TCP-заголовка имеет переменную величину.

Значение этого поля измеряется в 32-битных словах. Таким образом, при минимальном размере заголовка поле `Offset` будет равно 5.

Далее заголовок содержит шесть управляющих флагов `Flags`, каждый из которых занимает отведенный ему бит:

URG	Указывает, что сегмент содержит экстренные данные, и поле <code>Urgent pointer</code> заголовка определяет их положение в сегменте.
ACK	Указывает, что заголовок содержит подтверждение полученных данных В поле <code>Acknowledgement Number</code> .
PSH	Указывает, что <b>буферизированные</b> данные должны быть переданы немедленно, не ожидая заполнения сегмента максимального размера.
RST	Указывает на необходимость уничтожения канала.
SYN	Указывает, что сегмент представляет собой управляющее сообщение, являющееся частью "тройного рукопожатия" для синхронизации порядковых номеров при создании канала.
FIN	Указывает, что сторона прекращает передачу данных и желает закрыть виртуальный канал.

Поле контрольной суммы `Checksum` используется для защиты от ошибок. Контрольная сумма вычисляется на основании 12-октетного псевдозаголовка, содержащего, в частности IP-адреса источника и получателя, а также номер протокола. Цель включения в контрольную сумму части заголовка IP та же, что и для протокола UDP — дополнительно защитить данные от получения не тем адресатом.

Поле `Urgent Pointer` позволяет указать расположение экстренных данных внутри сегмента. Это поле используется при установленном флаге `URG` и содержит порядковый номер октета, следующего за экстренными данными.

В конце заголовка располагается поле `Options` переменной длины, которое может содержать различные опции, например, максимальный размер сегмента (MSS). Это поле дополняется нулями (Padding) для того, чтобы заголовок всегда заканчивался на границе 32 бит.

## Состояния TCP-сессии

Как уже говорилось, передача данных с использованием протокола TCP предусматривает предварительное установление связи, или создание логического TCP-канала. Эта предварительная фаза призвана усилить надежность протокола. В процессе этой фазы определяется начало TCP-потоков в обоих направлениях, их характеристики (например, максимальный размер окна), в это же время могут быть обнаружены "полуразрушенные" TCP-каналы прошлых сеансов передачи, некорректно закрытые, напри-

мер, ввиду аварийного останова одной из сторон. Стороны выбирают произвольные начальные порядковые номера потоков, чтобы уменьшить вероятность обработки сегментов, принадлежащих "старым" сеансам<sup>6</sup>.

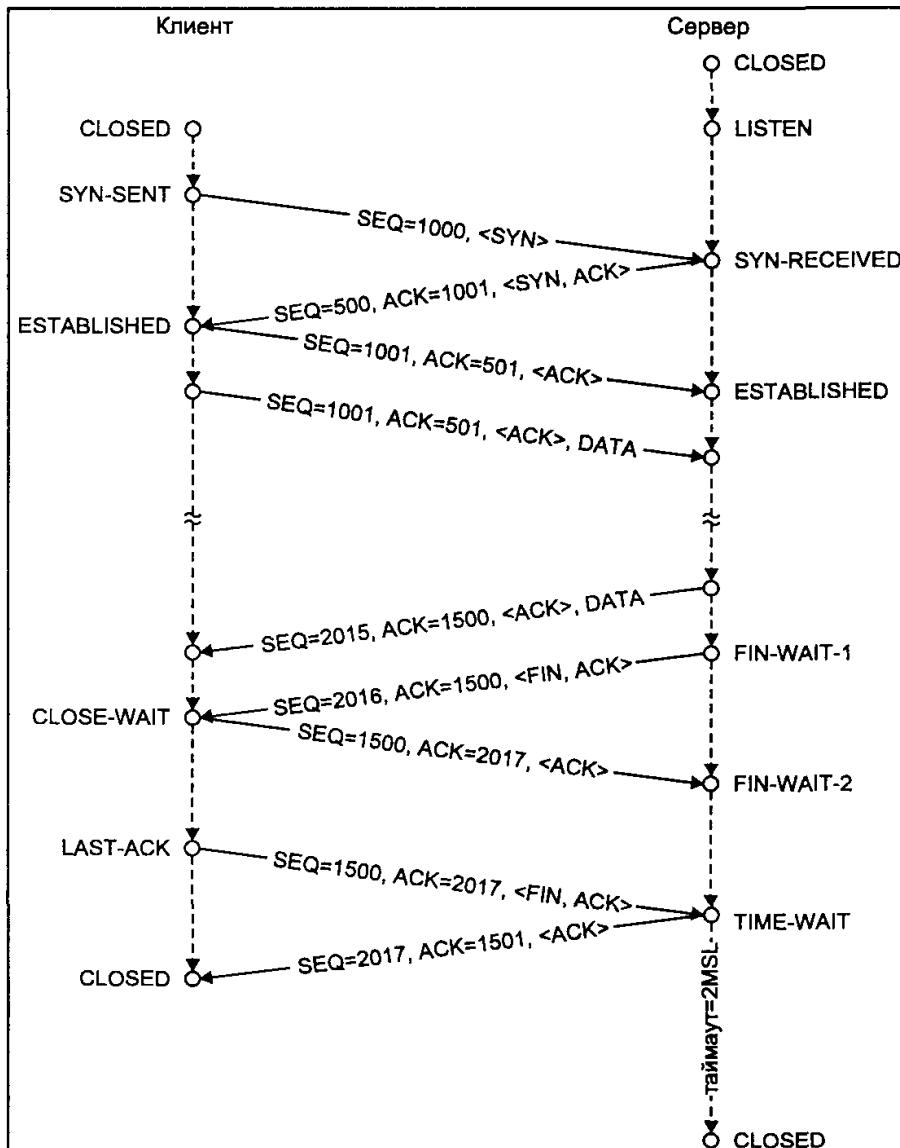
Начальная фаза сеанса передачи получила название "тройное рукопожатие" (three-way handshake), которое достаточно точно отражает процесс обмена служебными сегментами между сторонами. Этот процесс является асимметричным — одна из сторон, называемая клиентом, инициирует начало сеанса, посылая другой стороне — серверу сегмент SYN<sup>7</sup>. Как правило этот сегмент является числом служебным, т. е. не содержит полезных данных, его заголовок определяет номер порта и начальный порядковый номер потока клиент-сервер. Если сервер готов принять данные от клиента, он создает логический канал (размещая соответствующие структуры данных) и отправляет клиенту сегмент с установленным начальным порядковым номером потока сервер-клиент и флагами SYN и ACK, подтверждающим получение сегмента SYN и выражаяющим готовность сервера к получению данных. Наконец, и это третье рукопожатие, клиент отвечает сегментом с установленным флагом ACK, подтверждающим получение ответа от сервера и тем самым завершающим фазу создания TCP-канала. Процесс установления связи в TCP-сеансе представлен на рис. 6.12.

После этого обе стороны начинают передачу TCP-сегментов, каждый из которых содержит подтверждение полученных данных и новое значение окна. Начиная с подтвержденного октета, источник может передать, не дожидаясь подтверждения, количество данных, определенных значением окна. Если отправитель не получает подтверждения на посланные данные в течение определенного промежутка времени, он полагает, что данные утеряны, и их передача повторяется, начиная с последнего подтвержденного октета. Поскольку надежность передачи гарантируется протоколом, для данных приложения, переданных, но не подтвержденных, протокол хранит копию, которая уничтожается после получения подтверждения или вновь передается при отсутствии такового. Получение дублированных данных также подтверждается, хотя сами данные уничтожаются, поскольку дублирование могло быть вызвано неполучением подтверждения. Если одна из сторон получает неупорядоченные данные, они, как правило, сохраняются до получения недостающих последовательных сегментов. Разумеется, получение таких неупорядоченных данных не подтверждается, по-

<sup>6</sup> В нормальных условиях модули TCP хранят последние использованные порядковые номера. Поэтому при создании нового канала (сеанса) модуль выбирает следующее значение из адресного пространства порядковых номеров (которое составляет  $2^{32}$ ). При скорости передачи 2 Мбит/с потребуется 4,5 часа для передачи данных, адресуемых этими номерами (последовательными и подтверждений). Это время на несколько порядков превышает время жизни TCP-сегмента в сети, которое по умолчанию составляет 2 секунды. Это гарантирует, что новые номера не "догоняют" номера старых сегментов. Даже при скорости 100 Мбит/с полный цикл использования порядковых номеров составляет чуть больше 5 минут.

Сегмент SYN имеет установленный флаг SYN в заголовке — отсюда и его название.

скольку подтверждение отправляется только на полученный непрерывный последовательный поток октетов.



**Рис. 6.12.** Установление связи, передача данных и завершение TCP-сессии

Завершение сеанса в TCP происходит в несколько этапов. Любая из сторон может завершить передачу данных, отправив сегмент с установленным флагом FIN (рис. 6.12). Получение такого сегмента подтверждается другой стороной и эквивалентно достижению конца файла при его чтении. Однако другая сторона может продолжать передавать данные, также впоследствии завершив передачу сегментом FIN. Подтверждение этого сегмента полностью разрушает канал и завершает сеанс. Для того чтобы гарантировать синхронизацию завершения сеанса, сторона, отправившая подтверждение на последний сегмент FIN, должна поддерживать сеанс достаточно долго, чтобы иметь возможность вновь подтвердить повторные сегменты FIN данного сеанса в случае, когда подтверждение не было получено другой стороной.

На рис. 6.12 также проиллюстрированы состояния коммуникационных узлов TCP-канала.

Как видно из рисунка, начальное состояние узла (сервера или клиента) — состояние CLOSED. Готовность сервера к обработке инициирующих запросов от клиента определяется переходом его в состояние LISTEN. С этого момента сервер может принимать и обрабатывать инициирующие сеанс сегменты SYN. При отправлении такого сегмента клиент переходит в состояние SYN-SENT и ожидает ответного запроса от сервера. Сервер при получении сегмента также отправляет сегмент SYN с подтверждением ACK и переходит в состояние SYN-RECEIVED. Подтверждение от клиента завершает "рукопожатие" и сеанс переходит в состояние ESTABLISHED. После завершения обмена данными одна из сторон (например, клиент) отправляет сегмент FIN, переходя при этом в состояние FIN-WAIT-1. Приняв этот сегмент другая сторона (например, сервер) отправляет подтверждение ACK и переходит в состояние CLOSE-WAIT, при этом канал становится симплексным — передача данных возможна только в направлении от сервера к клиенту. Когда клиент получает подтверждение он переходит в состояние FIN-WAIT-2, в котором находится до получения сегмента FIN. После подтверждения получения этого сегмента канал окончательно разрушается.

Расшифровка состояний приведена в табл. 6.5.

**Таблица 6.5.** Состояния TCP-сеанса

<b>Состояние</b>	<b>Описание</b>
LISTEN	Готовность узла к получению запроса на соединение от любого удаленного узла.
SYN-SENT	Ожидание ответного запроса на соединение.
SYN-RECEIVED	Ожидание подтверждения получения ответного запроса на соединение.
ESTABLISHED	Состояние канала, при котором возможен дуплексный обмен данными между клиентом и сервером.
CLOSE-WAIT	Ожидание запроса на окончание связи от локального процесса, использующего данный коммуникационный узел.
LAST-ACK	Ожидание подтверждения запроса на окончание связи, отправленного удаленному узлу. Предварительно от удаленного узла уже был получен запрос на окончание связи и канал стал симплексным.
FIN-WAIT-1	Ожидание подтверждения запроса на окончание связи, отправленного удаленному узлу (инициирующий запрос, канал переходит в симплексный режим).
FIN-WAIT-2	Ожидание запроса на окончание связи от удаленного узла
CLOSING	Ожидание подтверждения от удаленного узла на запрос окончания связи.

**Таблица 6.5(продолжение)**

<b>Состояние</b>	<b>Описание</b>
TIME-WAIT	Таймаут перед окончательным разрушением канала, достаточный для того, чтобы удаленный узел получил подтверждение своего запроса окончания связи. Величина тайм-аута составляет 2 MSL (Maximum Segment Lifetime) <sup>8</sup> .
CLOSED	Фиктивное состояние, при котором коммуникационный узел и канал фактически не существуют.

Для обеспечения правильной обработки данных для каждого логического TCP-канала хранится полная информация о его состоянии, различных таймерах и о текущих порядковых номерах переданных и принятых октетов. Это необходимо, например, для корректной обработки служебных сегментов SYN и FIN<sup>9</sup>.

### **Передача данных**

После создания виртуального канала взаимодействующие процессы получают возможность обмениваться данными в дуплексном режиме.

Хотя фактически передача данных осуществляется в виде сегментов, ее логический вид представляет собой последовательный поток октетов, каждый из которых адресуется порядковым номером. Каждый сегмент хранит в заголовке порядковый номер первого октета данных. Данные буферизируются обоими коммуникационными узлами TCP-канала. Как правило, модуль TCP самостоятельно принимает решение, когда именно сформировать сегмент для отправки и когда передать полученные данные процессу-адресату.

В случае, когда требуется немедленная передача данных, без ожидания заполнения сегмента определенного размера, протокол верхнего уровня (приложение) устанавливает флаг PSH, который указывает модулю TCP на необходимость немедленной доставки данных, находящихся в очереди на отправление. Это может потребоваться, например, при передаче пользовательского ввода при удаленном доступе (протокол Telnet).

Как уже говорилось, протокол TCP обеспечивает надежный последовательный виртуальный канал передачи данных между приложениями. По-

Значение MSL, рекомендованное в RFC 793 "Transmission Control Protocol", составляет 2 минуты. Однако в реальных системах типичными значениями MSL являются 30 секунд, 1 или 2 минуты.

Эта информация представлена соответствующими структурами данных, называемыми TCB (Transmission Control Block). Как правило, коммуникационный узел, представляющий сетевой интерфейс для взаимодействующих процессов, хранит указатель на эти управляющие данные. Более подробно архитектура сетевых интерфейсов UNIX описана в следующих разделах.

скольку нижележащий сетевой протокол IP является по определению не-надежным, а среда передачи вносит дополнительные ошибки, переданные данные могут быть утеряны, продублированы или испорчены, при этом порядок их доставки может быть нарушен. В случае ошибочности полученного сегмента модуль TCP узнает об этом, проверив контрольную сумму. Другие ошибки являются более сложными, и TCP должен обеспечить их определение и исправление.

Рассмотренные выше порядковый номер и номер подтверждения играют ключевую роль в обеспечении надежности доставки. По существу порядковый номер адресует каждый октет логического потока данных между источником и получателем, позволяя последнему определить правильность доставки (порядок доставки и потерю отдельных октетов). TCP является протоколом с позитивным подтверждением и повторной передачей (Positive Acknowledgement and Retransmission, PAR). Это означает, что если данные доставлены без ошибок, получатель подтверждает это сегментом ACK. Если отправитель не получает подтверждения в течение некоторого времени, он повторно посыпает данные. В любом случае отсутствует негативное подтверждение (NAK).

В качестве примера рассмотрим передачу данных между двумя хостами сети А и В, проиллюстрированную на рис. 6.13. Для простоты предположим симплексную передачу большого количества данных от хоста А к В. Начиная с  $SEQ=100$  хост А посыпает хосту В 200 октетов. Первый посланный сегмент ( $SEQ=300$ ) доставлен без ошибок и подтвержден хостом В ( $ACK=301$ ). Следующий сегмент передан с ошибкой и не доставлен получателю. Таким образом, хост А не получает подтверждения на второй сегмент и повторно посыпает его после определенного тайм-аута<sup>10</sup>. В конечном итоге все данные, переданные хостом А будут получены и подтверждены хостом В.

Говоря об управлении потоком данных, следует отметить, что TCP представляет собой протокол со скользящим окном. Окно определяет объем данных, который может быть послан (send window — окно передачи) или получен (receive window — окно приема) TCP-модулем. Размеры окон фактически отражают состояние буферов приема коммуникационных узлов. Так окно приема свидетельствует о количестве данных, которое принимающая сторона готова получить, а окно передачи определяет количество данных, которое отправителю позволяет послать, не ожидая подтверждения о получении. Несомненно, между этими двумя параметрами существует связь — окно передачи одного узла отражает состояние буфера

<sup>10</sup> На самом деле ситуация, скорее всего, окажется более печальной, поскольку хост А продолжит отправку последующих сегментов в пределах окна отправки, не дожидаясь подтверждений. Не получив подтверждения на второй сегмент, хост А по таймауту вынужден будет повторно передать все сегменты, начиная со второго. Более подробно мы рассмотрим этот аспект в разделе "Стратегии реализации TCP" далее в этой главе.

ров другого (его окно приема) и наоборот. Принимающая сторона имеет возможность изменять окно передачи отправителя (с помощью подтверждения или явного обновления значения окна в поле Window заголовка передаваемого сегмента), и, таким образом, регулировать трафик.

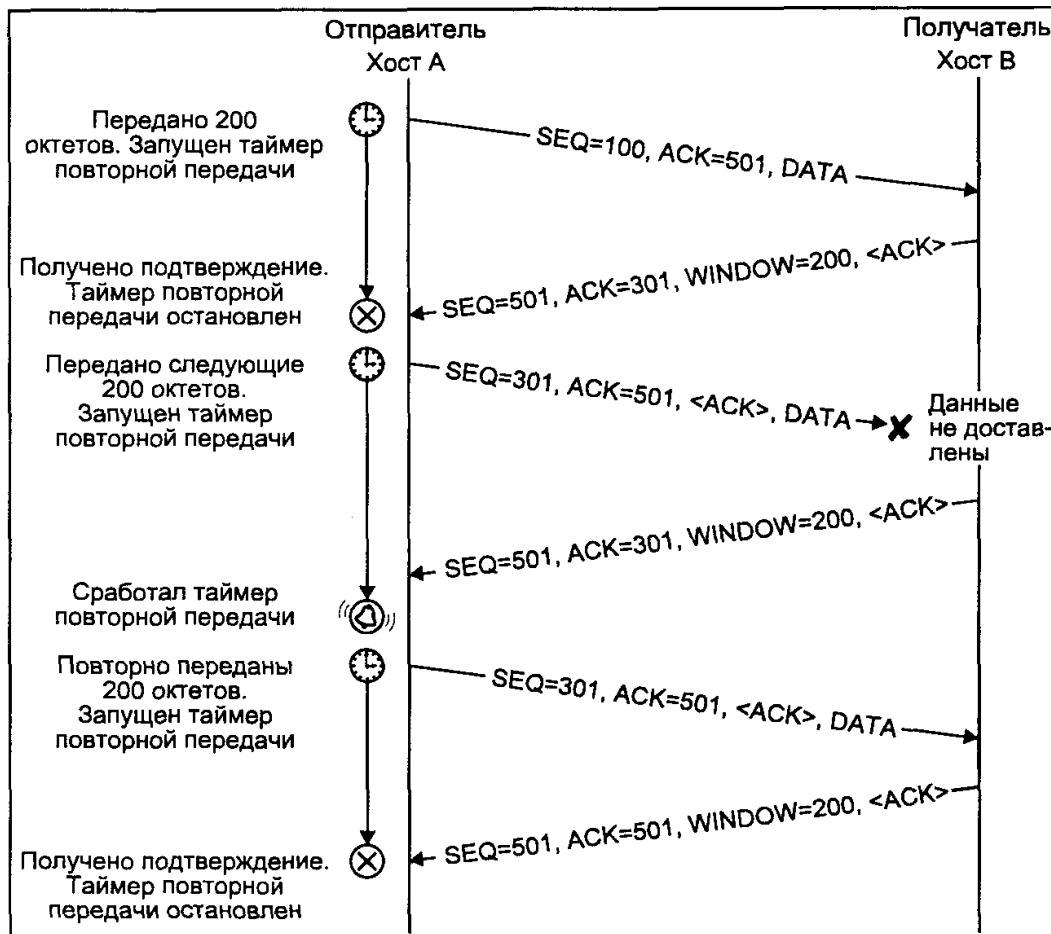


Рис. 6.13. Повторная передача

Интерпретация отправителем окна передачи показана на рис. 6.14. Размер окна передачи отправителя в данном случае покрывает с 4 по 8 байт. Это означает, что отправитель получил подтверждения на все байты, включая 3, а получатель анонсировал размер окна равным 5 байтам. Это также означает, что отправитель может еще передать 2 байта (7 и 8). По мере подтверждения получения данных окно будет смещаться вправо, открывая новые "горизонты" для передачи. Однако окно может изменять свои размеры, при этом имеет значение, смещение какого края окна (правого или левого) приводит к изменению размера.

- Окно закрывается по мере смещения левого края вправо. Это происходит при отправлении данных.
- Окно открывается по мере смещения правого края вправо. Это происходит в соответствии с освобождением буфера приема получателя данных.

- Окно сжимается, когда правый край смещается влево. Хотя такое поведение не рекомендуется, модуль TCP должен быть готов к обработке этой ситуации.

Если левый край окна достигает правого, размер окна становится равным нулю, что запрещает дальнейшую передачу данных.

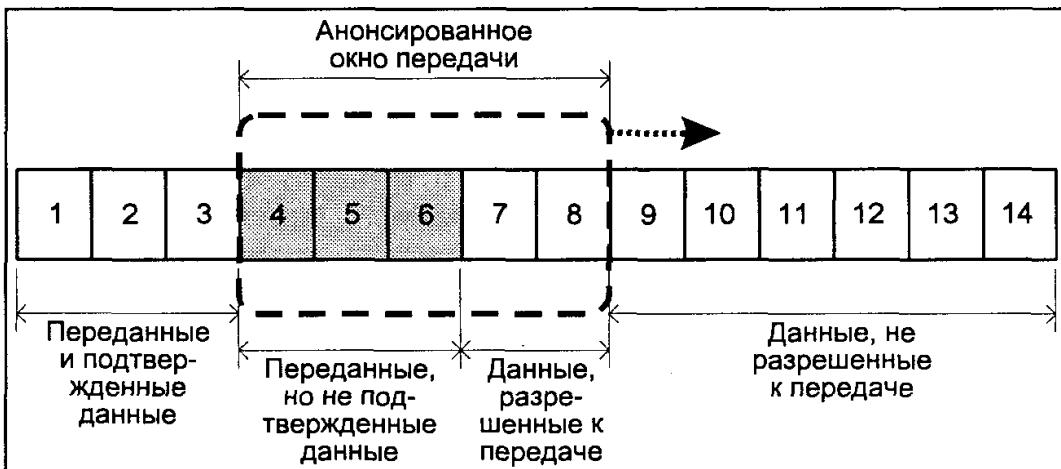


Рис. 6.14. Окно передачи TCP

Суммируя вышесказанное, можно отметить, что размер окна, сообщаемый получателем данных отправителю, является *предлагаемым окном* (offered window), которое в простейшем случае равно размеру свободного места в буфере приема. При получении этого значения отправитель данных вычисляет фактическое, *доступное для использования окно* (usable window), которое равно предлагаемому за вычетом объема отправленных, но не подтвержденных данных. Таким образом, доступное для использования, или просто *доступное*, окно меньше или равно предлагаемому. Неэффективная стратегия подтверждений может привести к чрезвычайно малым значениям доступного окна и, как следствие, к низкой производительности передачи данных. Это явление, известное под названием *синдром "глупого окна"* (Silly Window Syndrome, SWS), будет рассмотрено ниже.

### Стратегии реализации TCP

Рассмотренный стандарт протокола TCP определяет взаимодействие между удаленными объектами, достаточное для обеспечения совместимых реализаций. Другими словами, модуль протокола, в частности следующий спецификации стандарта, является гарантированно совместимым с модулями TCP, разработанными другими производителями. Тем не менее ряд вопросов функционирования протокола остается за рамками стандарта и допускает различные реализации, в конечном итоге влияющие не на совместимость, а на производительность приложений, использующих этот протокол. В данном разделе мы рассмотрим различные подходы к реализации TCP, направленные на повышение его производительности.

### Синдром "глупого окна"

Механизм подтверждения получения данных является ключевым в протоколе TCP. Стандарт указывает, что подтверждение должно быть передано без задержки, но не определяет конкретно, насколько быстро данные должны быть подтверждены, и объем подтверждаемых данных. К сожалению, корректная с точки зрения спецификации протокола, но не оптимальная реализация стратегии подтверждения приводит к неудовлетворительной работе механизма управления потоком данных (оконного механизма), что приводит к синдрому "глупого окна" (SWS).

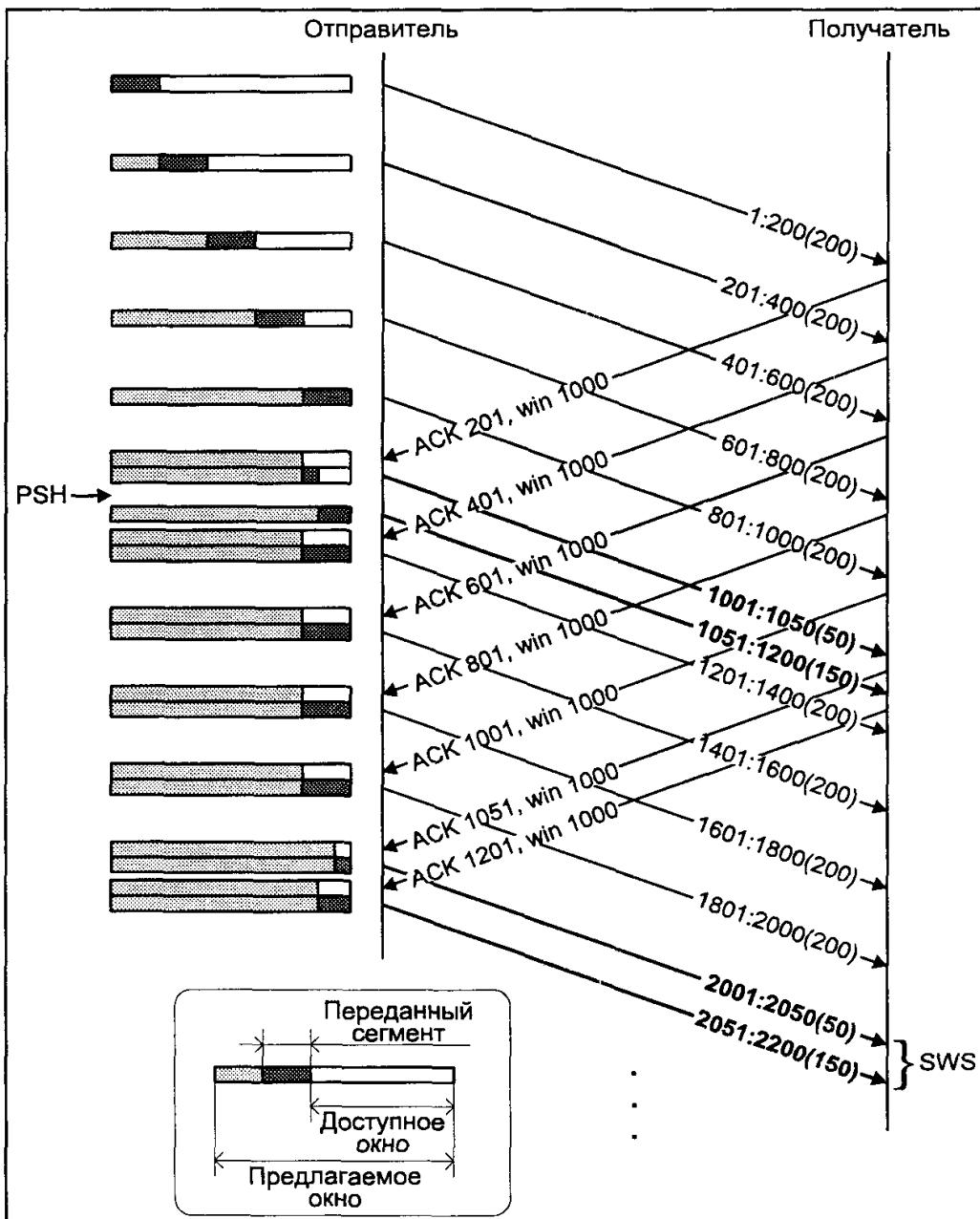
Для иллюстрации этого явления рассмотрим передачу файла большого размера между двумя приложениями, использующими протокол TCP. Допустим, что модуль протокола осуществляет передачу сегментами, размер которых составляет 200 октетов. В начале передачи предлагаемое окно отправителя — 1000 октетов. Он полностью использует этот кредит, послав пять сегментов по 200 октетов каждый. После обработки первого полученного сегмента адресат отправляет подтверждение (сегмент ACK), которое также содержит обновленное значение предлагаемого окна. Предположим, что адресат передал полученные данные приложению, и таким образом его буфер приема вновь содержит 1000 байтов свободного места. Поэтому обновленное значение окна будет также равным 1000 октетов. Эта ситуация показана на рис. 6.15.

При получении подтверждения отправитель вычисляет доступное окно. Поскольку получение 800 октетов данных еще не подтверждено, значение доступного окна получается равным 200.

Рассмотрим теперь процесс возникновения SWS. Предположим, что отправитель вынужден передать сегмент размером 50 октетов (например, если приложение указало флаг PSH). Таким образом, он отправляет 50 байтов, и вслед за этим следующий сегмент, размером 150 октетов (поскольку размер доступного окна равен 200). Через некоторое время адресат получит 50 байтов, обработает их и подтвердит получение, не изменяя значения предлагаемого окна (1000 октетов). Однако теперь при вычислении доступного окна, отправитель обнаружит, что не подтверждены 950 байтов, и, таким образом, его окно равняется всего 50 октетам. В результате отправитель вновь вынужден будет передать всего 50 байтов, хотя приложение этого уже не требует.

Если мы продолжим анализировать передачу данных, то заметим, что рисунок транзакций будет периодически повторяться, т. е. отправитель будет вынужден периодически передавать сегмент необоснованно малого размера. Этот порочный круг не может быть разорван естественным образом. Происхождение сегментов малого размера очевидно: периодически у отправителя возникает необходимость разделить доступное окно на несколько мелких сегментов. При непрерывной передаче больших объемов данных такие ситуации будут время от времени возникать, оставляя неизгла-

димый след на характере транзакций. В результате это может привести к "засорению" сети множеством мелких пакетов в одну сторону и множеством подтверждений в другую.



**Рис. 6.15.** Возникновение SWS

Описанный синдром может также порождаться и принимающей стороной, которая анонсирует чересчур маленькие окна. Таким образом, для преодоления этих ситуаций, необходима модификация алгоритмов TCP как для отправления, так и для приема данных. К счастью, SWS легко избежать, обязав модули выполнять следующие правила:

1. Принимающая сторона не должна анонсировать маленькие окна. Говоря более конкретно, адресат не должен анонсировать размер окна,

больший текущего (который скорее всего равен 0), пока последний не может быть увеличен либо на размер максимального сегмента (Maximum Segment Size, MSS), либо на 1/2 размера буфера приема, в зависимости от того, какое значение окажется меньшим.

2. Отправитель должен воздержаться от передачи, пока он не сможет передать сегмент максимального размера или сегмент, размер которого больше половины максимального размера окна, который когда-либо анонсировался принимающей стороной.

Однако как мы уже заметили, анализируя причины возникновения SWS, поспешные подтверждения полученных данных сыграли не последнюю роль в этом процессе. С одной стороны, немедленное подтверждение позволяет постоянно держать отправителя "в курсе дела", тем самым избегая ненужных повторных передач. Подтверждение также приводит к смещению окна, и таким образом, позволяет отправителю продолжить передачу данных. С другой стороны, немедленное подтверждение может привести к возникновению SWS и дополнительным накладным расходам.

Хорошим компромиссом между немедленным и отложенным подтверждением можно считать следующую схему. При получении сегмента адресат не отправляет подтверждение, если, во-первых, сегмент не содержит флага PSH (дающего основание полагать, что вслед за полученным сегментом вскоре последуют дополнительные данные), и, во-вторых, отсутствует необходимость отправки обновленного значения окна.

Тем не менее получатель должен установить таймер, который позволит послать подтверждение, если в передаче данных произошел определенный перерыв, что может быть вызвано, например, потерей сегментов.

### **Медленный старт**

Старые реализации TCP начинали передачу, отправляя сегменты в пределах предлагаемого окна, не дожидаясь подтверждения. Это вызывало взрывообразный рост трафика в сети и могло привести к переполнению, в результате которого часть сегментов отбрасывалась и требовалась повторная передача.

Алгоритм, направленный на избежание подобной ситуации, получил название *медленного старта* (slow start). Основная идея, лежащая в основе этого алгоритма, заключается в том, что на начальном этапе передачи сегменты должны отправляться со скоростью, пропорциональной скорости получения подтверждений.

Реализация этого алгоритма предусматривает использование дополнительного к рассмотренным ранее окна отправителя — *окна переполнения* (congestion window). При установлении связи с адресатом значение этого окна *cwnd* устанавливается равным одному сегменту (значению MSS, анонсированному адресатом, или некоторому значению по умолчанию,

обычно 536 или 512 байтов). При вычислении доступного окна отправитель использует меньшее из предлагаемого окна и окна переполнения. Каждый раз, когда отправитель получает подтверждение полученного сегмента, его окно переполнения увеличивается на величину этого сегмента.

Легко заметить, что предлагаемое окно служит для управления потоком со стороны получателя, в то время как окно переполнения служит для управления со стороны отправителя. Если первое из них связано с наличием свободного места в буфере приема адресата, то второе — с представлением о загрузке сети у отправителя данных.

Обычно предлагаемое окно больше одного сегмента, поэтому отправитель передает один сегмент и ожидает подтверждения. Когда подтверждение приходит, он увеличивает значение окна переполнения до двух сегментов, таким образом, два сегмента разрешены к передаче. После того как получение каждого из этих сегментов подтверждено, размер окна переполнения становится равным четырем сегментам. Можно показать, что по мере отправления сегментов и получения подтверждений размер окна переполнения растет экспоненциально, соответственно растет и эффективная скорость передачи<sup>11</sup>.

Начиная с некоторого значения скорость передачи достигнет эффективной пропускной способности виртуального канала между источником и получателем, и ее дальнейший рост приведет к потере данных. Начиная с этого момента, включается механизм устранения заторов, который будет обсужден ниже.

## Устранение затора

Переполнение, или затор, может возникнуть в сети по многим причинам. Например, если данные поступают к шлюзу по высокоскоростному каналу и должны быть переданы в низкоскоростной канал. Или данные нескольких каналов мультиплексируются в один канал, пропускная способность которого меньше суммы входящих. Во всех этих случаях неизбежна потеря пакетов.

Алгоритмы, позволяющие избежать заторов, основываются на предположении, что потеря данных, вызванная ошибками передачи по физической среде, пренебрежимо мала (гораздо меньше 1%). Следовательно, потеря

<sup>11</sup> Легко вывести формулу изменения размера окна, предполагая, что время передачи данных от отправителя к получателю и обратно (Round Trip Time, RTT) гораздо больше времени передачи сегмента отправителем. Здесь параметр sz равен размеру сегмента (например, MSS):

$$\begin{aligned} \text{cwnd}_0 &= \text{sz} \\ \text{cwnd}_1 &= \text{cwnd}_0 + (\text{cwnd}_0/\text{sz}) * \text{sz} = 2 * \text{cwnd}_0 \end{aligned}$$

$$\text{cwnd}_n = 2 * \text{cwnd}_{n-1} = 2^n * \text{sz}$$

данных свидетельствует о заторе, произошедшем где-то на пути следования пакета. В свою очередь, о потере данных отправитель может судить по двум событиям: значительной паузе в получении подтверждения или получении дубликата(ов) подтверждения.

Хотя устранение затора и медленный старт являются независимыми механизмами, каждый из которых имеет свою цель, обычно они реализуются совместно. Для их работы необходимо два дополнительных параметра виртуального канала<sup>12</sup>: окно переполнения *cwnd* и порог медленного старта *ssthresh*. Работа комбинированного алгоритма определяется следующим правилам:

1. Начальные значения *cwnd* и *ssthresh* инициализируются равными размеру одного сегмента и 65535 байтов соответственно.
2. Максимальное количество данных, которое может передать отправитель, не превышает меньшего из значений окна переполнения и предлагаемого окна.
3. При возникновении затора (что определяется по тайм-ауту или получению дубликатов подтверждений) параметр *ssthresh* устанавливается равным половине текущего окна, но не меньше размера двух сегментов. Если же свидетельством затора является тайм-аут, то дополнительно размер *cwnd* устанавливается равным одному сегменту, или, другими словами, включается медленный старт.
4. Когда отправитель получает подтверждение, он увеличивает размер *cwnd*, однако новый размер зависит от того, выполняет ли модуль медленный старт или устранение затора.

Если значение *cwnd* меньше или равно *ssthresh*, то TCP находится в фазе медленного старта, в противном случае производится устранение затора. Таким образом, режим медленного старта продолжается до тех пор, пока эффективная скорость передачи не достигнет половины скорости, при которой был обнаружен затор<sup>13</sup>. После этого включается процедура устранения затора.

Как мы только что видели, медленный старт начинается с отправления одного сегмента, затем двух, затем четырех и т. д., что порождает экспоненциальный рост размера окна. В фазе устранения затора вычисление

<sup>12</sup> Для простоты мы рассматриваем несимметричный виртуальный канал, в котором данные передаются в одну сторону, а управляющие сообщения (подтверждения, обновления окна и т. д.) передаются в обратную сторону. Эти рассуждения легко могут быть распространены и на случай полнодуплексного канала, когда каждая из сторон одновременно является и получателем и отправителем данных.

<sup>13</sup> Поскольку скорость передачи определяется текущим окном, половина размера окна, сохраненная в *ssthresh*, определяет 1/2 скорости, при которой произошел затор.

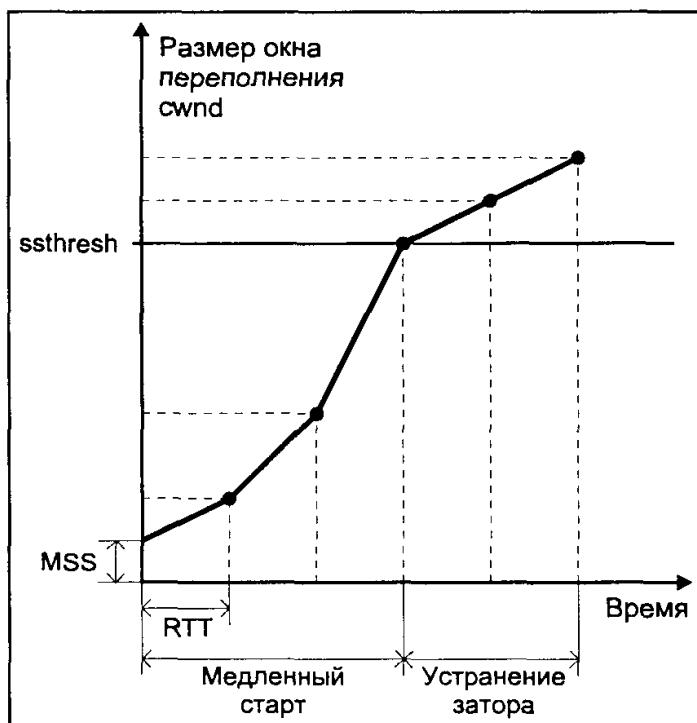
нового значения  $cwnd$  производится по следующей формуле при каждом подтверждении сегмента<sup>14</sup>:

$$cwnd_{n+1} = cwnd_n + 1/cwnd_n$$

Таким образом, формула дает зависимость роста размера окна, при которой максимальная скорость приращения составит не более одного сегмента за время передачи данных туда и обратно (Round Trip Time, RTT), независимо от того, сколько подтверждений было получено. Это утверждение легко доказать. Допустим, в какой-то момент времени размер окна составлял  $cwnd_n$ . Тогда отправитель может передать максимум  $cwnd_n/sz$  сегментов размером  $sz$ , на которые он получит такое же число подтверждений. Можно показать, что

$$cwnd_{n+1} < cwnd_n + (cwnd_n/sz) \times (1/cwnd_n) = cwnd_n + sz$$

На рис. 6.16 показан рост окна переполнения при медленном старте и последующем устранении затора. Заметим, что переход в фазу устранения затора происходит при превышении размером окна порогового значения  $ssthresh$ .



**Рис. 6.16.** Рост окна переполнения при медленном старте и устранении затора

### Повторная передача

До сих пор рассматривалось получение дублированных подтверждений как свидетельство потери сегментов и затора в сети. Однако согласно RFC 1122 "Requirements for Internet Hosts — Communication Layers", модуль TCP

<sup>14</sup> Предполагается, что получатель подтверждает каждый сегмент. На самом деле это не так, и в этом случае приращение производится фактически на число подтвержденных сегментов.

может отправить немедленное подтверждение при получении неупорядоченных сегментов. Цель такого подтверждения — уведомить отправителя, что был получен неупорядоченный сегмент, и указать порядковый номер ожидаемых данных. Поскольку ожидаемый порядковый номер остался прежним (получение неупорядоченного сегмента не изменит его значение), данное подтверждение может явиться дубликатом уже отправленного ранее.

Таким образом, получение дублированных подтверждений может быть вызвано двумя причинами: потерей сегмента, как следствием затора в сети, и получением неупорядоченного сегмента. Чтобы установить истинную причину, модуль TCP ждет получения еще нескольких дублированных подтверждений. Если причина в получении неупорядоченного сегмента, вызванном буферизацией на промежуточных шлюзах или различными путями передачи датаграмм, то, вероятнее всего, вскоре ожидаемый сегмент будет получен и порядок будет восстановлен, что выразится в получении нового (уже не дубликата) подтверждения. Если получено три или более дубликатов, следует полагать, что произошла потеря данных. В этом случае отправитель совершает повторную передачу утраченного сегмента. Эта процедура получила название *быстрой повторной передачи* (fast retransmit). При этом, включается механизм устранения затора, но не медленный старт. Причиной такого поведения является то, что получение сегмента, хотя и не упорядоченного, свидетельствует об относительно невысоком уровне переполнения в сети, и необходимость в столь радикальных мерах, как медленный старт, отсутствует.

Однако потеря данных может вызвать ответное молчание. Для обработки подобной ситуации отправитель должен установить таймер и повторно передать данные по тайм-ауту, начиная с последнего подтверждения. Данный механизм является запасным и гарантирует повторную передачу, хотя и вызывает довольно большие задержки.

## **Программные интерфейсы**

### **Программный интерфейс сокетов**

Вы уже познакомились с интерфейсом сокетов при обсуждении реализации межпроцессного взаимодействия в BSD UNIX. Поскольку сетевая поддержка впервые была разработана именно для BSD UNIX, интерфейс сокетов и сегодня является весьма распространенным при создании сетевых приложений. В разделе "Поддержка сети в BSD UNIX" мы вновь вернемся к сокетам, когда будем рассматривать внутреннюю архитектуру сетевой подсистемы в UNIX ветви BSD. Сейчас же рассмотрим простой пример приложения клиент-сервер, который демонстрирует возможности сокетов при обеспечении взаимодействия между удаленными процессами. Несмотря на то что взаимодействие затрагивает передачу данных по сети,

приведенная программа мало отличается от примера, рассмотренного в разделе "Межпроцессное взаимодействие в BSD UNIX. Сокеты" главы 3. Логика приложения сохранена — клиент отправляет серверу сообщение, сервер передает его обратно, а клиент, в свою очередь, выводит полученное сообщение на экран. Наиболее существенным отличием является коммуникационный домен сокетов — в данном случае AF\_INET. Соответственно изменилась и схема адресации коммуникационного узла. Согласно схеме адресации TCP/IP, коммуникационный узел однозначно идентифицируется двумя значениями: адресом хоста (IP-адрес) и адресом процесса (адрес порта). Это отражает и структура sockaddr\_in, которая является конкретным видом общей структуры адреса сокета sockaddr. Структура sockaddr\_in имеет следующий вид:

```
struct sockaddr_in {  
    short           sin_family;   Коммуникационный домен — AF_INET  
    u_short         sin_port;     Номер порта  
    struct in_addr  sin_addr;    IP-адрес хоста  
    char            sin_zero[8];  
};
```

Адрес порта должен быть предварительно оговорен между клиентом и сервером.

В заключение, прежде чем перейти непосредственно к текстам программы, заметим, что интерфейс сокетов также поддерживается и в UNIX System V, наряду с другим программным интерфейсом — ТЫ, который будет рассмотрен в следующем разделе.

Приведенный пример в качестве транспортного протокола использует TCP. Это значит, что перед передачей прикладных данных клиент должен установить соединение с сервером. Эта схема, приведенная на рис. 6.17, несколько отличается от рассмотренной в разделе "Межпроцессное взаимодействие в BSD UNIX. Сокеты", где передача данных осуществлялась без предварительного установления связи и в данном случае соответствовала бы использованию протокола UDP.

В соответствии с этой схемой сервер производит связывание с портом, номер которого предполагается известным для клиентов (*bind(2)*), и сообщает о готовности приема запросов (*listen(2)*). При получении запроса он с помощью функции *accept(2)* создает новый сокет, который и обслуживает обмен данными между клиентом и сервером. Для того чтобы сервер мог продолжать обрабатывать поступающие запросы, он порождает отдельный процесс на каждый поступивший запрос. Дочерний процесс, в свою очередь, принимает сообщения от клиента (*recv(2)*) и передает их обратно (*send(2)*).

Клиент не выполняет связывания, поскольку ему безразлично, какой адрес будет иметь его коммуникационный узел. Эту операцию выполняет систе-

ма, выбирая свободный адрес порта и установленный адрес хоста. Далее клиент направляет запрос на установление соединения (*connect(2)*), указывая адрес сервера (IP-адрес и номер порта). После установления соединения ("тройное рукопожатие") клиент передает сообщение (*send(2)*), принимает от сервера ответ (*recv(2)*) и выводит его на экран.

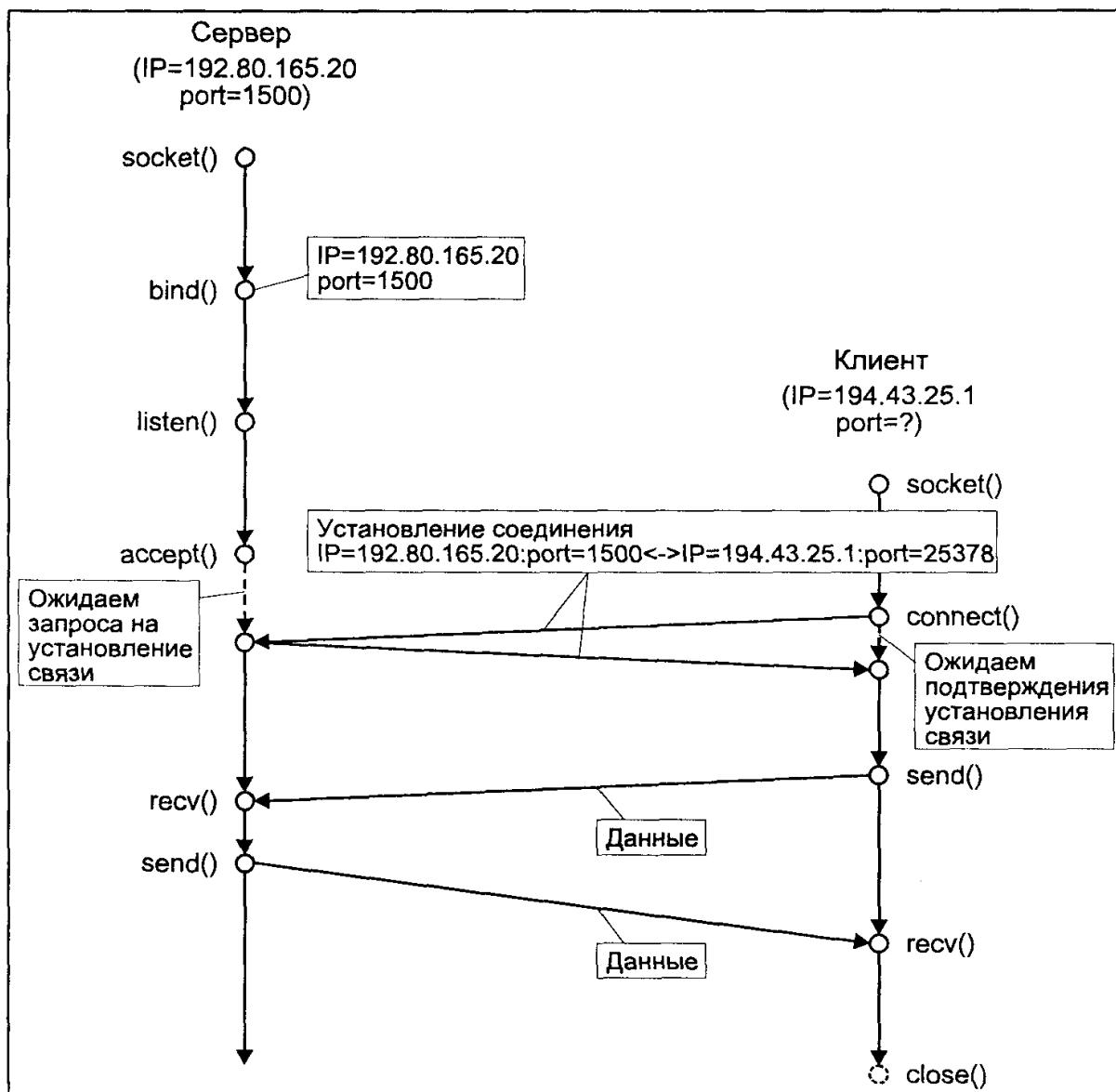


Рис. 6.17. Схема установления связи и передачи данных между клиентом и сервером

В программе используются несколько функций, которые не рассматривались. Эти функции значительно облегчают жизнь программисту, выполняя, например, такие действия, как трансляцию доменного имени хоста в его IP-адрес (*gethostbyname(3N)*), приведение в соответствие порядка следования байтов в структурах данных, который может различаться для хоста и сети (*htons(3N)*), а также преобразование IP-адресов и их составных частей в соответствии с привычной "человеческой" нотацией, например 127.0.0.1 (*inet\_ntoa(3N)*). Мы не будем подробнее останавливаться на этих функци-

ях, предоставляя читателю самостоятельно обратиться к соответствующим разделам электронного справочника *man(1)*.

Ниже приведены тексты программ сервера и клиента.

## Сервер

```
#include<sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include<stdio.h>
#include <fcntl.h>
#include <netdb.h>

/*Номер порта сервера, известный клиентам*/
#define PORTNUM 1500

main(argc, argv)
int argc;
char *argv[];
{
    int s, ns;
    int pid;
    int nport;
    struct sockaddr_in serv_addr, clnt_addr;
    struct hostent *hp;
    char buf[80], hname[80];
/*Преобразуем порядок следования байтов к сетевому формату*/
    nport = PORTNUM;
    nport = htons((u_short)nport );

/*Создадим сокет, использующий протокол TCP*/
    if((s=socket(AF_INET, SOCK_STREAM, 0))==-1)
    {
        perror("Ошибка вызова socket()"); exit(1);
    }

/*Зададим адрес коммуникационного узла*/
    bzero(&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = nport;

/*Свяжем сокет с этим адресом*/
    if(bind(s, (struct sockaddr *)&serv_addr,
            sizeof(serv_addr))==-1)
    {
        perror("Ошибка вызова bind()"); exit(1);
    }

/*Выведем сообщение с указанием адреса сервера*/
    fprintf(stderr, "Сервер готов: %s\n",
            inet_ntoa(serv_addr.sin_addr));
/*Сервер готов принимать запросы на установление соединения.
Максимальное число запросов, ожидающих обработки -5. Как правило,
```

```

этого числа достаточно, чтобы успеть выполнить accept(2) и
породить дочерний процесс*/
    if(listen(s,5)==-1)
    {
        perror("Ошибка вызова listen()"); exit(1);
    }

/*Бесконечный цикл получения запросов и их обработки*/
    while(1)
    {
        int addrlen;

        bzero(&clnt_addr, sizeof(clnt_addr));
        addrlen = sizeof(clnt_addr);

/*Примем запрос. Новый сокет ns становится коммуникационным узлом
созданного виртуального канала*/
        if((ns=accept(s, (struct sockaddr *)&clnt_addr,
                      &addrlen))==-1)
        {
            perror("Ошибка вызова accept ()"); exit(1);
        }

/*Выведем информацию о клиенте*/
        fprintf(stderr, "Клиент = %s\n",
                inet_ntoa(clnt_addr.sin_addr));
/*Создадим процесс для работы с клиентом*/
        if((pid=fork())==-1)
        {
            perror("Ошибка вызова fork()"); exit(1);
        }

        if(pid==0)
        {
            int nbytes;
            int fout;

/*Дочерний процесс: этот сокет нам не нужен. Он по-прежнему
используется для получения запросов*/
            close(s);
/*Получим сообщение от клиента и передадим его обратно*/
            while ((nbytes = recv(ns, buf, sizeof(buf), 0))!=0)
            {
                send(ns, buf, sizeof(buf), 0);
            }
            close(ns);
            exit(0);
        }

/*Родительский процесс: этот сокет нам не нужен. Он используется
дочерним процессом для обмена данными*/
        close(ns);
    }
}

```

**Клиент**

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <fcntl.h>
#include <netdb.h>

/*Номер порта, который обслуживается сервером*/
#define PORTNUM 1500

main(argc, argv)
char *argv[];
int argc;
{
    int s;
    int pid;
    int i,j;
    struct sockaddr_in serv_addr;
    struct hostent *hp;
    char buf[80]={"Hello, World!"};

    /*В качестве аргумента клиенту передается доменное имя хоста, на
    котором запущен сервер. Произведем трансляцию доменного имени в
    адрес*/
    if((hp=gethostbyname(argv[1]))==0)
    {
        perror("Ошибка вызова gethostbyname()"); exit(3);
    }
    bzero(&serv_addr, sizeof(serv_addr));
    bcopy(hp->h_addr, &serv_addr.sin_addr, hp->h_length);
    serv_addr.sin_family = hp->h_addrtype;
    serv_addr.sin_port = htons(PORTNUM);

    /*Создадим сокет*/
    if((s=socket(AF_INET, SOCK_STREAM, 0))==-1)
    {
        perror("Ошибка вызова socket()"); exit(1);
    }

    fprintf(stderr, "Адрес клиента: %s\n",
            inet_ntoa(serv_addr.sin_addr));

    /*Создадим виртуальный канал*/
    if(connect(s, (struct sockaddr *)&serv_addr,
               sizeof(serv_addr))==-1)
    {
        perror("Ошибка вызова connect()"); exit(1);
    }

    /*Отправим серверу сообщение и получим его обратно*/
    send(s, buf, sizeof(buf), 0);
```

```

if (recv(s, buf, sizeof(buf), 0) < 0)
{
    perror("Ошибка вызова recv()"); exit(1);
}

/* Выведем полученное сообщение на экран */
printf("Получено от сервера: %s\n", buf);
close (s);
printf("Клиент завершил работу \n\n");
}

```

## Программный интерфейс TLI

При обсуждении реализации сетевой поддержки в BSD UNIX был рассмотрен программный интерфейс доступа к сетевым ресурсам, основанный на сокетах. В данном разделе описан *интерфейс транспортного уровня* (Transport Layer Interface, TLI), который обеспечивает взаимодействие прикладных программ с транспортными протоколами.

ТЫ был впервые представлен в UNIX System V Release 3.0 в 1986 году. Этот программный интерфейс тесно связан с сетевой подсистемой UNIX, основанной на архитектуре STREAMS, изолируя от прикладной программы особенности сетевой архитектуры. Вместо того чтобы непосредственно пользоваться общими функциями STREAMS, рассмотренными в предыдущей главе, ТЫ позволяет использовать специальный набор вызовов, специально предназначенных для сетевых приложений. Для преобразования вызовов ТЫ в функции интерфейса STREAMS используется библиотека ТЫ, которая в большинстве систем UNIX имеет название **libnsl.a** или **libnsl.so**.

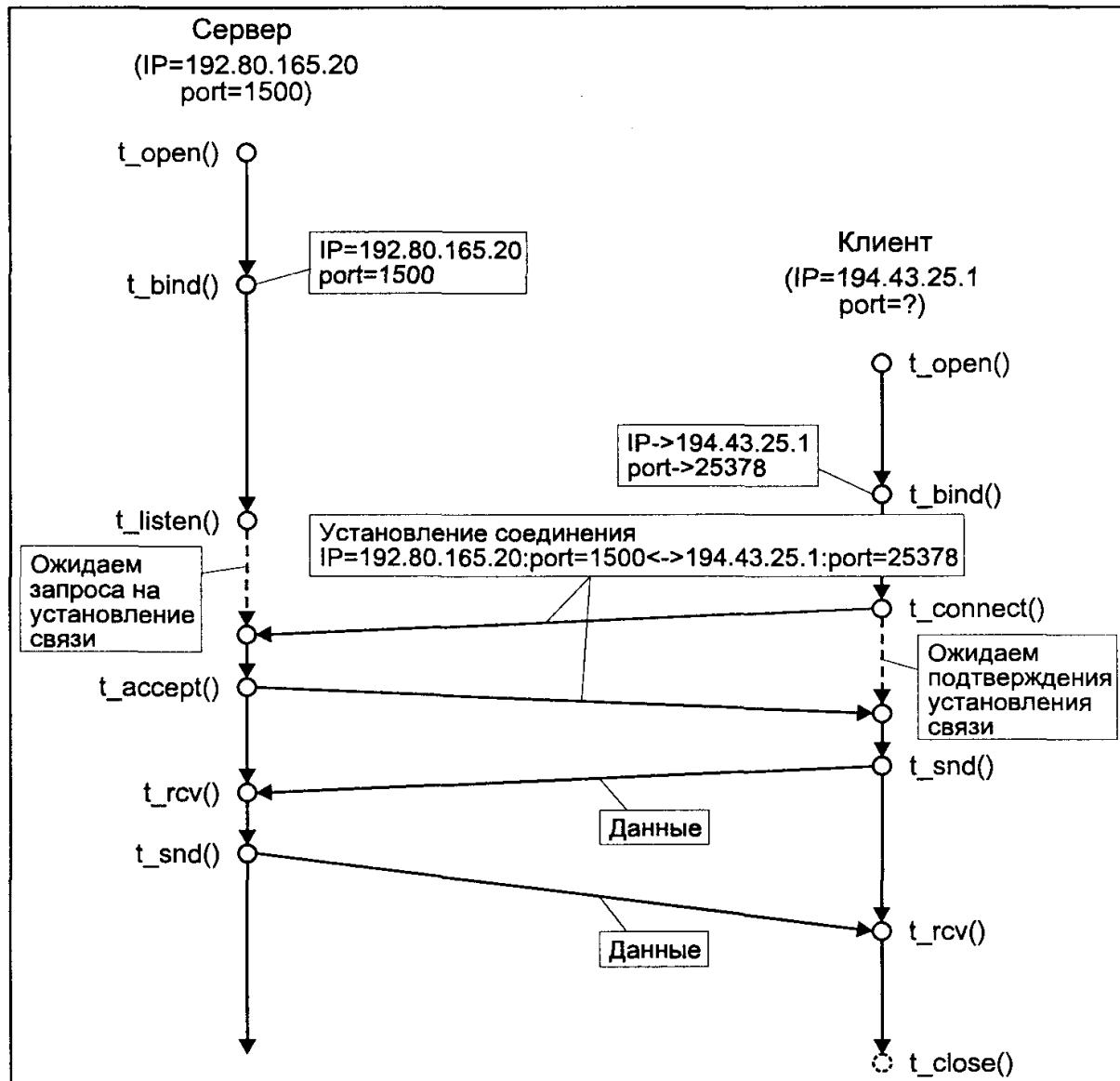
Схема использования функций ТЫ во многом сходна с рассмотренным интерфейсом сокетов и зависит от типа используемого протокола — с предварительным установлением соединения (например, TCP) или без него (например, UDP).

На рис. 6.18 и 6.19 представлены схемы использования функций ТЫ для транспортных протоколов с предварительным установлением соединения и без установления соединения. Можно отметить, что эти схемы очень похожи на те, с которыми мы уже встречались в разделе "Межпроцессное взаимодействие в BSD UNIX. Сокеты" главы 3 при обсуждении сокетов. Некоторые различия отмечены ниже при описании функций ТЫ.

Прежде чем перейти к обсуждению функций ТЫ, остановимся на определении адреса коммуникационного узла. ТЫ не накладывает никаких ограничений на формат адреса, возлагая его интерпретацию на протоколы нижнего уровня. Благодаря этому, один и тот же интерфейс может быть использован при работе с различными семействами сетевых протоколов.

Для определения адреса ТЫ предоставляет общую структуру данных `netbuf`, имеющую вид:

```
struct netbuf {
    unsigned int maxlen;
    unsigned int len;
    char *buf;
}
```



**Рис. 6.18.** Схема вызова функций TLI для протокола с предварительным установлением соединения

Поле `buf` указывает на буфер, в котором может передаваться адрес узла, `maxlen` определяет его размер, а `len` — количество данных в буфере, т. е. размер адреса. Эта структура по своему назначению похожа на структуру `sockaddr`, которая является общим определением адреса коммуникационного узла для сокетов. Далее рассматривается пример сетевого приложе-

ния, основанного на ТЫ, где показано, как netbuf используется при передаче адреса для протоколов TCP/IP.

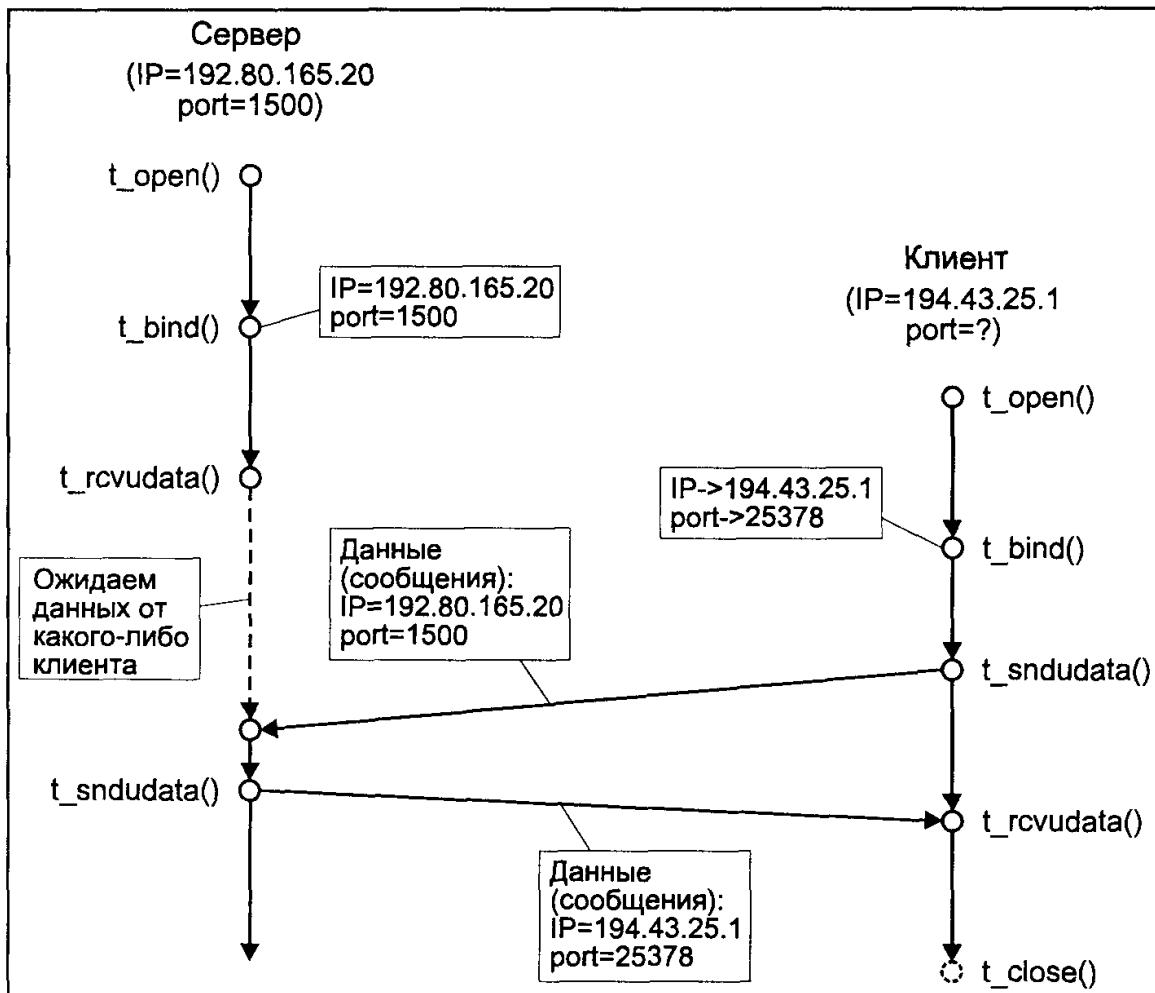


Рис. 6.19. Схема вызова функций TLI для протокола без предварительного установления соединения

Структура netbuf используется в ТЫ для хранения не только адреса, но и другой информации — опций протокола и прикладных данных. Эта структура является составной частью более сложных структур данных, используемых при передаче параметров в функциях ТЫ. Для упрощения динамического размещения этих структур библиотека ТЫ предоставляет две функции: *t\_alloc(3N)* для размещения структуры и *t\_free(3N)* для освобождения памяти. Эти функции имеют следующий вид:

```
#include <tiuser.h>

char *t_alloc(int fd, int struct_type, int fields);
int t_free(char *ptr, int struct_type);
```

Аргумент *struct\_type* определяет, для какой структуры данных выделяется память. Он может принимать следующие значения:

Значение поля <code>struct_type</code>	Структура данных
<code>T_BIND</code>	<code>struct t_bind</code>
<code>T_CALL</code>	<code>struct t_call</code>
<code>T_DIS</code>	<code>struct t_discon</code>
<code>T_INFO</code>	<code>struct t_info</code>
<code>T_OPTMGMT</code>	<code>struct t_optmgmt</code>
<code>T_UNITDATA</code>	<code>struct t_unitdata</code>
<code>T_UDERR</code>	<code>struct t_uderr</code>

Со структурами, приведенными в таблице, мы познакомимся при обсуждении функций ТЫ. Большинство из них включают несколько элементов `netbuf`. Поскольку в некоторых случаях может отсутствовать необходимость размещения всех элементов `netbuf`, поле `fields` позволяет указать, какие конкретно буферы необходимо разместить для данной структуры:

Значение поля <code>fields</code>	Размещаемые и инициализируемые поля
<code>T_ALL</code>	Все необходимые поля
<code>T_ADDR</code>	Поле <code>addr</code> в структурах <code>t_bind</code> , <code>t_call</code> , <code>t_unitdata</code> , <code>t_uderr</code>
<code>T_OPT</code>	Поле <code>opt</code> в структурах <code>t_call</code> , <code>t_unitdata</code> , <code>t_uderr</code> , <code>t_optmgmt</code>
<code>T_UDATA</code>	Поле <code>udata</code> в структурах <code>t_call</code> , <code>t_unitdata</code> , <code>t_discon</code>

Отметим одну особенность. Фактический размер буфера и, соответственно, структуры `netbuf` зависят от значения поля `maxlen` этой структуры. В свою очередь, этот параметр зависит от конкретного поставщика транспортных услуг — именно он определяет максимальный размер адреса, опций и прикладных данных. Чуть позже мы увидим, что эта информация ассоциирована с транспортным узлом и может быть получена после его создания с помощью функции `t_open(3N)`. Поэтому для определения фактического размера размещаемых структур в функции `t_alloc(3N)` необходим аргумент `fd`, являющийся дескриптором транспортного узла, который возвращается процессу функцией `t_open(3N)`.

Перейдем к основным функциям ТЫ.

Как видно из рис. 6.18 и 6.19, в качестве первого этапа создания коммуникационного узла используется функция `t_open(3N)`. Как и системный вызов `open(2)`, она возвращает дескриптор, который в дальнейшем адресует узел в функциях ТЫ. Функция имеет вид:

```
#include <tiuser.h>
#include <fcntl.h>
int t_open(const char *path, int oflags, struct t_info *info);
```

Аргумент `path` является именем специального файла устройства, являющегося поставщиком транспортных услуг, например, `/dev/tcp` или `/dev/udp`. Аргумент `oflags` определяет флаги открытия файла и соответствует аналогичному аргументу системного вызова `open(2)`. Приложение может получить информацию о поставщике транспортных услуг в структуре `info`, имеющей следующие поля:

<code>addr</code>	Определяет максимальный размер адреса транспортного протокола. Значение -1 говорит, что размер не ограничен, -2 означает, что прикладная программа не имеет доступа к адресам протокола. Протокол TCP устанавливает размер этого адреса (адрес порта) равным 16.
<code>options</code>	Определяет размер опций для данного протокола. Значение -1 свидетельствует, что размер не ограничен, -2 означает, что прикладная программа не имеет возможности устанавливать опции протокола.
<code>tsdu</code>	Определяет максимальный размер пакета данных протокола (Transport Service Data Unit, TSDU). Нулевое значение означает, что протокол не поддерживает пакетную передачу (т. е. не сохраняет границы записей). Значение -1 свидетельствует, что размер не ограничен, -2 означает, что передача обычных данных не поддерживается. Поскольку протокол TCP обеспечивает передачу неструктурированного потока данных, значение <code>tsdu</code> для него равно 0. Напротив, UDP поддерживает пакетную передачу.
<code>etsdu</code>	Определяет максимальный размер пакета экстренных данных протокола (Expedited Transport Service Data Unit, ETSDU). Нулевое значение означает, что протокол не поддерживает пакетную передачу (т. е. не сохраняет границы записей). Значение -1 свидетельствует, что размер не ограничен, -2 означает, что передача экстренных данных не поддерживается. TCP обеспечивает такую поддержку, а UDP – нет.
<code>connect</code>	Некоторые протоколы допускают передачу прикладных данных вместе с запросом на соединение. Поле <code>connect</code> определяет максимальный размер таких данных. Значение -1 свидетельствует, что размер не ограничен, -2 означает, что данная возможность не поддерживается. И TCP и UDP не поддерживают этой возможности.
<code>discon</code>	Определяет то же, что и <code>connect</code> , но при запросе на прекращение соединения. И TCP и UDP не поддерживают этой возможности.
<code>servtype</code>	Определяет тип транспортных услуг, предоставляемых протоколом. Значение <code>T_COTS</code> означает передачу с предварительным установлением соединения, <code>T_COTS_ORD</code> – упорядоченную передачу с предварительным установлением соединения, <code>T_CLTS</code> – передачу без предварительного установления соединения. Протокол TCP обеспечивает услугу <code>T_COTS_ORD</code> , а UDP – <code>T_CLTS</code> .

Прежде чем передача данных будет возможна, транспортному узлу должен быть присвоен адрес. Эта фаза называется *операцией связывания* и мы уже сталкивались с ней при разговоре о сокетах в главе 3 и при обсуждении сетевой поддержки в BSD UNIX ранее в этой главе. В рассмотренных случаях связывание выполнял вызов *bind(2)*. В ТЫ для этого служит функция *t\_bind(3N)*, имеющая вид:

```
#include <tiuser.h>
int t_bind(int fd, const struct t_bind *req,
           struct t_bind *ret);
```

Аргумент *fd* адресует коммуникационный узел. Аргумент *req* позволяет программе явно указать требуемый адрес, а через аргумент *ret* возвращается значение, установленное протоколом.

Два последних аргумента описываются структурой *t\_bind*, имеющей следующие поля:

struct netbuf addr	<b>Адрес</b>
unsigned qlen	Максимальное число запросов на установление связи, которые могут ожидать обработки. Имеет смысл только для протоколов с предварительным установлением соединения

Рассмотрим три возможных формата аргумента *req*:

<i>req</i> == NULL	Позволяет поставщику транспортных услуг самому выбрать подходящий адрес
<i>req</i> != NULL <i>req-&gt;addr.len</i> == 0	Позволяет поставщику транспортных услуг самому выбрать подходящий адрес, но определяет максимальное число запросов на установление связи, которые могут ожидать обработки
<i>req</i> != NULL <i>req-&gt;addr.len</i> > 0	Явно указывает требуемый адрес и максимальное число запросов на установление связи, которые могут ожидать обработки

Во всех случаях фактическое значение адреса возвращается в структуре *ret*. Даже если программа явно указала требуемый адрес, необходимо проверить, совпадает ли он с адресом, указанным в *ret*.

Для протоколов с предварительным установлением соединения программе-клиенту необходимо использовать функцию *t\_connect(3N)*, отправляющую запрос на создание соединения с удаленным транспортным узлом. Функция *t\_connect(3N)* имеет вид:

```
#include <tiuser.h>
int t_connect(int fd, const struct t_call *sndcall,
              struct t_call *rcvcall);
```

Аргумент *sndcall* содержит информацию, необходимую поставщику транспортных услуг для создания виртуального канала. Формат этой информации описывается структурой *t\_call*, имеющей следующие поля:

struct netbuf addr	Адрес удаленного транспортного узла
struct netbuf opt	Требуемые опции протокола
struct netbuf udata	Прикладные данные, отправляемые вместе с управляющей информацией (запрос на установление соединения или подтверждение)
int sequence	в данном случае не имеет смысла

Через аргумент `rcvcall` программе возвращается информация о виртуальном канале после его создания: адрес удаленного узла, опции и прикладные данные, переданные удаленным узлом. Как уже отмечалось, ни TCP, ни UDP не позволяют передавать данные вместе с управляющей информацией. Программа может установить значение `rcvcall` равным NULL, если информация о канале ее не интересует.

Обычно возврат из функции `t_connect(3N)` происходит после окончательного установления соединения, когда виртуальный канал готов к передаче данных (конечно, в случае успешного завершения).

Для протоколов с предварительным установлением соединения программа-сервер вызывает функцию `t_listen(3N)`, блокируя свое выполнение до получения запроса на создание виртуального канала.

```
#include <tiuser.h>
int t_listen(int fd, struct t_call *call);
```

Информация, возвращаемая транспортным протоколом в аргументе `call`, содержит параметры, переданные удаленным узлом с помощью соответствующего вызова `t_connect(3N)`: его адрес, установленные опции протокола, а также, в ряде случаев, прикладные данные, переданные вместе с запросом. Поле `sequence` аргумента `call` содержит уникальный идентификатор данного запроса.

Хотя `t_listen(3N)`, несмотря на название, напоминает функцию `accept(2)`, используемую для сокетов, сервер должен выполнить вызов другой функции — `t_accept(3N)` для того, чтобы фактически принять запрос и установить соединение. Функция `t_accept(3N)` имеет вид:

```
#include <tiuser.h>
int t_accept(int fd, int connfd, struct t_call *call);
```

Аргумент `fd` адресует транспортный узел, принявший запрос (тот же, что и для функции `t_listen(3N)`). Аргумент `connfd` адресует транспортный узел, для которого будет установлено соединение с удаленным узлом. За создание нового транспортного узла отвечает сама программа (т. е. необходим явный вызов функции `t_open(3N)`), при этом `fd` может по-прежнему использоваться для обслуживания поступающих запросов.

Как и в случае `t_listen(3N)`, через аргумент `call` передается информация об удаленном транспортном узле.

После возврата из функции *t\_accept(3N)* между двумя узлами (connfd и удаленным узлом-клиентом) образован виртуальный канал, готовый к передаче прикладных данных.

Для обмена прикладными данными после установления соединения используются две функции: *t\_rcv(3N)* для получения и *t\_snd(3N)* для передачи. Они имеют следующий вид:

```
ttinclude<tiuser.h>
int t_rcv(int fildes, char *buf, unsigned nbytes, int *flags);
int t_snd(int fildes, char *buf, unsigned nbytes, int flags);
```

Первые три аргумента соответствуют аналогичным аргументам системных вызовов *read(2)* и *write(2)*. Аргумент *flags* функции *t\_snd(3N)* может содержать следующие флаги:

<b>T_EXPEDITED</b>	Указывает на отправление экстренных данных
<b>T_MORE</b>	Указывает, что данные составляют логическую запись, продолжение которой будет передано последующими вызовами <i>t_snd(3N)</i> . Напомним, что TCP обеспечивает неструктурированный поток и, следовательно, не поддерживает данной возможности

Эту информацию принимающий узел получает с помощью *t\_rcv(3N)* также через аргумент *flags*.

Для протоколов без предварительного установления соединения используются функции *t\_rcvudata(3N)* и *t\_sndudata(3N)* для получения и передачи датаграмм соответственно. Функции имеют следующий вид:

```
ttinclude<tiuser.h>
int t_rcvudata(int fildes, struct t_unitdata *unitdata,
                           int *flags);
int t_sndudata(int fildes, struct t_unitdata *unitdata);
```

Для передачи данных используется структура *unitdata*, имеющая следующие поля:

<b>struct netbuf addr</b>	<b>Адрес удаленного транспортного узла</b>
<b>struct netbuf opt</b>	<b>Опции протокола</b>
<b>struct netbuf udata</b>	<b>Прикладные данные</b>

Созданный транспортный узел может быть закрыт с помощью функции *t\_close(3N)*. Заметим, что при этом соединение, или виртуальный канал, с которым ассоциирован данный узел, в ряде случаев не будет закрыт. Функция *t\_close(3N)* имеет вид:

```
ttinclude <tiuser.h>
int t_close(int fd);
```

где *fd* определяет транспортный узел. Вызов этой функции приведет к освобождению ресурсов, связанных с транспортным узлом, а последующий

системный вызов *close(2)* освободит и файловый дескриптор. Судьба виртуального канала (если таковой существует) зависит от того, является ли транспортный узел, адресующий данный канал, единственным. Если это так, соединение немедленно разрывается. В противном случае, например, когда несколько файловых дескрипторов адресуют один и тот же транспортный узел, виртуальный канал продолжает существовать.

Завершая разговор о программном интерфейсе ТЫ, необходимо упомянуть об обработке ошибок. Для большинства функций ТЫ свидетельством ошибки является получение -1 в качестве возвращаемого значения. Напротив, в случае нормального завершения эти функции возвращают 0. Как правило, при неудачном завершении функции ТЫ код ошибки сохраняется в переменной *t\_errno*, подобно тому, как переменная *errno* хранит код ошибки системного вызова. Для вывода сообщения, расшифровывающего причину ошибки, используется функция *t\_error(3N)*:

```
#include <tiuser.h>
void t_error(const char *errmsg);
```

При вызове *t\_error(3N)* после неудачного завершения какой-либо функции ТЫ будет выведено сообщение *errmsg*, определенное разработчиком программы, за которым следует расшифровка ошибки, связанной с кодом *t\_errno*. Если значение *t\_errno* равно *TSYSERR*, то расшифровка представляет собой стандартное сообщение о системной ошибке, связанной с переменной *errno*.

В заключение в качестве иллюстрации программного интерфейса ТЫ приведем пример приложения клиент-сервер. Как и в предыдущих примерах, сервер принимает сообщения от клиента и отправляет их обратно. Клиент, в свою очередь, выводит полученное сообщение на экран. В качестве сообщения, как и прежде, выступает жизнерадостное приветствие "Здравствуй, мир!".

## Сервер

```
#include <sys/types.h>
#include <sys/socket.h>
#include <tiuser.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <fcntl.h>
#include <netdb.h>

/*Номер порта, известный клиентам*/
#define PORTNUM 1500

main(argc, argv)
int argc;
char *argv[];
```

```
/*Дескрипторы транспортных узлов сервера*/
    int tn, ntn;
    int pid, flags;
    int nport;
/*Адреса транспортных узлов сервера и клиента*/
    struct sockaddr_in serv_addr, *clnt_addr;
    struct hostent *hp;
    char buf[80], hname[80];
    struct t_bind req;
    struct t_call *call;
/*Создадим транспортный узел. В качестве поставщика транспортных услуг
выберем модуль TCP*/
    if((tn=t_open("/dev/tcp", O_RDWR, NULL))==-1)
    {
        t_error("Ошибка вызова t_open()"); exit(1);
    }

/*Зададим адрес транспортного узла — он должен быть известен клиенту */
    nport = PORTNUM;
/*Приведем в соответствие порядок следования байтов для хоста и сети*/
    nport = htons((u_short)nport );
    bzero(&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = nport;
    req.addr maxlen = sizeof(serv_addr);
    req.addr.len = sizeof(serv_addr);
    req.addr.buf = (char *)&serv_addr;
/*Максимальное число запросов, ожидающих обработки, установим равным 5*/
    req.qlen = 5;
/*Связем узел с адресом*/
    if(t_bind(tn, &req, (struct t_bind *)0) < 0)
    {
        t_error("Ошибка вызова t_bind()"); exit(1);
    }
    fprintf(stderr, "Адрес сервера: %s\n",
inet_ntoa(serv_addr.sin_addr));
/*Поскольку в структуре t_call нам понадобится только буфер для хранения
адреса клиента, разместим ее динамически*/
    if ((call =
        (struct t_call*) t_alloc(tn, T_CALL, T_ADDR))==NULL)
    {
        t_error("Ошибка вызова t_alloc()"); exit(2);
    }
    call->addr maxlen = sizeof(serv_addr);
    call->addr.len = sizeof(serv_addr);
    call->opt.len = 0;
    call->udata.len = 0;
```

```

/*Бесконечный цикл получения и обработки запросов*/
while(1)
{
    /*Ждем поступления запроса на установление соединения*/
    if (t_listen(s, call) <0)
    {
        t_error("Ошибка вызова t_listen()"); exit(1);
    }
    /*Выведем информацию о клиенте, сделавшем запрос*/
    clnt_addr = (struct sockaddr_in *)call->addr.buf;
    printf("Клиент: %s\n", inet_ntoa(clnt_addr->sin_addr));
    /*Создадим транспортный узел для обслуживания запроса*/
    if((tn = t_open("/dev/tcp", O_RDWR,
                    (struct t_info *) 0)) <0)
    {
        t_error("Ошибка вызова t_open()"); exit(1);
    }
    /*Пусть система сама свяжет его с подходящим адресом*/
    if (t_bind(ntn, (struct t_bind *)0,
               (struct t_bind *)0) <0)
    {
        t_error("Ошибка вызова t_bind()"); exit(1);
    }
    /*Примем запрос и переведем его обслуживание на новый транспортный
    узел*/
    if (t_accept(tn, ntn, call) <0)
    {
        t_error("Ошибка вызова t_accept()"); exit(1);
    }

    /*Создадим новый процесс для обслуживания запроса. При этом родительский
    процесс продолжает принимать запросы от клиентов*/
    if((pid=fork())==-1)
    {
        t_error("Ошибка вызова fork()"); exit(1);
    }

    if(pid==0)
    {
        int nbytes;

        /*Дочерний процесс: этот транспортный узел уже не нужен, он используется
        родителем*/
        close(tn);
        while ((nbytes = t_rcv(ntn, buf,
                               sizeof(buf), &flags)) !=0)
        {
            t_snd(ntn, buf, sizeof(buf), 0);
        }
        t_close(ntn);
    }
}

```

```

        exit(0);
    }
/*Родительский процесс: этот транспортный узел не нужен, он используется
дочерним процессом для обмена данными с клиентом*/
    t_close(ntn);
}
t_close(tn);
}

```

## Клиент

```

#include <sys/types.h>
#include<sys/socket.h>
#include <tiuser.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include <stdio.h>
#include<fcntl.h>
#include<netdb.h>
#define PORTNUM 1500

main(argc, argv)
char *argv[];
int argc;
{
    int tn;
    iht flags;
    struct sockaddr_in serv_addr;
    struct hostent *hp;
    char buf[80]={"Здравствуй, мир!"};
    struct t_call *call;
/*В качестве аргумента клиенту передается доменное имя хоста, на котором
запущен сервер. Произведем трансляцию доменного имени в адрес*/
    if((hp=gethostbyname(argv[1]))==0)
    {
        perror("Ошибка вызова gethostbyname()"); exit(1);
    }
/*Создадим транспортный узел. В качестве поставщика транспортных услуг
выберем модуль TCP*/
    printf("Сервер готов\n");
    if((tn=t_open("/dev/tcp", O_RDWR, NULL))==-1)
    {
        t_error("Ошибка вызова t_open()"); exit(1);
    }
/*Предоставим системе самостоятельно связать узел с подходящим адресом*/
    if(t_bind(tn, (struct t_bind *)0, (struct t_bind *)0) < 0)
    {
        t_error("Ошибка вызова t_bind()"); exit (1);
    }
    fprintf(stderr, "Адрес клиента: %s\n",
            inet_ntoa(serv_addr.sin_addr));

```

```

/*Укажем адрес сервера, с которым мы будем работать*/
bzero(&serv_addr, sizeof(serv_addr));
bcopy(hp->h_addr, &serv_addr.sin_addr, hp->h_length);
serv_addr.sin_family = hp->h_addrtype;
/*Приведем в соответствие порядок следования байтов для хоста и сети*/
serv_addr.sin_port = htons(PORTNUM);
/*Поскольку в структуре t_call нам понадобится только буфер для хранения
адреса сервера, разместим ее динамически*/
if ( (call =
        (struct t_call*) t_alloc(tn, T_CALL, T_ADDR)) ==NULL)
{
    t_error("Ошибка вызова t_alloc()"); exit(2);
}
call->addr maxlen = sizeof(serv_addr);
call->addr.len = sizeof(serv_addr);
call->addr.buf = (char *) &serv_addr;
call->opt.len = 0;
call->udata.len = 0;
/*Установим соединение с сервером*/
if(t_connect(tn, call, (struct t_call *)0)==-1)
{
    t_error("Ошибка вызова t_connect()"); exit(1);
}
/*Передадим сообщение и получим ответ */
t_snd(tn, buf, sizeof(buf), 0);
if (t_rcv(tn, buf, sizeof(buf), &flags) <0)
{
    t_error("Ошибка вызова t_rcv()"); exit (1);
}
/*Выведем полученное сообщение на экран*/
printf("Получено от сервера: %s\n", buf);
t_close(tn);
printf("Клиент завершил работу!\n\n");
}

```

В рассмотренном примере большая часть исходного текста посвящена созданию транспортных узлов и установлению соединения, в то время как завершение сеанса связи представлено скучными вызовами *t\_close(3N)*. На самом деле, вызов *t\_close(3N)* приводит к немедленному разрыву соединения, запрещая дальнейшую передачу или прием данных. Однако виртуальный канал, обслуживаемый протоколом TCP, является полнодуплексным и, как было показано, TCP предусматривает односторонний разрыв связи, позволяя другой стороне продолжать передачу данных. Действиям, предписываемым TCP, больше соответствуют две функции *t\_sndrel(3N)* и *t\_rcvrel(3N)*, которые обеспечивают *"корректное" прекращение связи* (orderly release). Разумеется, эти рассуждения справедливы лишь для транспортного протокола, обеспечивающего передачу данных с предварительным установлением связи, каковым, в частности, является протокол TCP.

Функции *t\_sndrel(3N)* и *t\_rcvrel(3N)* имеют вид:

```
ttinclude<tiuser.h>
int t sndrel(int fd) ;
int t rcvrel(int fd);
```

Вызывая функцию *t\_sndrel(3N)*, процесс отправляет другой стороне уведомление об одностороннем прекращении связи, это означает, что процесс не намерен больше передавать данные. В то же время процесс может принимать данные — файловый дескриптор fd доступен для чтения.

Другая сторона подтверждает получение уведомления вызовом функции *t\_rcvrel(3N)*. Однако поскольку получение такого уведомления носит асинхронный характер, процесс должен каким-то образом узнать, что запрос поступил. Такой индикацией является завершение с ошибкой попытки получения данных от удаленного узла, например, с помощью функции *t\_rcv(3N)*. В этом случае вызов функции *t\_rcv(3N)* завершится с ошибкой TLOOK.

Эта ошибка свидетельствует, что произошло событие, связанное с коммуникационным узлом, анализ которого позволяет получить дополнительную информацию о причине неудачи. Текущее событие может быть получено с помощью функции *t\_look(3N)*:

Функция возвращает идентификатор, соответствующий одному из событий, перечисленных в табл. 6.6.

**Таблица 6.6.** События, связанные с коммуникационным узлом

<b>Событие</b>	<b>Значение</b>
	Узлом получено подтверждение создания соединения
	Узлом получен запрос на разрыв соединения
T_DATA	Узлом получены данные
T_EXDATA	Узлом получены экстренные данные
T_LISTEN	Узлом получен запрос на установление соединения
T_ORDREL	Узлом получен запрос на корректное прекращение связи
T_ERROR	Свидетельствует о фатальной ошибке
T_UDERR	Свидетельствует об ошибке датаграммы

Если в рассматриваемом случае событием, связанным с ошибкой *t\_rcv(3N)*, является T\_ORDREL, это означает, что удаленный узел завершил передачу данных и более не нуждается в соединении. Если узел, получивший запрос на прекращение связи, не возражает против полного прекращения сеанса,

он вызывает функцию *t\_sndrel(3N)*. Впрочем, при необходимости, коммуникационный узел может продолжить передачу данных. Единственное, отчего ему следует воздержаться, это от попытки получения данных, или, другими словами, от вызова *t\_rcv(3N)*, поскольку в этом случае выполнение процесса будет навсегда заблокировано, т. к. данные от удаленного узла поступать не будут.

Проиллюстрируем описанную процедуру фрагментом программы, обрабатывающей корректное прекращение связи:

```
/*Выполняем обработку принятых данных*/
if (t_errno == T_LOOK && t_look(fd) == T_ORDREL)
{
    /*Значит, получен запрос на корректное прекращение связи. Мы
    согласны на завершение сеанса, поэтому также корректно завершаем
    связь*/
    t_rcvrel(fd);
    t_sndrel(fd);
    exit(0);

    t_error("Ошибка получения данных (t_rcv)");
    exit(1);
}
```

## Программный интерфейс высокого уровня. Удаленный вызов процедур

В предыдущих разделах рассматривался программный интерфейс достаточно низкого уровня — по существу программа взаимодействовала непосредственно с транспортным протоколом, самостоятельно реализуя некоторый протокол верхнего уровня при обмене данными. В приведенных примерах легко заметить, что значительная часть кода этих программ посвящена созданию коммуникационных узлов, установлению и завершению связи.

С точки зрения разработчика программного обеспечения, более перспективным является подход, когда используется прикладной программный интерфейс более высокого уровня, изолирующий программу от специфики сетевого взаимодействия. В данном разделе мы рассмотрим один из таких подходов, на базе которого, в частности, разработана файловая система NFS, получивший название *удаленный вызов процедур* (Remote Procedure Call, RPC).

Использование подпрограмм в программе — традиционный способ структурировать задачу, сделать ее более ясной. Наиболее часто используемые подпрограммы собираются в библиотеки, где могут использоваться различными программами. В данном случае речь идет о локальном (местном) вызове, т. е. и вызывающий, и вызываемый объекты работают в рамках одной программы на одном компьютере.

В случае удаленного вызова процесс, выполняющийся на одном компьютере, запускает процесс на удаленном компьютере (т. е. фактически запускает код процедуры на удаленном компьютере). Очевидно, что удаленный вызов процедуры существенным образом отличается от традиционного локального, однако с точки зрения программиста такие отличия практически отсутствуют, т. е. архитектура удаленного вызова процедуры позволяет сымитировать вызов локальной.

Однако если в случае локального вызова программа передает параметры в вызываемую процедуру и получает результат работы через стек или общие области памяти, то в случае удаленного вызова передача параметров превращается в передачу запроса по сети, а результат работы находится в пришедшем отклике.

Данный подход является возможной основой создания распределенных приложений, и хотя многие современные системы не используют этот механизм, основные концепции и термины во многих случаях сохраняются. При описании механизма RPC мы будем традиционно называть вызывающий процесс — клиентом, а удаленный процесс, реализующий процедуру, — сервером.

Удаленный вызов процедуры включает следующие шаги:

1. Программа-клиент производит локальный вызов процедуры, называемой *заглушки* (stub). При этом клиенту "кажется", что, вызывая заглушки, он производит собственно вызов процедуры-сервера. И действительно, клиент передает заглушки необходимые параметры, а она возвращает результат. Однако дело обстоит не совсем так, как это себе представляет клиент. Задача заглушки — принять аргументы, предназначаемые удаленной процедуре, возможно, преобразовать их в некий стандартный формат и сформировать сетевой запрос. Упаковка аргументов и создание сетевого запроса называется *сборкой* (marshalling).
2. Сетевой запрос пересыпается по сети на удаленную систему. Для этого в заглушки используются соответствующие вызовы, например, рассмотренные в предыдущих разделах. Заметим, что при этом могут быть использованы различные транспортные протоколы, причем не только семейства TCP/IP.
3. На удаленном хосте все происходит в обратном порядке. Заглушка сервера ожидает запрос и при получении извлекает параметры — аргументы вызова процедуры. *Извлечение* (unmarshalling) может включать

необходимые преобразования (например, изменения порядка расположения байтов).

4. Заглушка выполняет вызов настоящей процедуры-сервера, которой адресован запрос клиента, передавая ей полученные по сети аргументы.
5. После выполнения процедуры управление возвращается в заглушку сервера, передавая ей требуемые параметры. Как и заглушка клиента, заглушка сервера преобразует возвращенные процедурой значения, формируя сетевое сообщение-отклик, который передается по сети системе, от которой пришел запрос.
6. Операционная система передает полученное сообщение заглушке клиента, которая, после необходимого преобразования, передает значения (являющиеся значениями, возвращенными удаленной процедурой) клиенту, воспринимающему это как нормальный возврат из процедуры.

Таким образом, с точки зрения клиента, он производит вызов удаленной процедуры, как он это сделал бы для локальной. То же самое можно сказать и о сервере: вызов процедуры происходит стандартным образом, некий объект (заглушка сервера) производит вызов локальной процедуры и получает возвращенные ею значения. Клиент воспринимает заглушку как вызываемую процедуру-сервер, а сервер принимает собственную заглушку за клиента.

Таким образом, заглушки составляют ядро системы RPC, отвечая за все аспекты формирования и передачи сообщений между клиентом и удаленным сервером (процедурой), хотя и клиент и сервер считают, что вызовы происходят локально. В этом-то и состоит основная концепция RPC — полностью спрятать распределенный (сетевой) характер взаимодействия в коде заглушек. Преимущества такого подхода очевидны: и клиент и сервер являются независимыми от сетевой реализации, оба они работают в рамках некой распределенной виртуальной машины, и вызовы процедур имеют стандартный интерфейс<sup>15</sup>.

## Передача параметров

Передача параметров-значений не вызывает особых трудностей. В этом случае заглушка клиента размещает значение параметра в сетевом запросе, возможно, выполняя преобразования к стандартному виду (например, изменяя порядок следования байтов). Гораздо сложнее обстоит дело с пере-

<sup>15</sup> Кроме прочего, благодаря такому подходу, достигается независимость основных компонентов распределенного приложения (клиента и сервера) не только от сетевой реализации, но и от типа операционных систем, под управлением которых они выполняются, и от языка программирования, на котором написаны сами компоненты. Скажем, сервер может быть создан в виде программы на языке C, выполняющейся под управлением UNIX, в то время как в качестве клиента может выступать программа, разработанная на языке Pascal, выполняющаяся в среде Windows NT-

дачей указателей, когда параметр представляет собой адрес данных, а не их значение. Передача в запросе адреса лишена смысла, так как удаленная процедура выполняется в совершенно другом адресном пространстве. Самым простым решением, применяемым в RPC, является запрет клиентам передавать параметры иначе, как по значению, хотя это, безусловно на-кладывает серьезные ограничения<sup>16</sup>.

## Связывание (binding)

Прежде чем клиент сможет вызвать удаленную процедуру, необходимо связать его с удаленной системой, располагающей требуемым сервером. Таким образом, задача связывания распадается на две:

- Нахождение удаленного хоста с требуемым сервером
- Нахождение требуемого серверного процесса на данном хосте

Для нахождения хоста могут использоваться различные подходы. Возможный вариант — создание некоего централизованного справочника, в котором хосты анонсируют свои серверы, и где клиент при желании может выбрать подходящие для него хост и адрес процедуры.

Каждая процедура RPC однозначно определяется номером программы и процедуры. Номер программы определяет группу удаленных процедур, каждая из которых имеет собственный номер. Каждой программе также присваивается номер версии, так что при внесении в программу незначительных изменений (например, при добавлении процедуры) отсутствует необходимость менять ее номер. Обычно несколько функционально сходных процедур реализуются в одном программном модуле, который при запуске становится сервером этих процедур, и который идентифицируется номером программы.

Таким образом, когда клиент хочет вызвать удаленную процедуру, ему необходимо знать номера программы, версии и процедуры, предоставляющей требуемый сервис.

Для передачи запроса клиенту также необходимо знать сетевой адрес хоста и номер порта, связанный с программой-сервером, обеспечивающей требуемые процедуры. Для этого используется демон *portmap(1M)* (в некоторых системах он называется *rpcbind(1M)*). Демон запускается на хосте, который предоставляет сервис удаленных процедур, и использует общеизвестный номер порта. При инициализации процесса-сервера он регистрирует в *portmap(1M)* свои процедуры и номера портов. Теперь, когда клиенту требуется знать номер порта для вызова конкретной процедуры, он посыпает запрос на сервер *portmap(1M)*, который, в свою очередь, либо возвра-

<sup>16</sup> Более сложные среды распределенного программирования (например CORBA) лишены подобных ограничений и обладают рядом дополнительных возможностей, что позволяет с их помощью создавать сложные распределенные системы.

шает номер порта, либо перенаправляет запрос непосредственно серверу удаленной процедуры и после ее выполнения возвращает клиенту отклик. В любом случае, если требуемая процедура существует, клиент получает от сервера *portmap(1M)* номер порта процедуры, и дальнейшие запросы может делать уже непосредственно на этот порт.

### Обработка особых ситуаций (exception)

Обработка особых ситуаций при вызове локальных процедур не представляет собой проблемы. UNIX обеспечивает обработку ошибок процессов, таких как деление на ноль, обращение к недопустимой области памяти и т. д. В случае вызова удаленной процедуры вероятность возникновения ошибочных ситуаций увеличивается. К ошибкам сервера и заглушек добавляются ошибки, связанные, например, с получением ошибочного сетевого сообщения.

Например, при использовании UDP в качестве транспортного протокола производится повторная передача сообщений после определенного таймаута. Клиенту возвращается ошибка, если, спустя определенное число попыток, отклик от сервера так и не был получен. В случае, когда используется протокол TCP, клиенту возвращается ошибка, если сервер оборвал TCP-соединение.

### Семантика вызова

Вызов локальной процедуры однозначно приводит к ее выполнению, после чего управление возвращается в головную программу. Иначе дело обстоит при вызове удаленной процедуры. Невозможно установить, когда конкретно будет выполняться процедура, будет ли она выполнена вообще, а если будет, то какое число раз? Например, если запрос будет получен удаленной системой после аварийного завершения программы сервера, процедура не будет выполнена вообще. Если клиент при неполучении отклика после определенного промежутка времени (тайм-аута) повторно посылает запрос, то может создаться ситуация, когда отклик уже передается по сети, а повторный запрос вновь принимается на обработку удаленной процедурой. В этом случае процедура будет выполнена несколько раз.

Таким образом, выполнение удаленной процедуры можно характеризовать следующей семантикой:

- Один и только один раз.* Данного поведения (в некоторых случаях наиболее желательного) трудно требовать ввиду возможных аварий сервера.
- Максимум раз.* Это означает, что процедура либо вообще не была выполнена, либо была выполнена только один раз. Подобное утверждение можно сделать при получении ошибки вместо нормального отклика.
- Хотя бы раз.* Процедура наверняка была выполнена один раз, но возможно и больше. Для нормальной работы в такой ситуации удаленная

процедура должна обладать свойством идемпотентности (от англ. *idempotent*). Этим свойством обладает процедура, многократное выполнение которой не вызывает кумулятивных изменений. Например, чтение файла идемпотентно, а добавление текста в файл — нет.

## Представление данных

Когда клиент и сервер выполняются в одной системе на одном компьютере, проблем с несовместимостью данных не возникает. И для клиента и для сервера данные в двоичном виде представляются одинаково. В случае удаленного вызова дело осложняется тем, что клиент и сервер могут выполняться на системах с различной архитектурой, имеющих различное представление данных (например, представление значения с плавающей точкой, порядок следования байтов и т. д.)

Большинство реализаций системы RPC определяют некоторые стандартные виды представления данных, к которым должны быть преобразованы все значения, передаваемые в запросах и откликах.

Например, формат представления данных в RPC фирмы Sun Microsystems следующий:

Порядок следования байтов	Старший — последний
Представление значений с плавающей точкой	IEEE
Представление символа	ASCII

## Сеть

По своей функциональности система RPC занимает промежуточное место между уровнем приложения и транспортным уровнем. В соответствии с моделью OSI этому положению соответствуют уровни представления и сеанса. Таким образом, RPC теоретически независим от реализации сети, в частности, от сетевых протоколов транспортного уровня.

Программные реализации системы, как правило, поддерживают один или два протокола. Например, система RPC разработки фирмы Sun Microsystems поддерживает передачу сообщений с использованием протоколов TCP и UDP. Выбор того или иного протокола зависит от требований приложения. Выбор протокола UDP оправдан для приложений, обладающих следующими характеристиками:

- Вызываемые процедуры идемпотентны
- Размер передаваемых аргументов и возвращаемого результата меньше размера пакета UDP — 8 Кбайт.
- Сервер обеспечивает работу с несколькими сотнями клиентов. Поскольку при работе с протоколами TCP сервер вынужден поддерживать соединение с каждым из активных клиентов, это занимает зна-

чительную часть его ресурсов. Протокол UDP в этом отношении является менее ресурсоемким

С другой стороны, TCP обеспечивает эффективную работу приложений со следующими характеристиками:

- Приложению требуется надежный протокол передачи
- Вызываемые процедуры неидемпонентны
- Размер аргументов или возвращаемого результата превышает 8 Кбайт

Выбор протокола обычно остается за клиентом, и система по-разному организует формирование и передачу сообщений. Так, при использовании протокола TCP, для которого передаваемые данные представляют собой поток байтов, необходимо отделить сообщения друг от друга. Для этого, например, применяется *протокол маркировки записей*, описанный в RFC1057 "RPC: Remote Procedure Call Protocol specification version 2", при котором в начале каждого сообщения помещается 32-разрядное целое число, определяющее размер сообщения в байтах.

По-разному обстоит дело и с семантикой вызова. Например, если RPC выполняется с использованием ненадежного транспортного протокола (UDP), система выполняет повторную передачу сообщения через короткие промежутки времени (тайм-ауты). Если приложение-клиент не получает отклик, то с уверенностью можно сказать, что процедура была выполнена ноль или большее число раз. Если отклик был получен, приложение может сделать вывод, что процедура была выполнена хотя бы однажды. При использовании надежного транспортного протокола (TCP) в случае получения отклика можно сказать, что процедура была выполнена один раз. Если же отклик не получен, определенно сказать, что процедура выполнена не была, нельзя<sup>17</sup>.

## Как это работает?

По существу, собственно система RPC является встроенной в программу-клиент и программу-сервер. Отрадно, что при разработке распределенных приложений, не придется вникать в подробности протокола RPC или программировать обработку сообщений. Система предполагает существование соответствующей среды разработки, которая значительно облегчает жизнь создателям прикладного программного обеспечения. Одним из ключевых моментов в RPC является то, что разработка распределенного приложения начинается с определения *интерфейса объекта* — формального описания функций сервера, сделанного на специальном языке. На основании этого интерфейса затем автоматически создаются заглушки

<sup>17</sup> Даже при использовании надежных транспортных протоколов в случае аварийного завершения работы сервера требуется повторное установление связи (после продолжительного тайм-аута) и повторная передача. В этом случае семантика также меняется.

клиента и сервера. Единственное, что необходимо сделать после этого, — написать фактический код процедуры.

В качестве примера рассмотрим RPC фирмы Sun Microsystems.

Система состоит из трех основных частей:

- *rpcgen(l)* — RPC-компилятор, который на основании описания интерфейса удаленной процедуры генерирует заглушки клиента и сервера в виде программ на языке C.
- Библиотека XDR (eXternal Data Representation), которая содержит функции для преобразования различных типов данных в машинно-независимый вид, позволяющий производить обмен информацией между разнородными системами.
- Библиотека модулей, обеспечивающих работу системы в целом.

Рассмотрим пример простейшего распределенного приложения для ведения журнала событий. Клиент при запуске вызывает удаленную процедуру записи сообщения в файл журнала удаленного компьютера.

Для этого придется создать как минимум три файла: спецификацию интерфейсов удаленных процедур **log.x** (на языке описания интерфейса), собственно текст удаленных процедур **log.c** и текст головной программы клиента **main()** — **client.c** (на языке C) .

Компилятор *rpcgen(l)* на основании спецификации **log.x** создает три файла: текст заглушки клиента и сервера на языке C (**log\_clnt.c** и **log\_svc.c**) и файл описаний **log.h**, используемый обеими заглушками.

Итак, рассмотрим исходные тексты программ.

### **log.x**

В этом файле указываются регистрационные параметры удаленной процедуры — номера программы, версии и процедуры, а также определяется интерфейс вызова — входные аргументы и возвращаемые значения. Таким образом, определена процедура RLOG, в качестве аргумента принимающая строку (которая будет записана в журнал), а возвращаемое значение стандартно указывает на успешное или неудачное выполнение заказанной операции.

```
program LOG_PROG {
    version LOG_VER (
        int RLOG (string) = 1;
    } = i;
} = 0x31234567;
```

Компилятор *rpcgen(l)* создает файл заголовков **log.h**, где, в частности, определены процедуры:

**log.h**

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.

ttifndef _LOG_H_RPCGEN
#define _LOG_H_RPCGEN
#include <rpc/rpc.h>

/*Номер программы*/
ftdefine LOG_PROG ((unsigned long)(0x31234567))
ftdefine LOG_VER ((unsigned long)(1))      /*Номер версии*/
#define RLOG ((unsigned long)(1))      /*Номер процедуры*/
extern int *rlog_1();

/*Внутренняя процедура — нам ее использовать не придется*/
extern int log_prog_1_freeresult();
#endif /* !_LOG_H_RPCGEN */
```

Рассмотрим этот файл внимательно. Компилятор транслирует имя RLOG, определенное в файле описания интерфейса, в rlog\_1, заменяя прописные символы на строчные и добавляя номер версии программы с подчеркиванием. Тип возвращаемого значения изменился с int на int \*. Таково правило — RPC позволяет передавать и получать только адреса объявленных при описании интерфейса параметров. Это же правило касается и передаваемой в качестве аргумента строки. Хотя из файла **print.h** это не следует, на самом деле в качестве аргумента функции rlog\_1() также передается адрес строки.

Помимо файла заголовков компилятор *rpcgen(1)* создает модули заглушки клиента и заглушки сервера. По существу, в тексте этих файлов заключен весь код удаленного вызова.

Заглушки сервера является головной программой, обрабатывающей все сетевое взаимодействие с клиентом (точнее, с его заглушки). Для выполнения операции заглушки сервера производит локальный вызов функции, текст которой необходимо написать:

**log. c**

```
#include <rpc/rpc.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "log.h"
int *rlog_1(char **arg)
{
    /*Возвращаемое значение должно определяться как static*/
    static int result;
    int fd;      /*Файловый дескриптор журнала*/
    int len;
```

```

result = 1;

/*Откроем файл журнала (создадим, если он не существует), в
случае неудачи вернем код ошибки result == 1.*/
if !(fd=open("./server.log",
             O_CREAT | O_RDWR | O_APPEND) < 0) return(&result);

len = strlen(*arg);
if (write(fd, *arg, strlen(*arg)) != len)
    result = 1;
else
    result = 0;
close(fd);
return(&result); /*Возвращаем результат - адрес result*/
}

```

Заглушка клиента принимает аргумент, передаваемый удаленной процедуре, делает необходимые преобразования, формирует запрос на сервер *portmap(1M)*, обменивается данными с сервером удаленной процедуры и, наконец, передает возвращаемое значение клиенту. Для клиента вызов удаленной процедуры сводится к вызову заглушки и ничем не отличается от обычного локального вызова.

### ***client.c***

```

#include <rpc/rpc.h>
#include "log.h"
main(int argc, char *argv[])
{
CLIENT      *cl;
char*server, *mystring, *clnttime;
time_t      bintime;
int *result;

if (argc != 2) {
    fprintf(stderr, "Формат вызова: %s Адрес_хоста\n"
                "                  argv[0]);"
    exit(1);
}

server = argv[1];
/*Получим дескриптор клиента. В случае неудачи - сообщим о
невозможности установления связи с сервером*/
if ((cl = clnt_create(server,
                      LOG_PROG, LOG_VER, "udp")) == NULL)
    clnt_pcreateerror(server);
exit(2);
}

```

```

/*Выделим буфер для строки*/
mystring = (char *)malloc(100);
/*Определим время события*/
bintime = time((time_t *)NULL);
clntime = ctime(&bintime);

sprintf (mystring, "%s - Клиент запущен", clntime);

/*Передадим сообщение для журнала - время начала работы клиента.
В случае неудачи - сообщим об ошибке*/
if ((result = rlog_1(&mystring, cl)) == NULL) {
    fprintf(stderr, "error2\n");
    clnt_perror(cl, server);
    exit (3);
}

/*В случае неудачи на удаленном компьютере сообщим об ошибке*/
if (*result !=0 )
    fprintf(stderr, "Ошибка записи в журнал\n");
/*Освободим дескриптор*/
clnt_destroy(cl);

exit(0);
}

```

Заглушка клиента **log\_clnt.c** компилируется с модулем **client.c** для получения исполняемой программы клиента.

```
cc -o rlog client.c log_clnt.c -lnsl
```

Заглушка сервера **log\_svc.c** и процедура **log.c** компилируются для получения исполняемой программы сервера.

```
cc -o logger log_svc.c log.c -lnsl
```

Теперь на некотором хосте `server.nowhere.ru` необходимо запустить серверный процесс:

**\$ logger**

После чего при запуске клиента *Hog* на другой машине сервер добавит соответствующую запись в файл журнала.

Схема работы RPC в этом случае приведена на рис. 6.20. Модули взаимодействуют следующим образом:

1. Когда запускается серверный процесс, он создает сокет UDP и связывает любой локальный порт с этим сокетом. Далее сервер вызывает библиотечную функцию *svc\_register(3N)* для регистрации номеров программы и ее версии. Для этого функция обращается к процессу *portmap(1M)* и передает требуемые значения. Сервер *portmap(1M)* обычно запускается при инициализации системы и связывается с некоторым общеизвестным портом. Теперь *portmap(3N)* знает номер порта

для нашей программы и версии. Сервер же ожидает получения запроса. Заметим, что все описанные действия производятся заглушкой сервера, созданной компилятором *rpcgen(1M)*.

- Когда запускается программа *rlog*, первое, что она делает, — вызывает библиотечную функцию *clnt\_create(3N)*, указывая ей адрес удаленной системы, номера программы и версии, а также транспортный протокол. Функция направляет запрос к серверу *portmap(1M)* удаленной системы *server.nowhere.ru* и получает номер удаленного порта для сервера журнала.
- Клиент вызывает процедуру *rlog\_1()*, определенную в заглушки клиент, и передает управление заглушки. Та, в свою очередь, формирует запрос (преобразуя аргументы в формат XDR) в виде пакета UDP и направляет его на удаленный порт, полученный от сервера *portmap(1M)*. Затем она некоторое время ожидает отклика и в случае неполучения повторно отправляет запрос. При благоприятных обстоятельствах запрос принимается сервером *logger* (модулем заглушки сервера). Заглушка определяет, какая именно функция была вызвана (по номеру процедуры), и вызывает функцию *rlog\_1()* модуля *log.c*. После возврата управления обратно в заглушки последняя преобразует возвращенное функцией *rlog\_1()* значение в формат XDR, и формирует отклик также в виде пакета UDP. После получения отклика заглушки клиента извлекает возвращенное значение, преобразует его и возвращает в головную программу клиента.

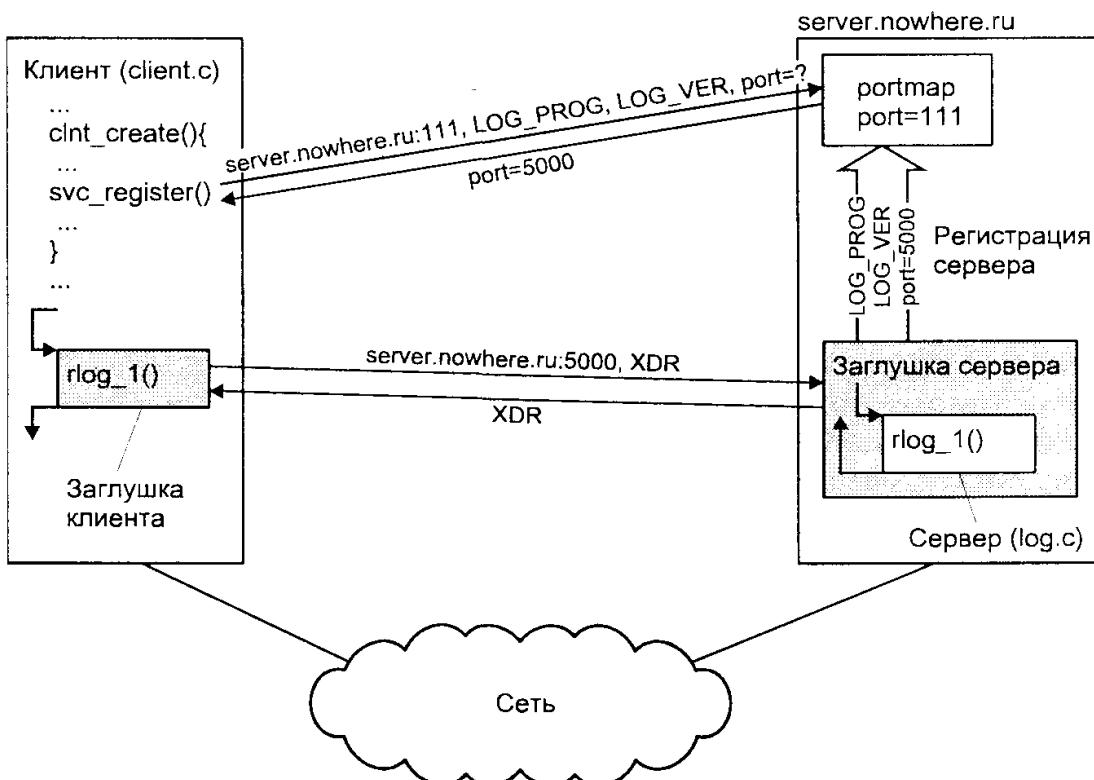


Рис. 6.20. Работа системы RPC

## Поддержка сети в BSD UNIX

Перейдем теперь к обсуждению внутренней архитектуры сетевого доступа в UNIX. Разговор начнем с ветви UNIX, в которой реализация TCP/IP появилась впервые — BSD UNIX.

Сетевая подсистема UNIX может быть представлена состоящей из трех уровней, каждый из которых отвечает за выполнение определенных функций:

<b>Транспортный уровень</b>	Обмен данными между процессами
<b>Сетевой уровень</b>	Маршрутизация сообщений
<b>Уровень сетевого интерфейса</b>	Передача данных по физической сети

Два верхних уровня представляют собой модули коммуникационных протоколов, а нижний уровень по существу является драйвером устройства. Легко заметить, что представленные уровни соответствуют транспортному, сетевому уровням и уровню канала данных модели OSI.

Транспортный уровень является самым верхним в системе и призван обеспечить необходимую адресацию и требуемые характеристики передачи данных, определенных коммуникационным узлом процесса, которым является сокет. Например, сокет потока предполагает надежную последовательную доставку данных, и в семействе TCP/IP модуль данного уровня реализует протокол TCP. Следующий, сетевой, уровень обеспечивает передачу данных, адресованных удаленному сетевому или транспортному модулю. Для этого модуль данного уровня должен иметь доступ к информации о маршрутах сети (таблице маршрутизации). Наконец, последний уровень отвечает за передачу данных хостам, подключенным к одной физической среде передачи (например, находящимся в одном сегменте Ethernet).

Внутренняя структура сетевой подсистемы изолирована от непосредственного доступа прикладных процессов. Единым (и единственным) интерфейсом доступа к сетевым услугам является интерфейс сокетов, рассмотренный в главе 3 в разделе "Межпроцессное взаимодействие в BSD UNIX. Сокеты". Для обеспечения возможности работы с конкретным коммуникационным протоколом соответствующий модуль экспортирует интерфейсы сокетов функцию пользовательского запроса. При этом данные от прикладного процесса передаются от интерфейса сокетов требуемым транспортным модулям с помощью соответствующих вызовов экспортированных функций. И наоборот, данные, полученные из сети, проходят обработку в соответствующих модулях протоколов и помещаются в очередь приема сокета-адресата.

Движение данных вниз (т. е. от верхних уровней к нижним) обычно инициируется системными вызовами и может иметь синхронный характер. Принимаемые данные из сети поступают в случайные моменты времени и передаются сетевым драйвером в очередь приема соответствующего прото-

кола. При этом функции модуля протокола и обработка данных не вызываются непосредственно сетевым драйвером. Вместо этого последний устанавливает бит соответствующего программного прерывания, в контексте которого система позднее и запускает необходимые функции. Если данные предназначены протоколу верхнего уровня (транспортному), его функция обработки будет вызвана непосредственно модулем сетевого уровня. Если же сообщение предназначено другому хосту, и система выполняет функции шлюза, сообщение будет передано уровню сетевого интерфейса для последующей передачи.

Прежде чем более подробно ознакомиться со взаимодействием различных модулей сетевой подсистемы BSD UNIX, рассмотрим сначала структуры данных, определяющие сокет, коммуникационный протокол и сетевой интерфейс.

## Структуры данных

Структура данных socket, описывающая сокет, представлена на рис. 6.21. В этой структуре хранится информация о типе сокета (so\_type), его текущем состоянии (so\_state) и используемом протоколе (so\_proto).

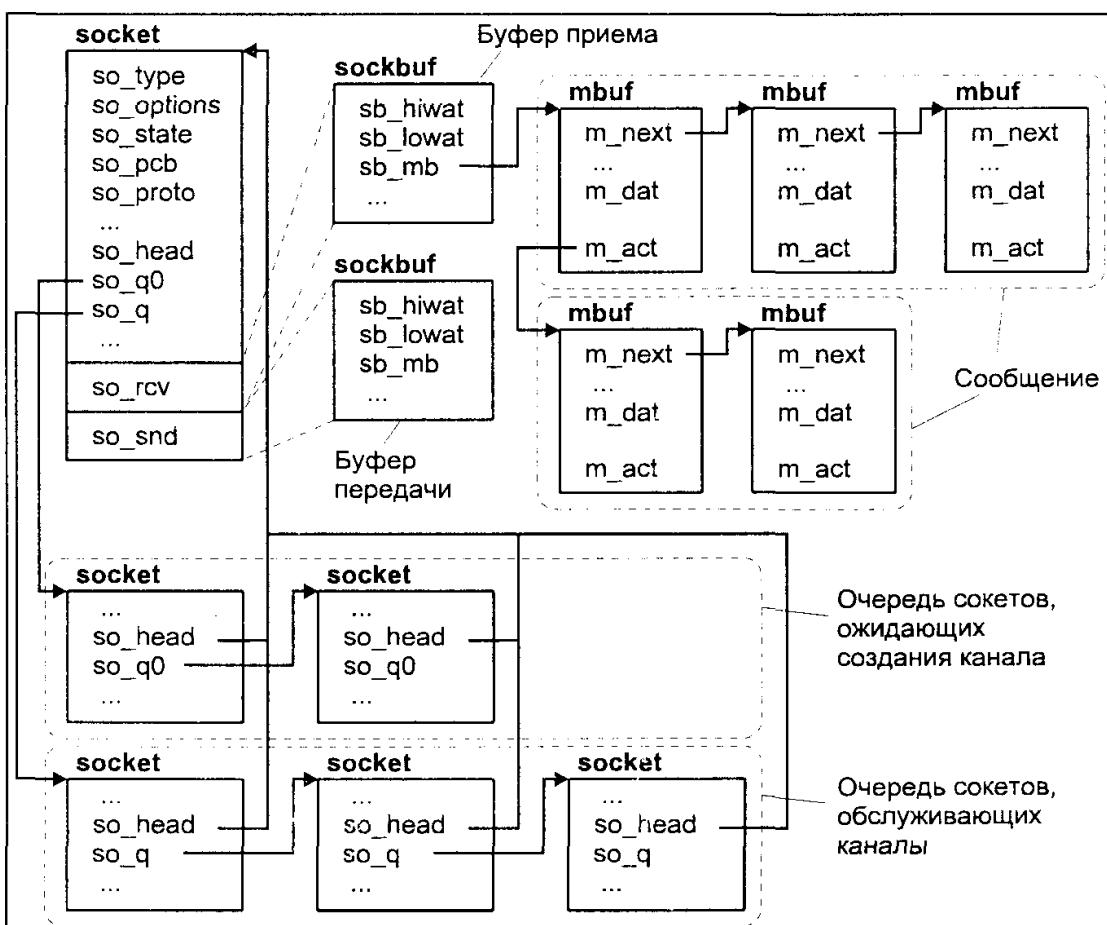


Рис. 6.21. Структуры данных сокета

Сокет является коммуникационным узлом и обеспечивает буферизацию получаемых и отправляемых данных. Как только данные попадают в распоряжение сокета в результате системного вызова (например, *write(2)* или *send(2)*), сокет немедленно передает их модулю протокола для последующего отправления. Данные передаются в виде связанного списка специальных буферов *mbuf*, структура которых также показана на рис. 6.21. Модуль протокола может ожидать подтверждения получения отправленных данных или отложить их отправку. В обоих случаях сообщения остаются в буфере передачи сокета до момента окончательной отправки или получения подтверждения. Аналогично, данные, полученные из сети, в конечном итоге буферизируются в приемной очереди сокета-адресата, пока не будут извлечены оттуда системным вызовом (например, *read(2)* или *recv(2)*).

Для избежания переполнения буфер (структура *sockbuf*) хранит параметр *sb\_hiwater* — значение верхней ватерлинии. Модуль коммуникационного протокола может использовать это значение для управления потоком данных. Например, модуль TCP устанавливает максимальное значение окна приема равным этому параметру.

Сокеты, используемые для приема и обработки запросов на установление связи (зарегистрированные с помощью системного вызова *listen(2)*), адресуют два связанных списка: список сокетов, связь для которых не полностью установлена, и список сокетов, обеспечивающих доступ к созданным каналам передачи данных.

Следующая структура данных, которую мы рассмотрим, относится к коммуникационным протоколам. Каждый модуль протокола представляет собой набор функций обработки и структур данных и описывается структурой данных, называемой *коммутатором протокола*. Коммутатор протокола хранит адреса стандартных функций протокола, например, функций ввода (*pr\_input()*) и вывода (*pr\_output()*), и выполняет ту же роль, что и элемент коммутатора устройств, рассмотренный в главе 5. Поле *so\_proto* сокета содержит адрес этой структуры для соответствующего протокола. Вид коммутатора протокола показан на рис. 6.22.

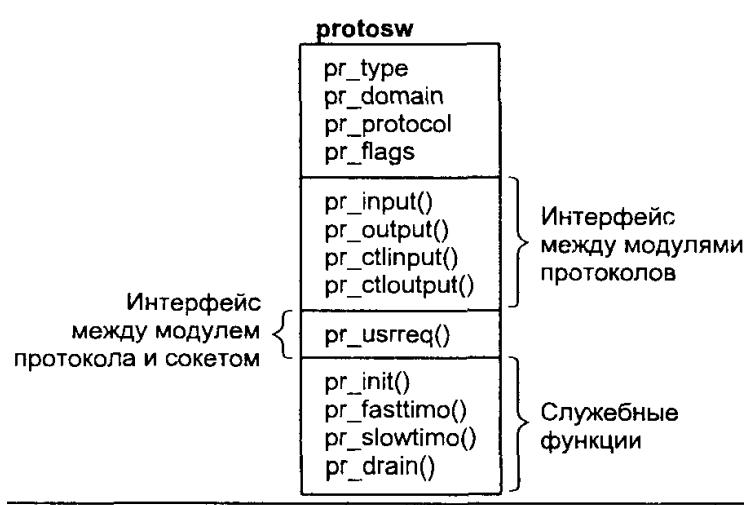


Рис. 6.22. Коммутатор протокола

Перед первым использованием модуля вызывается функция его инициализации `pr_init()`. После этого система будет вызывать функции таймера модуля протокола `pr_fasttim()` каждые 200 миллисекунд и `pr_slowtim()` каждые 500 миллисекунд, если протокол определил эти функции. Например, модуль протокола TCP использует функции таймера для обработки тайм-аутов при установлении связи и повторных передачах. Функция `pr_drain()` вызывается системой при недостатке свободной памяти и позволяет модулю уничтожить некритичные сообщения для освобождения места.

С помощью функции `pr_usrreq()` модулю протокола передаются сообщения от прикладного процесса. Таким образом, эта функция определяет интерфейс взаимодействия между сокетом и протоколом нижнего уровня. Одним из параметров этой функции является номер запроса, зависящий от произведенного системного вызова. Интерфейс взаимодействия сокета с прикладными процессами является стандартным интерфейсом системных вызовов и преобразует вызовы `bind(2)`, `listen(2)`, `send(2)`, `sendto(2)` и т. д. в соответствующие запросы функции `pr_usrreq()`. Некоторые из них приведены в табл. 6.7.

**Таблица 6.7.** Запросы функции `pr_usrreq()`

Системный вызов	Значение	Запрос
<code>close(2)</code>	Прекратить обмен данными	PRU_ABORT
<code>accept(2)</code>	Обработать запрос на установление связи	PRU_ACCEPT
<code>bind(2)</code>	Связать сокет с адресом	PRU_BIND
<code>connect(2)</code>	Установить связь	PRU_CONNECT
<code>listen(2)</code>	Разрешить обслуживание запросов	PRU_LISTEN
<code>send(2)</code> , <code>sendto(2)</code>	Отправить данные	PRU_SEND
<code>fstat(2)</code>	Определить состояние сокета	PRU_SENSE
<code>getsockname(2)</code>	Получить адрес локального сокета	PRU_SOCKADDR
<code>getpeername(2)</code>	Получить адрес удаленного сокета	PRU_PEERADDR
<code>ioctl(2)</code>	Передать команду модулю протокола	PRU_CONTROL

Функции `pr_input()` и `pr_output()` определяют интерфейс взаимодействия протокол-протокол и служат для передачи данных между модулями соседних уровней. Аналогично для обмена управляющими командами между модулями протоколов используются функции `pr_ctlinput()` и `pr_ctloutput()`. Цепочка взаимодействующих протоколов производит размещение и освобождение памяти при обмене сообщениями, которые передаются посредством рассмотренных структур `mbuf`: при передаче сообщений от сети прикладному процессу за освобождение буферов `mbuf` отвечает модуль верхнего уровня и наоборот, при передаче сообщений в сеть память, занимаемая сообщением, освобождается на самом нижнем уровне.

Поле `pr_flags` определяет некоторые характеристики протокола и режим его функционирования, которые в основном относятся к уровню сокетов. Например, протоколы, предусматривающие предварительное установление связи, указывают это с помощью флага `PR_CONNREQUIRED`, не позволяя тем самым функциям сокета передавать данные модулю до создания виртуального канала. Если установлен флаг `PR_WANTRCV`, соответствующие функции сокета будут уведомлять модуль протокола, когда прикладной процесс получает данные из буфера приема. Это может служить сигналом протоколу для отправления подтверждения о получении, а также для обновления значения окна в соответствии с освободившимся местом.

Заметим, что каждый модуль протокола имеет собственные очереди сообщений, используемые для приема и передачи данных.

Каждый сетевой интерфейс системы представлен структурой данных, показанной на рис. 6.23. Сетевой интерфейс обычно связан с соответствующим сетевым адаптером, хотя это не является обязательным условием. Например, внутренний сетевой интерфейс `loopback` представляет собой псевдоустройство, используемое для унифицированного взаимодействия сетевых процессов в рамках одного хоста, отладки и т. п.

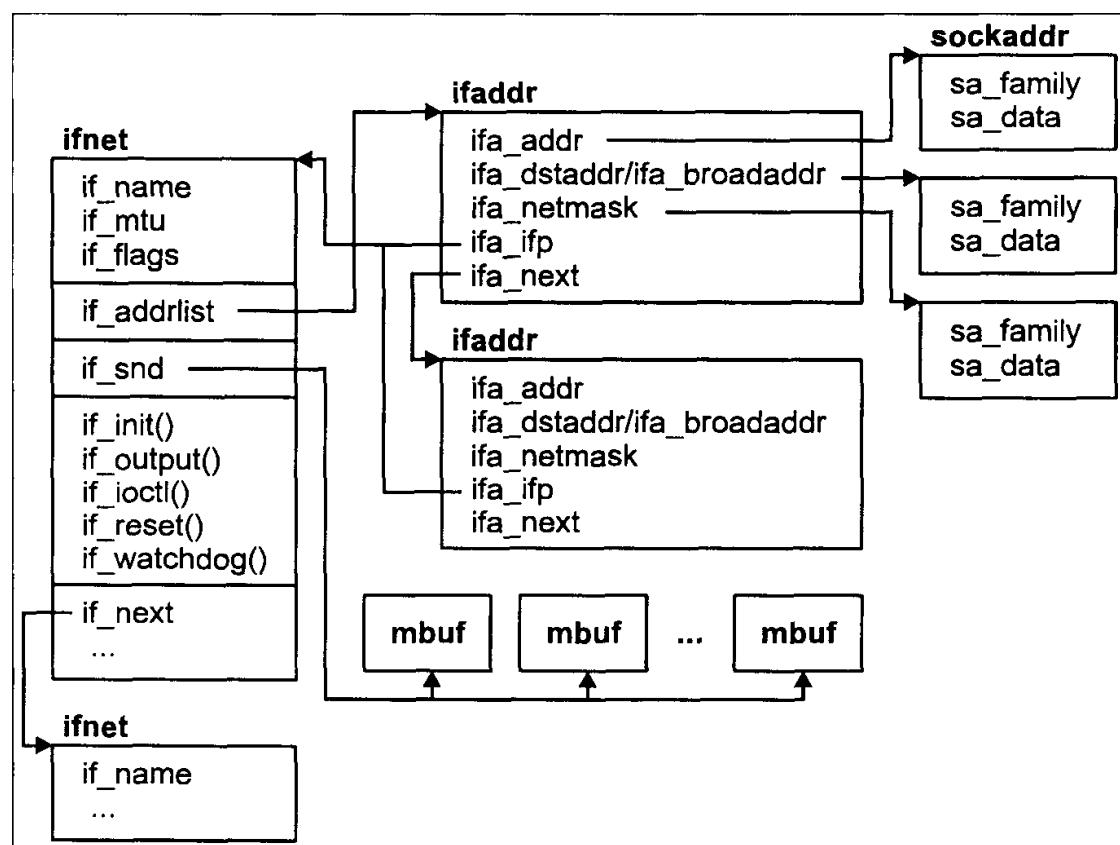


Рис. 6.23. Сетевой интерфейс

Решение об использовании того или иного сетевого интерфейса для передачи сообщения базируется на таблице маршрутизации и производится

модулем сетевого уровня. Интерфейс может обслуживать протоколы различных коммуникационных доменов. Соответственно, один и тот же интерфейс может иметь несколько адресов, определенных для каждого семейства протоколов. Структуры, определяющие локальный и широковещательный (broadcast) адреса интерфейса, а также сетевую маску, хранятся в виде связанного списка.

Каждый сетевой интерфейс имеет очередь, в которую помещаются сообщения для последующей передачи, выполняемой функцией `if_output()`. Интерфейс также может определить процедуры инициализации `if_init()`, сброса `if_reset()` и обработки таймера `if_watchdog()`. Последняя может использоваться для управления потенциально ненадежными устройствами или для периодического сбора статистики устройства.

Состояние интерфейса характеризуется флагами, хранящимися в поле `if_flags`. Возможные флаги приведены в табл. 6.8.

**Таблица 6.8.** Состояния интерфейса

Флаг	Значение
<code>IFF_UP</code>	Интерфейс доступен для использования
<code>IFF_BROADCAST</code>	Интерфейс поддерживает широковещательные адреса
<code>IFF_MULTICAST</code>	Интерфейс поддерживает групповые адреса
<code>IFF_DEBUG</code>	Интерфейс обеспечивает возможность отладки
<code>IFF_LOOPBACK</code>	Программный внутренний интерфейс
<code>IFF_POINTOPOINT</code>	Интерфейс для канала точка-точка
<code>IFF_RUNNING</code>	Ресурсы интерфейса успешно размещены
<code>IFF_NOARP</code>	Интерфейс не использует протокол трансляции адреса

Флаг `IFF_UP` свидетельствует о готовности интерфейса передавать сообщения. Если сетевой интерфейс подключен к физической сети, поддерживающей широковещательную адресацию (broadcast), например, Ethernet, для интерфейса будет установлен флаг `IFF_BROADCAST` и определен широковещательный адрес (поле `ifa_broadaddr` структуры адресов `ifa_addr` для соответствующего коммуникационного домена). Если же интерфейс используется для канала точка-точка, будет установлен флаг `IFF_POINTOPOINT` и определен адрес хоста (интерфейса), расположенного на противоположном конце (поле `ifa_dstaddr`). Заметим, что эти два флага являются взаимоисключающими, а `ifa_broadaddr` и `ifa_dstaddr` являются различными именами одного и того же поля. Интерфейс устанавливает флаг `IFF_RUNNING` после размещения необходимых структур данных и отправления начального запроса на чтение устройству (например, сетевому адаптеру), с которым он ассоциирован.

Состояние интерфейса и ряд других параметров можно просмотреть с помощью команды *ifconfig(1M)*:

```
$ ifconfig le0
le0: flags=863<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
      inet 194.85.160.50 netmask ffffff00 broadcast 194.85.160.255
```

Легко заметить, что команда выводит значение следующих полей структуры *ifnet* для интерфейса *le0* (*if\_name*): *if\_flags*, *if\_mtu* (Maximum Transmission Unit, MTU) определяющее максимальный размер пакета, который может быть передан по физической сети, а также значения полей структуры *ifaddr*: адрес интерфейса *inet* (*ifa\_addr*), маску *netmask* (*ifa\_netmask*) и широковещательный адрес *broadcast* (*ifa\_broadaddr*).

Интерфейс хранит статистическую информацию, которая может быть использована при мониторинге сети. В частности, эта информация включает число полученных пакетов уровня канала (*if\_ipackets*), количество ошибок при приеме (*if\_ierrors*), число отправленных пакетов уровня канала (*if\_opackets*), количество ошибок при передаче (*if\_oerrors*) и число коллизий (*if\_collisions*). Команда *netstat(1M)* позволяет получить эту информацию для сконфигурированных интерфейсов в системе:

```
$ netstat -in
Name  Mtu    Net/Dest      Address        Ipkts    Ierrs    Opkts    Oerrs    Collis
lo0    823    127.0.0.0    127.0.0.1    168761    0        168761    0        0
le0    1500   194.85.160.0 194.85.160.50 1624636   1042    110166   1933    382604
```

## Маршрутизация

Сетевая подсистема предназначена для работы в коммуникационной среде, представляющей собой набор сетевых сегментов, связанных между собой. Связь между отдельными сегментами достигается путем подключения их к хостам, имеющим несколько различных сетевых интерфейсов, как показано на рис. 6.24. Такие хосты при необходимости выполняют передачу данных от одного сегмента к другому (*forwarding*)<sup>18</sup>. Для сетей пакетной коммутации, о которых идет речь, выполнение этой задачи непосредственно связано с выбором маршрута прохождения пакетов данных (*routing*). Для этого система хранит *таблицы маршрутизации*, которые используются протоколами сетевого уровня (например, IP) для выбора требуемого интерфейса для передачи пакета адресату.

Маршрутизационная информация хранится в виде двух таблиц, одна из которых предназначена для маршрутов к хостам, а другая — для маршрутов к сетям. Такой подход позволяет использовать универсальные меха-

<sup>18</sup> Заметим, что каждый интерфейс такого хоста-шлюза имеет собственный адрес, соответствующий той сети, к которой он непосредственно подключен. Например, для сетей с разделяемой средой передачи сетевая часть этого адреса равна адресу сети.

низмы определения маршрута как для сетей с разделяемой средой передачи (например, Ethernet), так и для сетей с каналами типа точка-точка. Например, для доставки пакета удаленному хосту, подключенному к сети первого типа, достаточно знать адрес этой сети, в то время как для каналов точка-точка необходимо явно задать адрес интерфейса противоположного конца канала<sup>19</sup>.

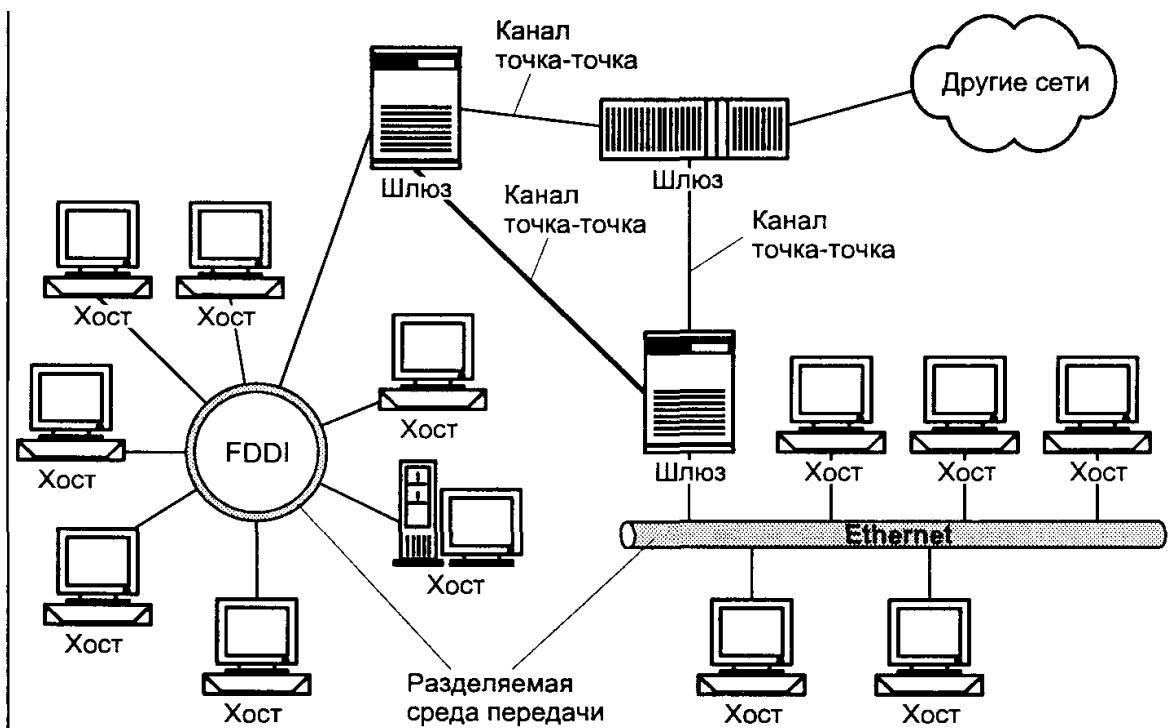


Рис. 6.24. Коммуникационная среда UNIX (internetwork)

При определении маршрута модуль сетевого протокола (IP) сначала просматривает элементы таблицы для хостов, а затем для сетей. Если оба поиска не дают результата, используется *маршрут по умолчанию* (если такой установлен), определенный как маршрут в сеть с адресом 0. Обычно используется первый найденный маршрут. Таким образом, порядок поиска обеспечивает приоритетность маршрутов к хостам по отношению к маршрутам к сетям, что естественно, поскольку первые представлены более конкретными адресами.

Каждый элемент таблицы маршрутизации, показанный на рис. 6.25, содержит адрес получателя (это может быть адрес сети получателя или адрес

<sup>19</sup> Вспомним, что IP-адрес состоит из двух частей — адреса сети и адреса хоста в этой сети. Для интерфейса, подключенного к разделяемой среде, какой является большинство локальных сетей, существенным является лишь первая часть адреса-получателя, поскольку через этот интерфейс непосредственно доступны все хосты с данным адресом сети. Напротив, через сетевой интерфейс типа точка-точка непосредственный доступ осуществляется к единственному хосту, расположенному на другом конце канала, и, таким образом, необходимо определение полного адреса удаленного интерфейса.

конкретного хоста). Это значение хранится в поле `rt_dst`. Следующее поле, `rt_gateway`, определяет следующий шлюз, которому необходимо направить пакет, чтобы последний в конечном итоге достиг адресата. Поле `rt_flags` определяет тип маршрута (к хосту или к сети), а также его состояние. В поле `rt_use` хранится число переданных по данному маршруту пакетов, а `rt_refcnt` определяет использование маршрута сетевыми процессами (виртуальными каналами). Наконец, поле `rt_ifp` адресует сетевой интерфейс, которому необходимо направить пакет для дальнейшей передачи по данному маршруту.

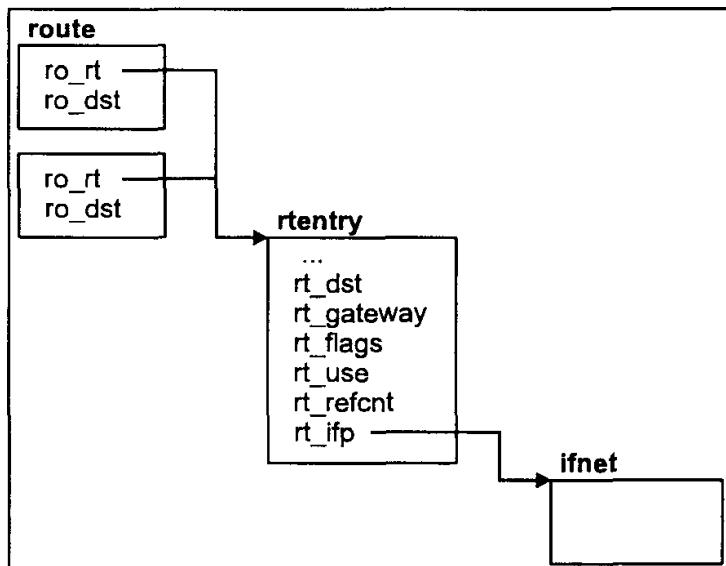


Рис. 6.25. Элемент таблицы маршрутизации

Различают не только маршруты к хостам и сетям, но также маршруты *прямые* (*direct*) и *косвенные* (*indirect*). Первое различие определяет критерий сравнения адреса получателя пакета с полем `rt_dst` элемента таблицы маршрутизации. Если маршрут к сети, то сравнивается только сетевая часть адреса, в противном случае требуется полное совпадение адресов.

Определение маршрута как прямого или косвенного зависит от того, имеется ли непосредственная связь между получателем, указанным в поле `rt_dst`, и сетевым интерфейсом, обслуживающим данный маршрут. Например, маршрут в сеть, непосредственно подключенную к сетевому интерфейсу, является прямым. Напротив, маршрут по умолчанию является косвенным маршрутом, поскольку всегда адресует получателя, расположенного вне непосредственно доступных сетевых сегментов. Эта информация необходима при формировании кадра уровня канала данных. Если пакет адресован хосту или сети, которые непосредственно не подключены к сетевому интерфейсу, то, хотя сетевой адрес этого пакета будет равен сетевому адресу фактического получателя данных, заголовок уровня канала данных будет адресовать соседний шлюз, используемый для дальнейшей передачи пакета. Если пакет не выходит за пределы непосредственно подключенной сети, адреса и сетевого уровня и уровня канала будут совпадать с соответствующими адресами фактического получателя.

Данный аспект проиллюстрирован на рис. 6.26. Здесь мы рассмотрели процесс передачи IP-датаграммы хосту, расположенному в удаленном сетевом сегменте Ethernet. Поскольку доставка датаграммы предполагает использование промежуточного шлюза, передача данных на канальном уровне требует соответствующей адресации: на первом "перегоне" в качестве адреса получателя используется MAC-адрес шлюза, и только затем — MAC-адрес фактического адресата.

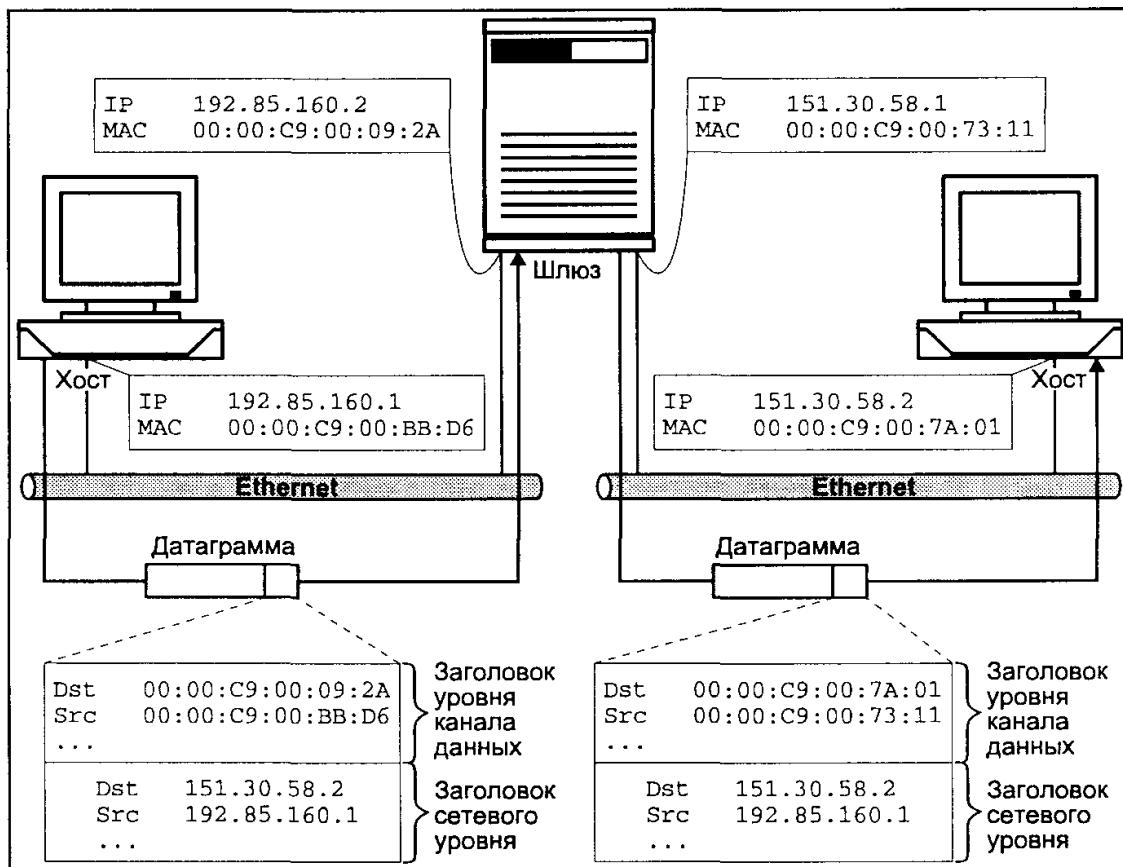


Рис. 6.26. Инкапсуляция пакетов для косвенных маршрутов

На то, что маршрут является косвенным, указывает флаг `RTF_GATEWAY` элемента таблицы маршрутов. В этом случае MAC-адрес получателя при формировании кадра канального уровня, будет определяться исходя из сетевого адреса шлюза, хранящегося в поле `rt_gateway`<sup>20</sup>.

Модуль протокола имеет возможность доступа к маршрутизационной информации с помощью трех функций: `rtalloc()` для получения маршрута, `rtfree()` для его освобождения и `rtredirect()` для обработки управляющих сообщений о перенаправлении маршрута (ICMP REDIRECT).

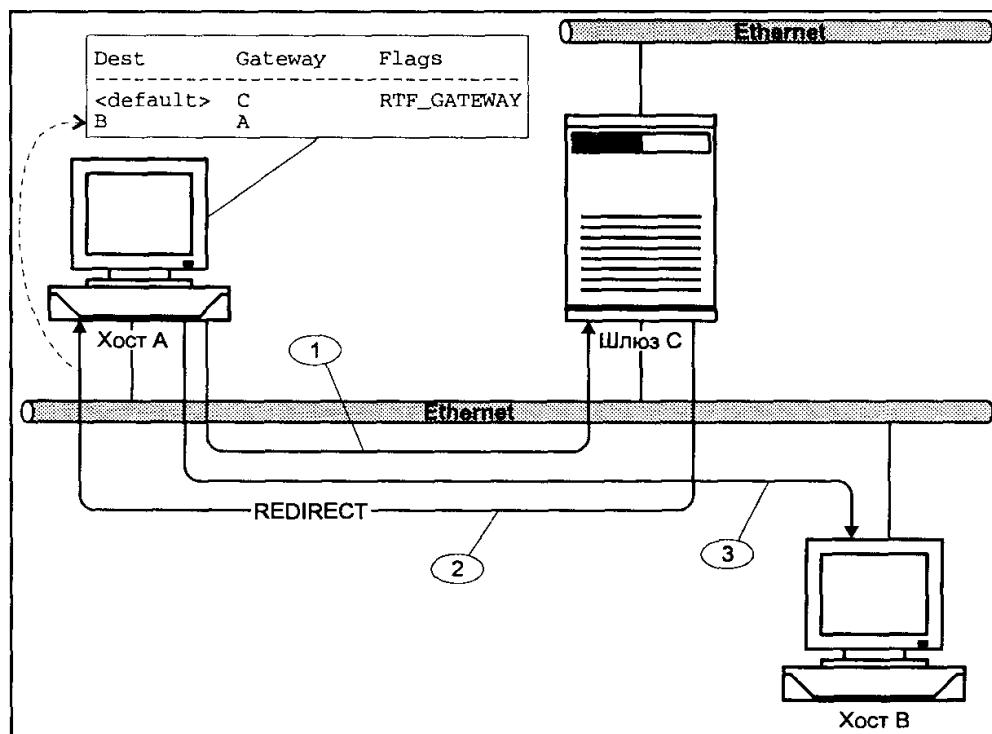
<sup>20</sup> Для определения соответствия между IP-адресами интерфейсов и их MAC-адресами используется протокол ARP (Address Resolution Protocol), позволяющий производить формирование адреса кадра уровня канала данных.

Функция `rtalloc()` позволяет модулю протокола определить маршрут к требуемому адресату. В результате модуль размещает структуру `route`, имеющую следующие поля:

<code>struct rtentry *ro_rt</code>	<b>Указатель на соответствующий элемент таблицы маршрутизации</b>
<code>struct sockaddr ro_dst</code>	<b>Адрес получателя данных</b>

Возвращаемый функцией `rtalloc()` маршрут может быть освобожден с помощью функции `rtfree()` (это не означает, что маршрут будет удален из таблицы маршрутизации). Время жизни маршрута зависит от протокола верхнего уровня. Например, модуль протокола TCP хранит маршрут на протяжении жизни виртуального канала.

Функция `rtredirect()` обычно вызывается модулем протокола в ответ на получение от соседних шлюзов управляющих сообщений о перенаправлении маршрута<sup>21</sup>. Шлюз генерирует такое сообщение в случае, когда обнаружен более предпочтительный маршрут для передаваемого пакета. Например, если хосты А и В находятся в одной и той же сети, и хост А направляет пакеты В через шлюз С, последний отправит А сообщение о перенаправлении маршрута, информирующее, что А в дальнейшем должен посыпать данные В непосредственно. Этот процесс показан на рис. 6.27.



**Рис. 6.27.** Пере- направление мар- шрутов

<sup>21</sup> В семействе протоколов TCP/IP для этих целей служит протокол ICMP. Сообщения о перенаправлении маршрутов ICMP REDIRECT формируются IP-модулем шлюза и информируют IP-модули соседних хостов (шлюзов) о существовании более выгодного маршрута к данному адресату.

Данная возможность может использоваться для упрощения процедуры формирования таблицы маршрутизации. Например, рабочие станции могут хранить только маршрут по умолчанию (в сеть 0), адресующий соседний шлюз. При передаче данных хостам той же сети, что и источник, шлюз будет информировать последний о перенаправлении маршрутов, позволяя тем самым заполнить элементы маршрутационной таблицы.

Функция `rtredirect()` вызывается с параметрами, указывающими на адрес получателя, новый адрес шлюза, который необходимо миновать для достижения адресата, а также источник перенаправления маршрута. Заметим, что сообщения о перенаправлении маршрута принимаются только от текущего шлюза для данного получателя. Если существует маршрут, отличный от маршрута по умолчанию, то для него изменяется поле `rt_gateway` согласно указанному в сообщении новому адресу шлюза. В противном случае создается новая запись таблицы маршрутизации.

Вопросы определения маршрутов в UNIX являются прерогативой специальных прикладных процессов, а не ядра операционной системы. Ядро размещает и хранит необходимую маршрутационную информацию, а также обеспечивает интерфейс доступа к этой информации. Процесс имеет возможность добавить или удалить маршрут с помощью системного вызова `ioctl(2)`. Для добавления маршрута используется команда `SIOCADDRT`, а для удаления — `SIOCDELRT`.

В качестве процессов, отвечающих за заполнение таблиц маршрутологии и ее динамическое обновление, можно назвать стандартный демон *routed(1M)*, использующий протокол RIP (Routing Information Protocol) для динамического определения и обновления маршрутов, а также демон *gated(1M)*, поддерживающий работу нескольких протоколов обмена маршрутационной информацией (**RIP, OSPF, BGP**).

Текущую таблицу маршрутологии можно увидеть, воспользовавшись командой *netstat(1M)*:

```
$ netstat -rn
```

Routing Table:

Destination	Gateway	Flags	Ref	Use	Interface
127.0.0.1	127.0.0.1	UH	0	5054	lo0
194.85.160.0	194.85.160.50	U	3	30926	1e0
default	194.85.160.1	UG	0	47150	1e0

Первая запись таблицы показывает маршрут для псевдохоста (`localhost`) логической сети операционной системы. Следующий маршрут адресует непосредственно подключенную к интерфейсу (его адрес 194.85.160.50) сеть (194.85.160.0). Наконец, последняя запись определяет маршрут по умолчанию, направляя все пакеты, адресованные получателям "внешнего мира", для которых наш хост не знает конкретных маршрутов, на шлюз с адресом 194.85.160.1, который обладает большей информацией о возможных маршрутах.

## Реализация TCP/IP

Прежде чем перейти к описанию функционирования модулей протоколов TCP/IP, рассмотрим еще одну структуру данных, называемую *управляющим блоком протокола* (Protocol Control Block, PCB), который в случае TCP/IP называется *Internet PCB*, и представлен структурой `inpcb`, определенной в файле `<netinet/in_pcbs.h>`. Вид структуры `inpcb` показан на рис. 6.28.

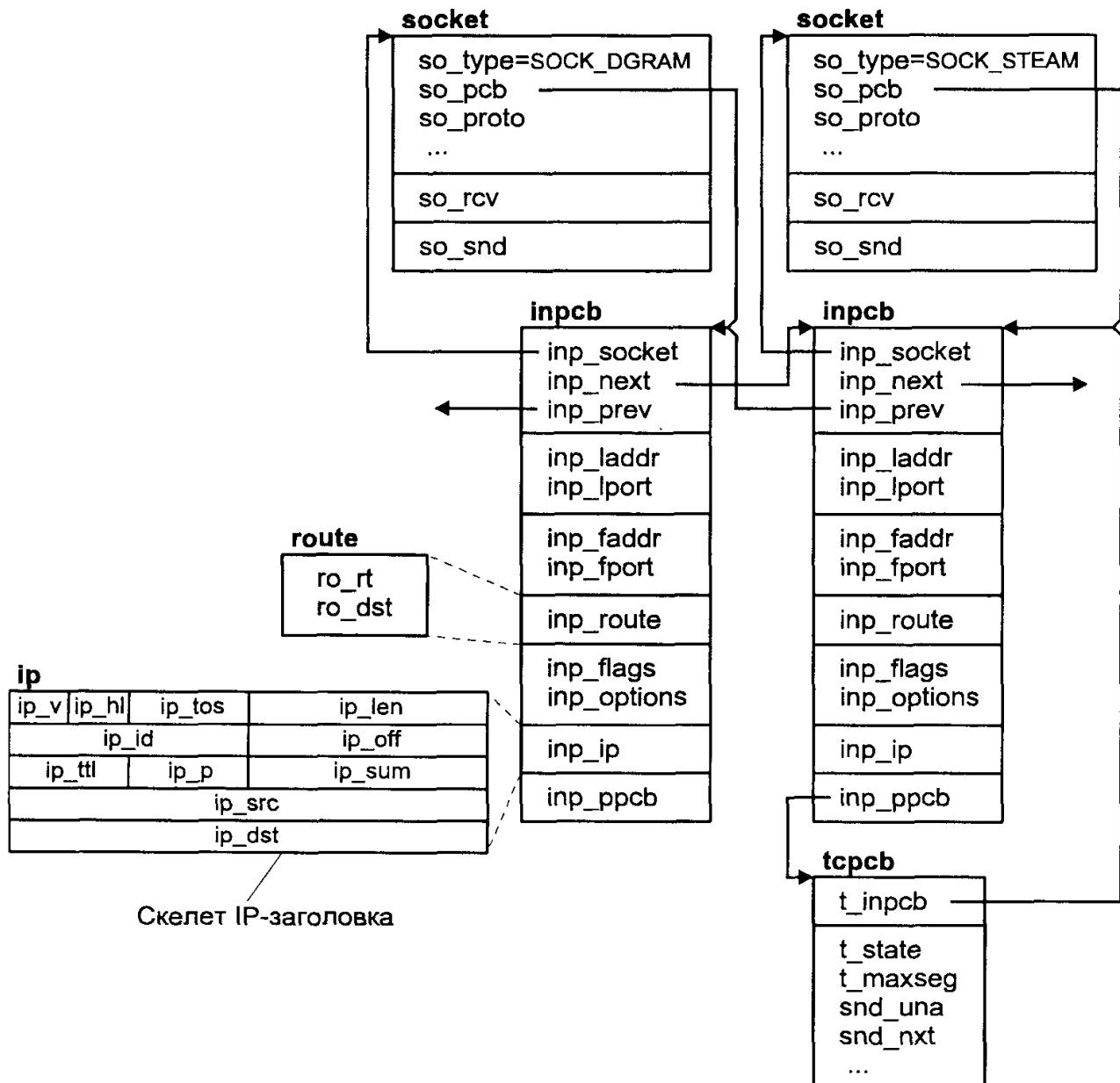


Рис. 6.28. Структуры данных протоколов TCP/IP

Эта структура создается для каждого активного сокета TCP или UDP и содержит информацию, необходимую для текущих транзакций протокола, такую как IP-адреса источника и получателя (`inp_laddr` и `inp_faddr`), номера портов (`inp_lport` и `inp_fport`), маршрутизационной информа-

ции (`inp_route`). TCP создает дополнительный управляющий блок, где хранятся данные, необходимые для работы этого протокола (такие как порядковые номера, номера подтверждений и т. д.)

Управляющие блоки размещаются в виде связанного списка, отдельного для TCP и UDP. Модули протокола имеют в своем распоряжении набор функций для создания, поиска и удаления управляющего блока. Модуль IP демультиплексирует сообщения на основании номера протокола, указанного в заголовке датаграммы, а протокол транспортного уровня, в свою очередь, производит поиск требуемого управляющего блока для доставки данных протоколам более высокого уровня (приложений).

Перейдем теперь к описанию взаимодействия рассмотренных модулей в сетевой подсистеме BSD UNIX (рис. 6.29).

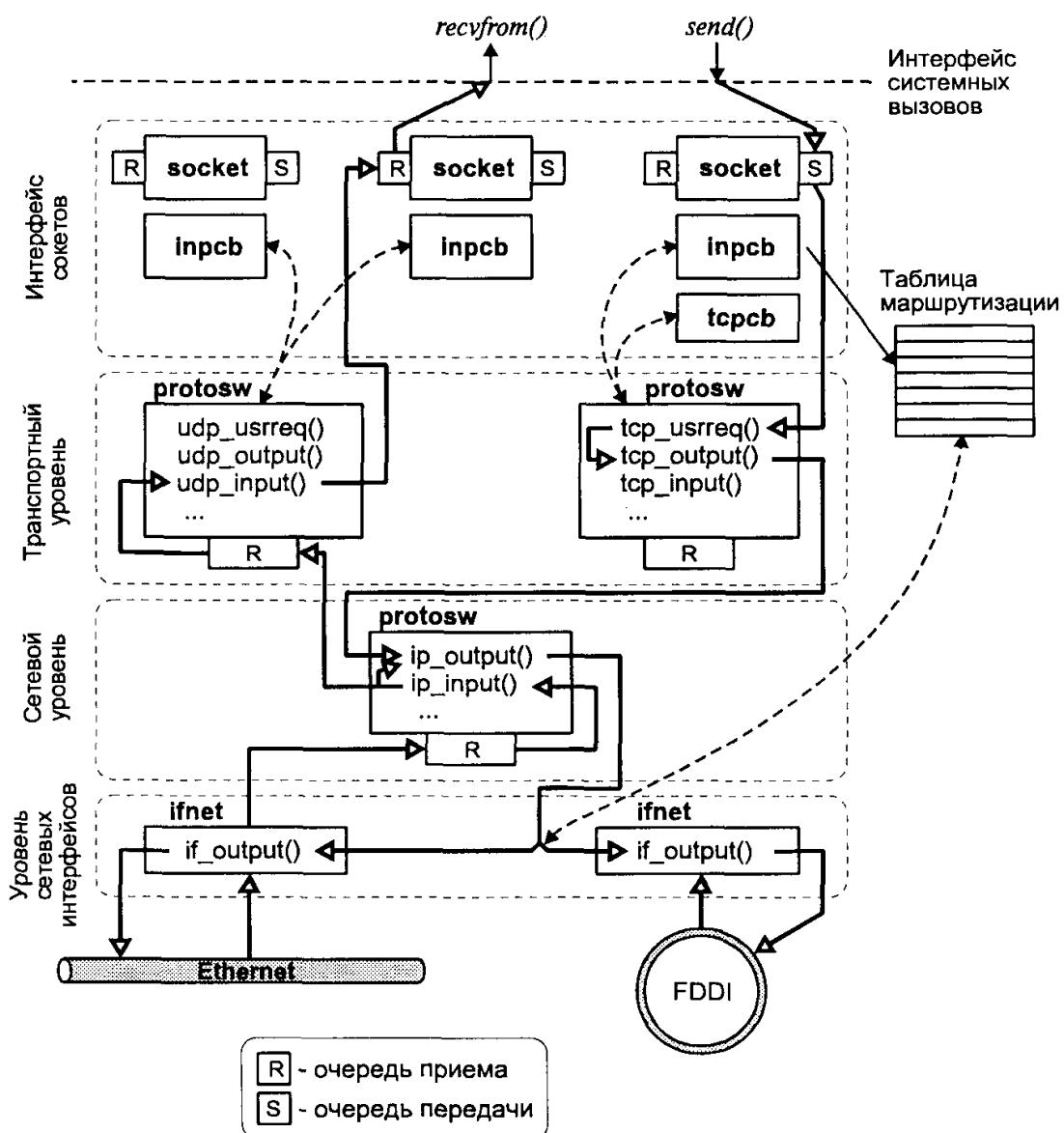


Рис. 6.29. Сетевая подсистема BSD UNIX

## Модуль IP

Сетевой интерфейс получает пакеты данных из сети и передает их соответствующему модулю сетевого уровня на основании информации, содержащейся в заголовке кадра уровня канала. В данном разделе мы не будем рассматривать поддержку различных сетевых протоколов, а остановимся только на взаимодействии с протоколом IP. В этом случае полученные пакеты помещаются в очередь приема модуля IP. После этого с помощью программного прерывания вызывается процедура `ip_input()`, которая поочередно извлекает пакеты из очереди и обрабатывает их. После обработки на основании информации заголовка IP-датаграммы данные либо передаются протоколу транспортного уровня, либо уничтожаются, если в данных обнаружена ошибка, либо передаются другому интерфейсу для последующей отправки фактическому адресату. В последнем случае система выполняет роль шлюза.

Датаграмма считается адресованной данному хосту, если адрес получателя совпадает с одним из адресов интерфейса данного хоста, или адрес получателя является широковещательным (или групповым) адресом данной сети. В случае получения фрагментированной датаграммы модуль производит ее реассемблирование. Для этого отдельные фрагменты собираются в специально организованной очереди, пока не будет сформирована исходная датаграмма. После этого данные передаются транспортному протоколу. Для демультиплексирования модуль IP использует поле `Protocol` заголовка, которое по существу является индексом таблицы, каждый элемент которой представлен коммутатором протокола, рассмотренного ранее в этой главе. Соответственно модуль IP имеет возможность непосредственно вызвать функцию `pr_input()` требуемого протокола следующего уровня.

В случае, когда полученная датаграмма не содержит ошибок, но не адресована данному хосту, она, возможно, должна быть передана на другой сетевой интерфейс для последующей передачи фактическому адресату. Эта процедура носит название *шлюзования* (*forwarding*) и включает выполнение следующих шагов:

- О Производится проверка разрешения шлюзования<sup>22</sup>. В случае отрицательного результата хост не может выполнять функции шлюза и данные уничтожаются.

<sup>22</sup> Возможность передачи на другой интерфейс определяется установкой соответствующего флага при конфигурации сетевой подсистемы (модуля IP). Например, в операционной системе SCO UNIX за это отвечает настраиваемый параметр ядра `ip_forwarding`.

- Производится проверка адреса получателя. Если адрес датаграммы не принадлежит адресному пространству сетей класса A, B или C, такие данные не могут быть переданы<sup>23</sup>.
- Определяется дальнейший маршрут передачи датаграммы.
- Если дальнейший путь датаграммы проходит через тот же интерфейс, с которого она была получена, и хост-отправитель расположен в той же сети, ему отправляется сообщение ICMP REDIRECT.
- Производится вызов функции `ip_output()`, выполняющей передачу датаграммы хосту-адресату или соседнему шлюзу для дальнейшей передачи.

При выполнении этих функций модуль IP может обнаружить несколько ошибочных ситуаций, например, отсутствие маршрута для датаграммы или невозможность передачи данных из-за переполнения в сети. В этих случаях модуль формирует соответствующее сообщение ICMP и передает его отправителю датаграммы. Эти сообщения ICMP и причины их отправки приведены в табл. 6.9.

**Таблица 6.9.** Сообщения ICMP

Сообщение	Причина
DESTINATION UNREACHABLE	Невозможно доставить датаграмму. Причин может быть несколько: <ol style="list-style-type: none"> <li>1. Отсутствует маршрут к сети</li> <li>2. Отсутствует маршрут к хосту</li> <li>3. Для передачи необходима фрагментация, но в заголовке установлен флаг DF (Don't Fragment)</li> </ol>
SOURCE QUENCH	Переполнение сети. Шлюз передает это сообщение, запрашивая отправителя на уменьшение скорости передачи данных
TIME EXCEEDED	Тайм-аут. Причины могут быть две: <ol style="list-style-type: none"> <li>1. Истекло время жизни датаграммы в сети (TTL=0)</li> <li>2. Произошел тайм-аут реассемблирования, т. е. через определенный промежуток времени получены не все фрагменты датаграммы</li> </ol>

При вызове функции `ip_output()` ей передается датаграмма, которую необходимо отправить, указатель на маршрут (структура `route`, хранящаяся в управляемом блоке), а также флаги (например, указание не использовать маршрутизационные таблицы). Передача маршрута не является обя-

<sup>23</sup> Адреса сетей класса D — *групповые* (multicast) адреса — используются для создания специальных наложенных сетей (Multicast backbone, Mbone), предназначенных для таких приложений, как видео-, аудиоконференции и т. п. Обработка таких датаграмм выполняется, как правило, специальными демонами отдельно от стандартных функций шлюзования. Если в системе включена поддержка групповых адресов, данные с указанными адресами будут передаваться этим демонам, которые и выполнят логическое шлюзование/передачу.

зательной. Если функции не передан указатель на маршрут, будет использован маршрут из таблицы маршрутизации. В противном случае будет произведена проверка переданного маршрута, и при необходимости его значение будет обновлено для последующего использования.

Функция `ip_output()` может быть вызвана и модулем транспортного протокола (UDP или TCP). Каким образом это происходит, описано в следующем разделе.

## Модуль UDP

Вернемся к рассмотрению ситуации, когда датаграмма адресована нашему хосту, не содержит ошибок (по крайней мере, с точки зрения IP) и должна быть передана транспортному протоколу. Поскольку целью данного раздела является иллюстрация схемы взаимодействия между модулями, рассмотрим более простой протокол UDP.

Итак, IP-модуль направляет датаграмму модулю UDP, вызывая функцию `udp_input()`, адрес которой был получен из соответствующего коммутатора протокола. Сначала функция `udp_input()` проверяет правильность контрольной суммы и допустимость установленных полей заголовка. Если указанные проверки закончились неудачно, пакет "молчаливо" уничтожается. Далее определяется получатель пакета. Для этого на основании адресов и номеров портов отправителя и получателя производится поиск соответствующего управляющего блока протокола<sup>24</sup>. В системе могут существовать несколько управляющих блоков с одинаковым номером локального порта, но с различными адресами и/или номерами портов отправителя. В этом случае выбирается блок, для которого найдено лучшее совпадение по всем четырем параметрам. Конечно, лучшим является точное совпадение, но если такого не найдено, будет выбран блок с совпадающим номером локального порта, но неуказанным адресом и/или номером порта отправителя. Таким образом, управляющий блок, у которого не указаны часть или все четыре параметра, является получателем всех пакетов, для которых не найдено лучшего совпадения<sup>25</sup>.

Если управляющий блок найден, данные и адрес отправителя помещаются в буфер приема сокета, связанного с управляющим блоком. В противном случае генерируется сообщение ICMP PORT UNREACHABLE.

<sup>24</sup>Функции `udp_input()` передается целиком датаграмма, включающая заголовок IP, заголовок UDP и данные протоколов верхнего уровня (приложений). Помимо того что эта информация необходима для определения адресата, по заголовку IP вычисляется контрольная сумма UDP. Такой подход гарантирует максимальную точность доставки данных нужному приложению.

<sup>25</sup>Возможность создания таких получателей "по умолчанию" используется в сетевом суперсервере `inetd`, который прослушивает все запросы и при необходимости запускает требуемый сервис (например FTP или Telnet). Это позволяет избежать запуска серверов без необходимости и тем самым сократить потребление ресурсов.

Передача данных от приложения инициируется системным вызовом `sendto(2)`, который на уровне сокета преобразуется в вызов функции `udp_usrreq()` с запросом `PRU_SEND`. Если передача инициирована системным вызовом `sendto()`, то вместе с данными передается адрес получателя. Если же данные были переданы с помощью системного вызова `send(2)`, то адрес получателя определяется из управляющего блока, где он был сохранен предшествующим вызовом `connect(2)`<sup>26</sup>.

Фактическая передача осуществляется функцией `udp_output()`, которая формирует заголовок пакета, устанавливает значения его полей и вычисляет контрольную сумму. После этого производится вызов уже рассмотренной ранее функции `ip_output()`.

## Модуль TCP

Как следует из предшествующего описания TCP, этот транспортный протокол обеспечивает гораздо более высокое качество передачи, чем UDP. Соответственно, его реализация также является гораздо более сложной. В предыдущих разделах уже встречались различные алгоритмы, используемые при реализации протокола. В этом разделе мы остановимся на одном важном механизме TCP — его таймерах.

Поскольку корректное функционирование протокола во многом зависит от порядка обмена управляющими сегментами, каждый канал обслуживается набором таймеров, позволяющих восстановить работу по тайм-ауту в случае потери управляющих пакетов. Эти таймеры хранятся в соответствующем управляющем блоке протокола TCP и, при их установке, обслуживаются<sup>27</sup> каждые 500 миллисекунд функцией `tcp_slowtimo()`.

Для обеспечения передачи данных используются два таймера. Первый из них — *таймер повторной передачи* (retransmit timer). Этот таймер запускается при передаче сегмента, если он уже не был запущен. Если подтверждение получено, и отсутствуют неподтвержденные данные — таймер останавливается. Если же такие данные существуют, значение таймера присваивается равным начальному, и таймер запускается снова. Если значение таймера становится равным нулю, наиболее старые неподтвержденные данные передаются повторно (как минимум один полный сегмент), а таймер запускается снова, но уже с большим значением. Скорость увеличения значения таймера (timer backoff) определяется по специальной таблице и имеет экспоненциальный характер.

<sup>26</sup> Протокол UDP не предусматривает предварительного установления связи с получателем данных. Поэтому, в отличие от TCP, вызов `connect(2)` не приводит к формированию управляющих сообщений и обмену ими между сторонами. В данном случае он служит лишь для сохранения адреса получателя в управляющем блоке.

<sup>27</sup> Обслуживание таймера заключается в уменьшении установленного значения и уведомлении модуля, когда значение таймера становится равным нулю.

Второй таймер — это *persist-таймер* (*таймер сохранения*). Этот таймер обеспечивает защиту от потери управляющих сообщений, содержащих обновленные значения окна. В случае, если отправитель готов передать данные, но анонсированное получателем окно слишком мало (равно нулю или меньше определенного значения), и отсутствуют неподтвержденные данные (т. е. таймер повторной передачи не включен), включается таймер сохранения. Если таймер срабатывает (его значение становится равным нулю), а обновленное значение так и не получено, отправитель передает максимально допустимый объем данных, определяемый текущим окном. Если же в этом случае значение текущего окна равно нулю (нулевое окно), то передается *пробный сегмент* (window probe), содержащий один октет данных, и таймер запускается снова. Если сообщение с обновленным значением окна было утеряно, или получатель по-прежнему отказывается изменить его размер, будет получено подтверждение, содержащее текущее значение окна. Такая ситуация, когда получатель не может принимать дополнительные данные, может продлиться достаточно долго. Например, пользователь может приостановить терминальный вывод и уйти на обед. В этом случае отправитель будет периодически посыпать пробные сегменты, а его окно будет по-прежнему закрыто.

Следующий таймер, который мы рассмотрим, — *keepalive-таймер*. Этот таймер предназначен для мониторинга каналов, по которым не передаются данные, и которые возможно в действительности прекратили свое существование, например, из-за аварийного останова одной из систем. Если за определенный промежуток времени данные по каналу переданы не были, модуль TCP отправляет пробный сегмент keepalive, ожидая в ответ либо подтверждения (это означает, что задержка в передаче данных временная), либо сообщения сброса канала (RST). Если получен сегмент RST, канал будет закрыт. Если после нескольких попыток, не будет получен отклик, канал будет уничтожен.

Последний таймер из рассматриваемых, это *2MSL-таймер* (2MSL — двойное максимальное время жизни сегмента в сети). Модуль TCP запускает этот таймер, когда производится завершение связи, и уже отправлено подтверждение полученному сегменту FIN. При этом отправитель не знает, получено ли его подтверждение. Поэтому он некоторое время ждет возможного повторного получения сегмента FIN, чтобы в свою очередь повторить подтверждение. Таймер запускается при переходе коммуникационного узла канала в состояние TIME-WAIT, и после его срабатывания соответствующий управляющий блок удаляется. Заметим, что это ожидание не блокирует процесс, выполнивший системный вызов *close(2)* сокета, отвечающего за данный канал. Другими словами, управляющий блок может существовать еще некоторое время после закрытия дескриптора сокета.

## Поддержка сети в UNIX System V

Многие из аспектов реализации поддержки сети в BSD UNIX справедливы и для архитектуры сетевых протоколов UNIX System V. Однако сам ме-

ханизм обеспечения взаимодействия модулей существенно отличается. Для поддержки сети в UNIX System V используется подсистема STREAMS, рассмотренная в главе 5.

Подсистема ввода/вывода, основанная на архитектуре STREAMS, позволяет в полной мере отразить уровневую структуру коммуникационных протоколов, когда каждый уровень имеет стандартные интерфейсы взаимодействия с другими (верхним и нижним) уровнями, и может работать независимо от конкретной реализации протоколов на соседних уровнях. Архитектура STREAMS полностью соответствует этой модели, позволяя создавать драйверы, которые являются объединениями независимых модулей.

Обмен данными между модулями STREAMS также соответствует характеру взаимодействия отдельных протоколов: данные передаются в виде сообщений, а каждый модуль выполняет требуемую их обработку. На рис. 6.30 приведена схема реализации протоколов TCP/IP в UNIX System V. Используя терминологию предыдущей главы, можно отметить, что модуль IP является гибридным мультиплексором, позволяя обслуживать несколько потоков, приходящих от драйверов сетевых адаптеров (в данном случае Ethernet и FDDI), и несколько потоков к модулям транспортных протоколов (TCP и UDP), а модули TCP и UDP — верхними мультиплексорами, обслуживающими прикладные программы, такие как сервер маршрутизации *routed(1M)*, сервер удаленного терминального доступа *telnetd(1M)*, сервер FTP *ftpd(1M)*, а также программы-клиенты пользователей (например *talk(1)*).

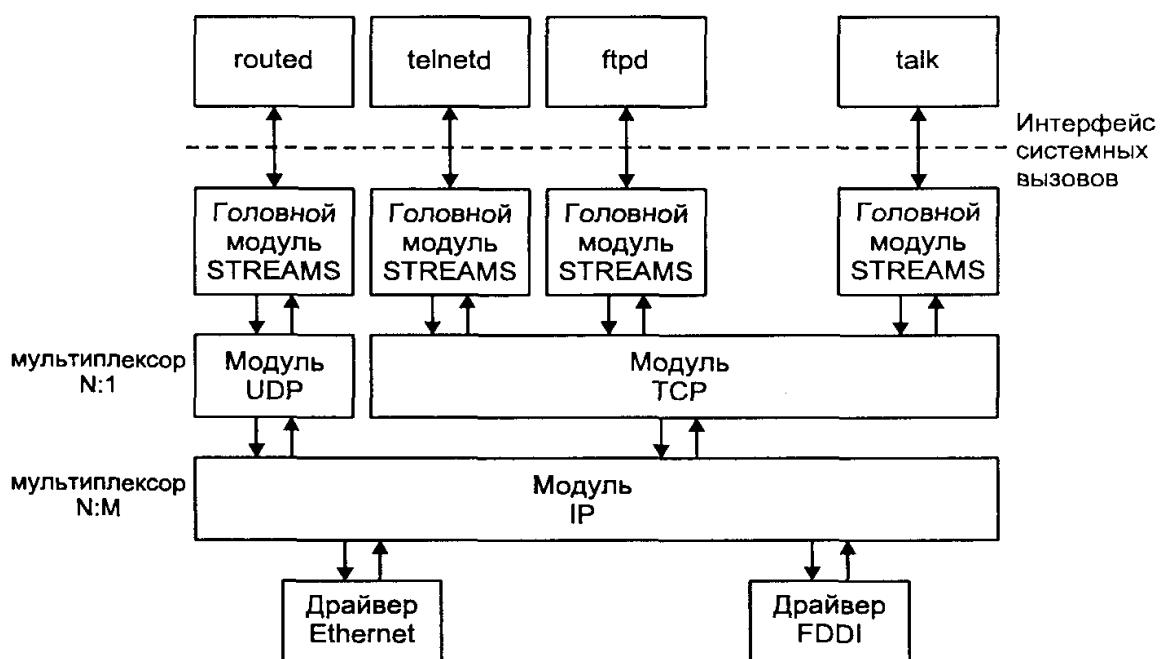


Рис. 6.30. Реализация протоколов TCP/IP на основе архитектуры STREAMS

Анализ программного обеспечения сетевой поддержки показывает, что как правило сетевые и транспортные протоколы, составляющие базовый стек TCP/IP, поставляются одним производителем, в то время как поддержка

уровней сетевого интерфейса и приложений может осуществляться продуктами различных разработчиков. Соответственно, можно выделить два основных интерфейса взаимодействия, стандартизация которых позволяет обеспечить совместную работу различных компонентов программного обеспечения. Первый интерфейс определяет взаимодействие транспортного уровня и уровня приложений и называется *интерфейсом поставщика транспортных услуг* (Transport Provider Interface, TPI). Второй интерфейс устанавливает правила и формат сообщений, передаваемых между сетевым уровнем и уровнем сетевого интерфейса, и называется *интерфейсом поставщика услуг канала данных* (Data Link Provider Interface, DLPI).

Вообще говоря, сетевая архитектура, основанная на архитектуре STREAMS, позволяет обеспечить поддержку любого стека протоколов, соответствующего модели OSI. Поэтому выражаясь более точно, перечисленные интерфейсы определяют взаимодействие транспортного уровня и уровня сеанса, и уровня канала и сетевого уровня, соответственно. Эти рассуждения проиллюстрированы на рис. 6.31<sup>28</sup>.

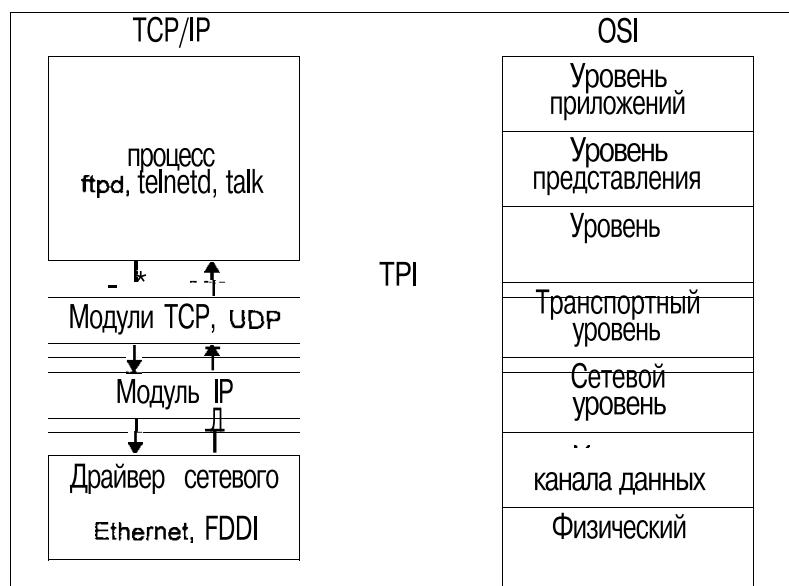


Рис. 6.31. Интерфейсы взаимодействия модулей протоколов

## Интерфейс TPI

TPI представляет собой интерфейс предоставления услуг транспортного уровня OSI модели как с предварительным установлением соединения (connection mode), так и без установления соединения (connectionless mode). Стандартизация этого интерфейса позволяет изолировать особенности реализации транспортного уровня от потребителя этих услуг и, тем самым, предоставить возможность разработки программного обеспечения, независимо от конкретного протокола и услуг им предоставляемых.

<sup>28</sup> Говоря еще более строго, данные интерфейсы определены самой моделью OSI. Однако в данной главе мы остановимся на практической реализации этих интерфейсов в подсистеме STREAMS.

**TPI** определяет набор и формат сообщений, с помощью которых протоколы верхнего уровня взаимодействуют с модулем транспортного протокола. Таким образом, TPI является интерфейсом между *поставщиком транспортных услуг* (transport provider) и *пользователем этих услуг* (transport user). Эти сообщения определяют *транспортные примитивы* (transport primitive), или команды, и могут иметь следующий формат:

- Сообщение состоит из блока типа `M_PROTO`, за которым может следовать несколько блоков `M_DATA`. Блок `M_PROTO` содержит управляющую информацию, включая тип команды и ее аргументы. В блоках `M_DATA` передаются ассоциированные с командой данные прикладной программы.
- Сообщение состоит из одного блока `M_PCPROTO`, который содержит управляющую информацию, включая тип команды и ее аргументы.
- Сообщение состоит из одного или более блоков `M_DATA`, в которых передаются данные прикладной программы.

**Таблица 6.10.** Основные управляющие сообщения TPI

Транспортный примитив	Тип сообщения	Значение								
<code>T_BIND_REQ</code>	<code>M_PROTO</code>	<p><b>Запрос на связывание.</b>            Этот примитив инициируется пользователем транспортных услуг и запрашивает связывание потока с адресом протокола. Сообщение состоит из одного блока <code>M_PROTO</code>, который содержит значение адреса и заказанное максимальное число запросов, ожидающих обслуживания со стороны пользователя. Последний параметр игнорируется для транспортных услуг без предварительного установления связи.</p> <p>Блок <code>M_PROTO</code> содержит следующие поля:</p> <table> <tr> <td><code>PRIM_type</code></td> <td>Тип примитива — <code>T_BIND_REQ</code></td> </tr> <tr> <td><code>ADDR_length</code></td> <td>Размер адреса протокола</td> </tr> <tr> <td><code>ADDR_offset</code></td> <td>Смещение адреса в блоке <code>M_PROTO</code></td> </tr> <tr> <td><code>CONIND_number</code></td> <td>Максимальное число запросов, ожидающих обслуживания</td> </tr> </table>	<code>PRIM_type</code>	Тип примитива — <code>T_BIND_REQ</code>	<code>ADDR_length</code>	Размер адреса протокола	<code>ADDR_offset</code>	Смещение адреса в блоке <code>M_PROTO</code>	<code>CONIND_number</code>	Максимальное число запросов, ожидающих обслуживания
<code>PRIM_type</code>	Тип примитива — <code>T_BIND_REQ</code>									
<code>ADDR_length</code>	Размер адреса протокола									
<code>ADDR_offset</code>	Смещение адреса в блоке <code>M_PROTO</code>									
<code>CONIND_number</code>	Максимальное число запросов, ожидающих обслуживания									
<code>T_BIND_ACK</code>	<code>M_PCPROTO</code>	<p>Подтверждение получения запроса на связывание.</p> <p>Этот примитив отправляется пользователю транспортных услуг и означает, что поток был связан с адресом протокола, заказанное максимальное число ожидающих запросов допустимо и поток был активизирован. Сообщение состоит из одного блока <code>M_PCPROTO</code>, содержащего значения указанных параметров. Заметим, что возвращаемый адрес может не совпадать с адресом, указанным в запросе <code>T_BIND_REQ</code>.</p> <p>Блок <code>M_PROTO</code> содержит следующие поля:</p> <table> <tr> <td><code>PRIM_type</code></td> <td>Тип примитива — <code>T_BIND_ACK</code></td> </tr> <tr> <td><code>ADDR_length</code></td> <td>Размер адреса протокола</td> </tr> <tr> <td><code>ADDR_offset</code></td> <td>Смещение адреса в блоке <code>M_PROTO</code></td> </tr> <tr> <td><code>CONIND_number</code></td> <td>Максимальное число запросов, ожидающих обслуживания</td> </tr> </table>	<code>PRIM_type</code>	Тип примитива — <code>T_BIND_ACK</code>	<code>ADDR_length</code>	Размер адреса протокола	<code>ADDR_offset</code>	Смещение адреса в блоке <code>M_PROTO</code>	<code>CONIND_number</code>	Максимальное число запросов, ожидающих обслуживания
<code>PRIM_type</code>	Тип примитива — <code>T_BIND_ACK</code>									
<code>ADDR_length</code>	Размер адреса протокола									
<code>ADDR_offset</code>	Смещение адреса в блоке <code>M_PROTO</code>									
<code>CONIND_number</code>	Максимальное число запросов, ожидающих обслуживания									

Таблица 6.10 (продолжение)

Транспортный примитив	Тип сообщения	Значение
T UNBIND REQ	M_PROTO	<p>Запрос на уничтожение связывания.</p> <p>Этот примитив инициируется пользователем транспортных услуг и запрашивает у поставщика уничтожение ранее созданного связывания потока с адресом протокола и деактивизацию потока.</p>
T CONN REQ	M_PROTO	<p>Запрос на установление связи.</p> <p>Этот примитив применим только для транспортных услуг с предварительным установлением связи. Он инициируется пользователем транспортных услуг и запрашивает установление связи с указанным адресатом. Сообщение состоит из одного блока M_PROTO, за которым может следовать один или несколько блоков типа M_DATA, содержащих прикладные данные, определенные пользователем. Заметим, что протокол TCP не позволяет передавать прикладные данные вместе с запросом. Блок M_PROTO содержит значение адреса получателя и опции, связанные с этим примитивом.</p> <p>Блок M_PROTO содержит следующие поля:</p> <ul style="list-style-type: none"> <li>PRIM_type Тип примитива — t_CONN_REQ</li> <li>DEST_length Размер адреса протокола</li> <li>DEST_offset Смещение адреса получателя в блоке M_PROTO</li> <li>OPT_length Размер опций</li> <li>OPT_offset Смещение опций в блоке M_PROTO</li> </ul>
T CONN IND	M_PROTO	<p>Индикация установления связи.</p> <p>Этот примитив применим только для транспортных услуг с предварительным установлением связи и свидетельствует о том, что удаленным пользователем с указанным адресом был сделан запрос на установление связи. Сообщение состоит из одного блока M_PROTO, за которым может следовать один или несколько блоков типа M_DATA, содержащих прикладные данные, определенные пользователем. Блок M_PROTO содержит значение адреса удаленного пользователя, отправившего запрос на установление связи, а также опции, связанные с этим примитивом.</p> <p>Блок M_PROTO содержит следующие поля:</p> <ul style="list-style-type: none"> <li>PRIM_type Тип примитива — t_CONN_IND</li> <li>SRC_length Размер адреса протокола</li> <li>SRC_offset Смещение адреса отправителя в блоке M_PROTO</li> <li>OPT_length Размер опций</li> <li>OPT_offset Смещение опций в блоке M_PROTO</li> <li>SEQ_number Идентификатор соединения</li> </ul>

Таблица 6.10 (продолжение)

Транспортный примитив	Тип сообщения	Значение
T_CONN_RES	M_PROTO	<p>Ответ на запрос на установление связи.</p> <p>Этот примитив применим только для транспортных услуг с предварительным установлением связи и свидетельствует о том, что поставщик транспортных услуг принимает предшествующий запрос на установление связи. Сообщение состоит из одного блока M_PROTO, за которым может следовать один или несколько блоков типа M_DATA, содержащих прикладные данные, определенные пользователем. Блок M_PROTO содержит указатель на очередь чтения потока, который будет обрабатывать запрос.</p> <p>Блок M_PROTO содержит следующие поля:</p> <ul style="list-style-type: none"> <li>PRIM_type Тип примитива — T_CONN_RES</li> <li>QUEUE_ptr Указатель на очередь потока, который должен быть использован в качестве узла созданного соединения</li> <li>OPT_length Размер опций</li> <li>OPT_offset Смещение опций в блоке M_PROTO</li> <li>SEQ_number Идентификатор соединения</li> </ul>
T_CONN_CON	M_PROTO	<p>Подтверждение установления связи.</p> <p>Этот примитив применим только для транспортных услуг с предварительным установлением связи. Он отправляется пользователю транспортных услуг в качестве подтверждения установления связи с удаленным пользователем. Сообщение состоит из одного блока M_PROTO, за которым может следовать один или несколько блоков типа M_DATA, содержащих прикладные данные, определенные пользователем. Блок M_PROTO содержит значение размера адреса, сам адрес удаленного пользователя, обслуживающего соединение, а также опции, связанные с этим примитивом.</p> <p>Блок M_PROTO содержит следующие поля:</p> <ul style="list-style-type: none"> <li>PRIM_type Тип примитива — T_CONN_CON</li> <li>RES_length Размер адреса протокола</li> <li>RES_offset Смещение адреса удаленного узла в блоке M_PROTO</li> <li>OPT_length Размер опций</li> <li>OPT_offset Смещение опций в блоке M_PROTO</li> </ul>

Таблица 6.10 (продолжение)

Транспортный примитив	Тип со-общения	Значение
T DISCON REQ	M_PROTO	<p>Запрос на разрыв связи.</p> <p>Этот примитив применим только для транспортных услуг с предварительным установлением связи. Он инициируется пользователем транспортных услуг и свидетельствует либо об отказе пользователем в установлении связи, либо о желании пользователя разорвать уже существующее соединение для данного потока. Сообщение состоит из одного блока M_PROTO, за которым может следовать один или несколько блоков типа M_DATA, содержащих прикладные данные, определенные пользователем.</p> <p>Блок M_PROTO содержит следующие поля:</p> <p>PRIM_type Тип примитива — T_DISCON_REQ SEQ_number Идентификатор соединения</p>
T DISCON IND	M_PROTO	<p>Индикация разрыва связи.</p> <p>Этот примитив применим только для транспортных услуг с предварительным установлением связи и свидетельствует о том, что удаленный пользователь либо отказывает в установлении связи, либо желает разорвать существующее соединение. Сообщение состоит из одного блока M_PROTO, за которым может следовать один или несколько блоков типа M_DATA, содержащих прикладные данные, определенные пользователем.</p> <p>Блок M_PROTO содержит следующие поля:</p> <p>PRIM_type Тип примитива — T_DISCON_IND DISCON_reason Причина разрыва связи SEQ_number Идентификатор соединения</p>
T ORDREL REQ	M_PROTO	<p>Запрос на "аккуратное" прекращение связи.</p> <p>Этот примитив применим только для транспортных услуг с предварительным установлением связи и указывает поставщику транспортных услуг, что пользователь завершил передачу данных. При этом соединение переходит в симплексный режим, позволяя пользователю принимать данные от удаленного узла. Сообщение состоит из одного блока M_PROTO.</p>
T ORDREL IND	M_PROTO	<p>Индикация "аккуратного" прекращения связи.</p> <p>Этот примитив применим только для транспортных услуг с предварительным установлением связи и отправляется пользователю транспортных услуг, свидетельствуя о том, что удаленный пользователь соединения завершил передачу данных. При этом соединение переходит в симплексный режим, позволяя пользователю передавать данные удаленному узлу. Сообщение состоит из одного блока M_PROTO.</p>

Таблица 6.10 (продолжение)

Транспортный примитив	Тип со-общения	Значение
T UNITDATA REQ	M_PROTO	<p>Запрос на передачу данных.</p> <p>Этот примитив применим только для транспортных услуг без предварительного установления связи и отправляется пользователем транспортных услуг в качестве запроса на передачу датаграммы. Сообщение состоит из одного блока M_PROTO, за которым может следовать один или несколько блоков типа M_DATA, содержащих прикладные данные пользователя. Блок M_PROTO содержит значение размера адреса и сам адрес получателя датаграммы, а также опции, связанные с этим примитивом.</p> <p>Блок M_PROTO содержит следующие поля:</p> <ul style="list-style-type: none"> <li>PRIM_type Тип примитива — T_UNITDATA_REQ</li> <li>DEST_length Размер адреса протокола</li> <li>DEST_offset Смещение адреса получателя в блоке M_PROTO</li> <li>OPT_length Размер опций</li> <li>OPT_offset Смещение опций в блоке M_PROTO</li> </ul>
T UNITDATA IND	M_PROTO	<p>Индикация получения данных.</p> <p>Этот примитив применим только для транспортных услуг без предварительного установления связи и указывает пользователю, что поставщиком транспортных услуг получена датаграмма от удаленного узла. Сообщение состоит из одного блока M_PROTO, за которым может следовать один или несколько блоков типа M_DATA, содержащих прикладные данные пользователя. Блок M_PROTO содержит значение адреса отправителя датаграммы, а также опции, связанные с этим примитивом.</p> <p>Блок M_PROTO содержит следующие поля:</p> <ul style="list-style-type: none"> <li>PRIM_type Тип примитива — T_UNITDATA_IND</li> <li>SRC_length Размер адреса протокола</li> <li>SRC_offset Смещение адреса отправителя в блоке M_PROTO</li> <li>OPT_length Размер опций</li> <li>OPT_offset Смещение опций в блоке M_PROTO</li> </ul>

Таблица 6.10 (продолжение)

Транспортный примитив	Тип со-общения	Значение
T_UDERROR_IND	M_PROTO	<p>Сообщение об ошибке датаграммы.</p> <p>Этот примитив применим только для транспортных услуг без предварительного установления связи и указывает пользователю, что датаграмма с указанным адресом получателя и опциями вызвала ошибку. Сообщение состоит из одного блока M_PROTO, содержащего размер адреса и сам адрес получателя, опции, а также код ошибки, зависящий от конкретного транспортного протокола.</p> <p>Блок M_PROTO содержит следующие поля:</p> <ul style="list-style-type: none"> <li>PRIM_type Тип примитива — T_UDERROR_IND</li> <li>DEST_length Размер адреса протокола</li> <li>DEST_offset Смещение адреса отправителя в блоке M_PROTO</li> <li>OPT_length Размер опций</li> <li>OPT_offset Смещение опций в блоке M_PROTO</li> <li>ERROR_type Код ошибки</li> </ul>
T_DATA_REQ	M_PROTO	<p>Запрос на передачу данных.</p> <p>Этот примитив применим только для транспортных услуг без предварительного установления связи и информирует поставщика транспортных услуг, что сообщение содержит пакет данных интерфейса (Transport Interface Data Unit, TIDU). Одно или более таких сообщений формируют пакет данных протокола TSDU. Сообщение состоит из одного блока M_PROTO, за которым может следовать один или несколько блоков типа M_DATA, содержащих прикладные данные пользователя. Блок M_PROTO содержит флаг MORE_flag, указывающий, является ли следующее сообщение T_DATA_REQ частью того же TSDU. На основании этого флага поставщик транспортных услуг компонует транспортные пакеты TSDU. Передача данных с помощью запросов T_DATA_REQ позволяет сохранить границы записи при передаче. Заметим, что протоколом TCP данная возможность не поддерживается.</p>
T_DATA_IND	M_PROTO	<p>Индикация получения данных.</p> <p>Этот примитив применим только для транспортных услуг без предварительного установления связи и информирует пользователя, что сообщение содержит пакет данных интерфейса TIDU. Сообщение состоит из одного блока M_PROTO, за которым может следовать один или несколько блоков типа M_DATA, содержащих прикладные данные удаленного пользователя. Блок M_PROTO содержит флаг MORE_flag, позволяющий пользователю определить границы TSDU.</p>

Таблица 6.10 (продолжение)

Транспортный примитив	Тип сообщения	Значение								
T EXDATA REQ	M_PROTO	<p>Запрос на передачу экстренных данных.</p> <p>Этот примитив аналогичен T_DATA_REQ, но служит для передачи экстренных данных. Протокол TCP поддерживает передачу экстренных данных с помощью функции <i>t_snd(3N)</i> с аргументом <i>flags</i>, содержащим флаг T_EXPEDITED и, возможно, T_MORE.</p>								
T EXDATA IND	M_PROTO	<p>Индикация получения экстренных данных.</p> <p>Этот примитив аналогичен T_DATA_IND, но служит для передачи пользователю экстренных данных.</p>								
T_OK_ACK	M_PCPROTO	<p>Положительное подтверждение.</p> <p>Этот примитив сообщает пользователю транспортных услуг, что предшествующий примитив, инициированный им, был успешно принят поставщиком транспортных услуг. В то же время, получение подтверждения не означает, что поставщиком были совершены какие-либо действия, связанные с предыдущим примитивом. Сообщение состоит из одного блока M_PCPROTO в котором хранится тип подтвержденного примитива CORRECT <i>prim</i>.</p>								
T_ERROR ACK	M_PCPROTO	<p>Сообщение об ошибке.</p> <p>Этот примитив сообщает пользователю услуг, что последний примитив, инициированный им, вызвал ошибку. Получение этого примитива может рассматриваться как отрицательное подтверждение, свидетельствующее, что никаких действий, связанных с ошибочным примитивом, не было предпринято. Сообщение состоит из одного блока M_PCPROTO, содержащего тип примитива, вызвавшего ошибку, код TLI и код системной ошибки UNIX.</p> <p>Блок M_PCPROTO содержит следующие поля:</p> <table> <tr> <td>PRIM_type</td> <td>Тип примитива — T_ERROR_ACK</td> </tr> <tr> <td>ERROR_prim</td> <td>Тип ошибочного примитива</td> </tr> <tr> <td>TLI_error</td> <td>Код ошибки TLI</td> </tr> <tr> <td>UNIX_error</td> <td>Код системной ошибки UNIX</td> </tr> </table>	PRIM_type	Тип примитива — T_ERROR_ACK	ERROR_prim	Тип ошибочного примитива	TLI_error	Код ошибки TLI	UNIX_error	Код системной ошибки UNIX
PRIM_type	Тип примитива — T_ERROR_ACK									
ERROR_prim	Тип ошибочного примитива									
TLI_error	Код ошибки TLI									
UNIX_error	Код системной ошибки UNIX									
T_INFO REQ	M_PCPROTO	<p>Запрос на получение параметров транспортного протокола.</p> <p>Этот примитив служит для запроса пользователем значений размеров различных параметров протокола, а также информации о текущем состоянии поставщика транспортных услуг. Сообщение состоит из одного блока M_PCPROTO.</p>								

Таблица 6.10 (продолжение)

Транспортный примитив	Тип со-общения	Значение
T_INFO_ACK	M_PCPROTO	<p>Параметры транспортного протокола.</p> <p>Этот примитив служит для передачи пользователю ранее запрошенных с помощью T_INFO_REQ параметров транспортного протокола. Сообщение состоит из одного блока M_PCPROTO, содержащего информацию, часть из которой возвращается функцией <i>t_open(3N)</i>, рассмотренной в разделе "Программный интерфейс сокетов" ранее в этой главе.</p> <p>Блок M_PCPROTO состоит из следующих полей:</p> <ul style="list-style-type: none"> <li>PRIM_type Тип примитива — T_INFO_ACK</li> <li>TSDU_size Определяет максимальный размер пакета данных протокола TSDU</li> <li>ETSDU_size Определяет максимальный размер пакета экстренных данных протокола ETSDU</li> <li>CDATA_size Определяет максимальный объем данных, передаваемых при установлении связи. Соответствует полю connect структуры info функции <i>t_open(3N)</i></li> <li>DDATA_size Определяет максимальный объем данных, передаваемых при разрыве связи. Соответствует полю discon структуры info функции <i>t_open(3N)</i></li> <li>ADDR_size Определяет максимальный объем транспортного протокола. Соответствует полю addr структуры info функции <i>t_open(3N)</i></li> <li>OPT_size Определяет размер опций для данного протокола. Соответствует полю options структуры info функции <i>t_open(3N)</i></li> <li>TIDU_size Определяет размер пакета данных интерфейса TIDU</li> <li>SERV_type Определяет тип транспортных услуг, предоставляемых поставщиком</li> <li>Соответствует полю servtype структуры info функции <i>t_open(3N)</i></li> <li>CURRENT_state Определяет текущее состояние поставщика транспортных услуг</li> <li>PROVIDER_flag Определяет дополнительные характеристики поставщика транспортных услуг</li> </ul>

Таблица 6.10 (окончание)

Транспортный примитив	Тип со-общения	Значение								
Т ОРТМГМТ REQ	М PROTO	<p>Управление опциями протокола.</p> <p>Этот примитив позволяет пользователю получить или установить опции протокола. Сообщение состоит из одного блока М_PROTO, включающего следующие поля:</p> <table> <tr> <td>PRIM_type</td><td>Тип примитива — Т_ОРТМГМТ_РЕQ</td></tr> <tr> <td>OPT_length</td><td>Размер опций</td></tr> <tr> <td>OPT_offset</td><td>Смещение опций в блоке М_PROTO</td></tr> <tr> <td>MGMT_flags</td><td>Флаги, определяющие характер запроса пользователя: Т_NEGOTIATE — установить опции, указанные пользователем. В результате опции, установленные поставщиком, могут отличаться от заказанных; Т_CHECK — проверить, поддерживаются ли опции, указанные пользователем, поставщиком; Т_DEFAULT — возвратить значения опций протокола.</td></tr> </table>	PRIM_type	Тип примитива — Т_ОРТМГМТ_РЕQ	OPT_length	Размер опций	OPT_offset	Смещение опций в блоке М_PROTO	MGMT_flags	Флаги, определяющие характер запроса пользователя: Т_NEGOTIATE — установить опции, указанные пользователем. В результате опции, установленные поставщиком, могут отличаться от заказанных; Т_CHECK — проверить, поддерживаются ли опции, указанные пользователем, поставщиком; Т_DEFAULT — возвратить значения опций протокола.
PRIM_type	Тип примитива — Т_ОРТМГМТ_РЕQ									
OPT_length	Размер опций									
OPT_offset	Смещение опций в блоке М_PROTO									
MGMT_flags	Флаги, определяющие характер запроса пользователя: Т_NEGOTIATE — установить опции, указанные пользователем. В результате опции, установленные поставщиком, могут отличаться от заказанных; Т_CHECK — проверить, поддерживаются ли опции, указанные пользователем, поставщиком; Т_DEFAULT — возвратить значения опций протокола.									
Т ОРТМГМТ ACK	М PCPROTO	<p>Положительное подтверждение.</p> <p>Этот примитив подтверждает завершение операции с опциями протокола, заказанными пользователем. Сообщение состоит из одного блока М_PROTO, включающего те же поля, что и Т_ОРТМГМТ_РЕQ.</p>								

### Взаимодействие с прикладными процессами

Рассмотренный ранее программный интерфейс ТЫ полностью реализует функциональность ТPI. Легко заметить соответствие между отдельными функциями ТЫ и примитивами ТPI, приведенными в табл. 6.10. Схема вызова функций ТЫ и обмена соответствующими примитивами ТPI между клиентом и сервером для типичного TCP-сеанса приведена на рис. 6.32.

Программный интерфейс потоков был рассмотрен в главе 5 при обсуждении подсистемы STREAMS. Основными функциями, обеспечивающими передачу и получение сообщений, являются системные вызовы *putmsg(2)* и *getmsg(2)*. Таким образом, большинство функций ТЫ, составляющих программный интерфейс доступа прикладных процессов к транспортным протоколам, являются удобной оболочкой (реализованной в виде библиотеки, например, **libnsl.so**) более фундаментальным системным вызовам *putmsg(2)* и *getmsg(2)*.

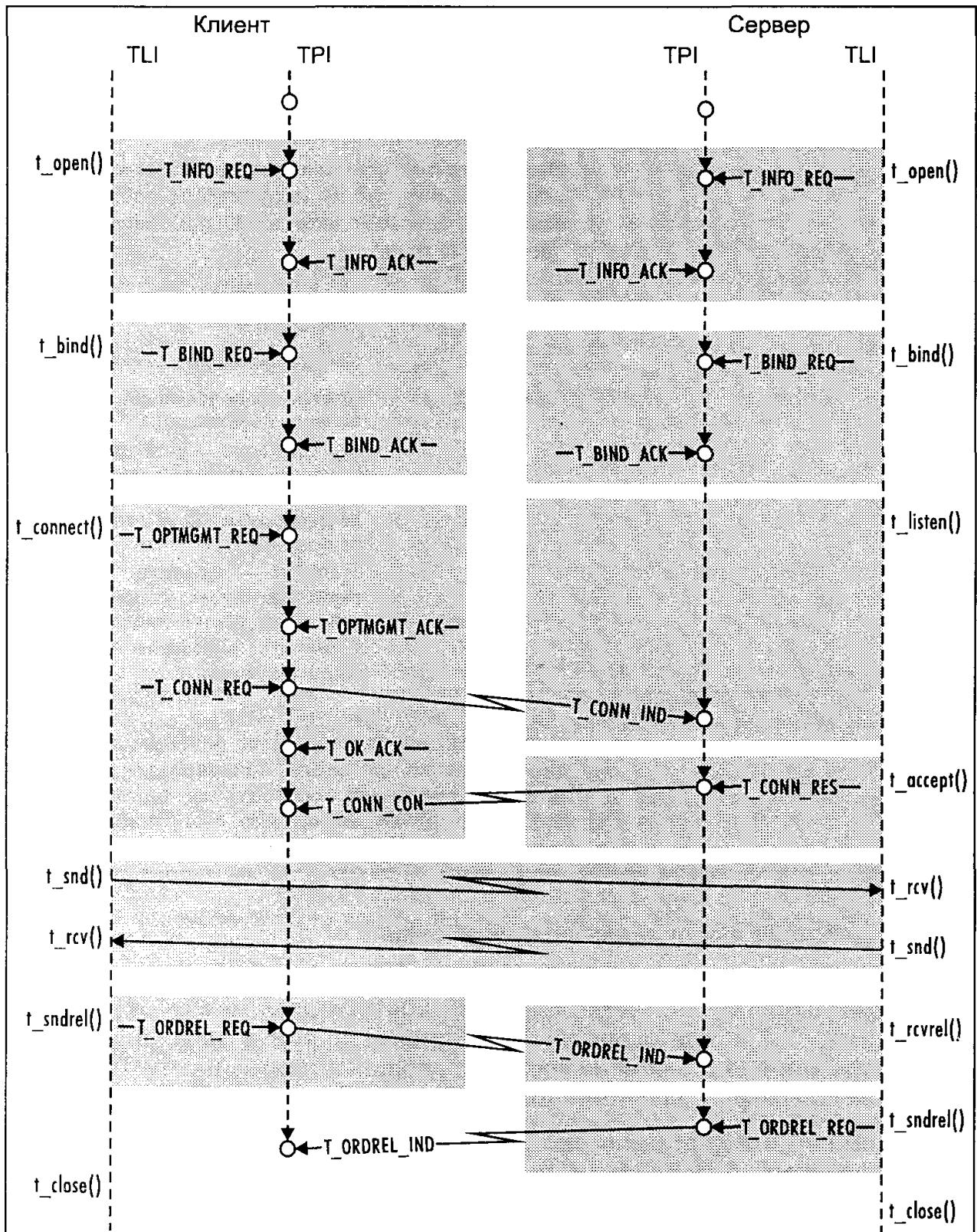


Рис. 6.32. Функции TLI и примитивы TPI

В качестве примера рассмотрим функцию *t\_connect(3N)*. Ее реализация может иметь следующий вид:

```
int t_connect(int fd, struct t_call *sndcall,
              struct t_call *recvcall)
{
    struct T_conn_req *connreq;
    struct T_conn_con *conncon;
    struct T_ok_ack *okack;
    struct T_error_ack *errack;
    struct strbuf connect, ack, confirm, m_data;
    struct netbuf      addr, opt, udata;
    char              *buf;
    int   flags;

/*Сохраним адреса буферов netbuf запроса sndcall*/
addr = sndcall->addr;
opt = sndcall->opt;
udata = sndcall->udata;

/*Заполним поля структуры strbuf для формирования управляющей части
(блок M_PROTO) сообщения T_CONN_REQ */
connect.len = sizeof(struct T_conn_req) + addr.len + opt.len;
connect maxlen = sizeof(struct T_conn_req) +
                  addr maxlen + opt maxlen;
buf = (char *)malloc(connect maxlen);
connect.buf = buf;

/*Заполним поля заголовка блока M_PROTO сообщения T_CONN_REQ в
соответствии с форматом структуры T_conn_req*/
connreq = (struct T_conn_req *)buf;
connreq->PRIM_type = T_CONN_REQ;
connreq->DEST_length = addr.len;
connreq->DEST_offset = sizeof (struct T_conn_req) ;
buf += sizeof(struct T_conn_req);
memcpy(buf, addr.buf, addr.len);
connreq->OPT_length = opt.len;
connreq->OPT_offset = connreq->DEST_offset + opt.len;
buf += addr.len;
memcpy(buf, opt.buf, opt.len);

/*Заполним поля структуры strbuf для формирования блока данных (блок
M_DATA)*/
m_data.len = udata.len;
m_data maxlen = udata maxlen;
m_data.buf = udata.buf;

/*Отправим запрос T_CONN_REQ поставщику транспортных услуг по потоку fd*/
putmsg(fd, &connect, &m_data, 0);
/*Подготовимся к приему подтверждения. Выделим максимальный размер для
получения негативного подтверждения, поскольку примитив T_ERROR_ACK
занимает больше места*/
ack.len = ack maxlen = sizeof(struct T_error_ack);
ack.buf = (char *)malloc(ack.len);
/*Подтверждение является приоритетным, поэтому установим флаг RS_HIPRI.
До получения подтверждения не предпринимаем никаких действий*/

```

```

flags = RS_HIPRI;
getmsg(fd, &ack, (struct strbuf *)0, &flags);
free(connect.buf);
okack = (struct T_ok_ack *)ack.buf;
/*Проверим получено ли положительное или негативное подтверждение*/
if (okack->PRIM_type == T_OK_ACK)
{
/*Если подтверждение положительное, подготовимся к получению согласия
удаленного пользователя на установление связи (примитив T_CONN_CON)*/
    free(ack.buf);
    if (recvcall != NULL)
    {
        addr = recvcall->addr;
        opt = recvcall->opt;
        udata = recvcall->udata;
        confirm.len = sizeof(struct T_conn_con) + addr.len +
                      opt.len;
        confirm maxlen = sizeof(struct T_conn_con)
                         + addr maxlen + opt maxlen;
        buf = (char *)malloc(confirm maxlen);
        confirm.buf = buf;
        m_data.len = udata.len;
        m_data maxlen = udata maxlen;
        m_data.buf = udata.buf;
/*Получим примитив T_CONN_CON*/
        getmsg(fd, &confirm, &m_data, &flags);
        free(buf);

        conncon = (struct T_conn_con *)confirm.buf;
        if (conncon->PRIM_type == T_CONN_CON)
        {
/*Если это действительно согласие, заполним структуру recvcall для
пользователя TLI*/
            addr.len = conncon->RES_length;
            opt.len = conncon->OPT_length;
            memcpy(addr.buf, conncon+conncon->RES_offset, addr.len);
            memcpy(opt.buf, conncon+conncon->OPT_offset, opt.len);
            free(confirm.buf);
/*Все закончилось удачно – возвращаем 0*/
            return(0);
        }
    }
else
{
/*В случае отказа мы готовы обработать примитив T_DISCON_IND*/
    return(-1);
}
else

```

```
/*Если получен примитив T_ERROR_ACK — обработаем его*/
errack = (struct T_error_ack *)ack.buf;

return(-1);
}
}
```

Подобным образом реализовано большинство функций ТЫ. Заметим, что в конкретном случае использования транспортного протокола TCP прием и передача данных осуществляются в виде потока, не содержащего каких-либо логических записей. В этом случае не требуется формирование примитивов типа T\_DATA\_REQ и T\_DATA\_IND. В то же время, для передачи и получения экстренных данных будут использованы примитивы T\_EXDATA\_REQ и T\_EXDATA\_IND. При использовании протокола UDP все данные будут передаваться С ПОМОЩЬЮ примитивов T\_UNITDATA\_REQ И T\_UNITDATA\_IND.

Описанная реализация программного интерфейса ТЫ имеет один существенный недостаток — операции функций не являются атомарными. Другими словами, выполнение функции *t\_connect(3N)* может быть прервано другими процессами, которые могут также связываться с удаленным узлом. Это возможно, поскольку выполнение значительной части операций происходит в режиме задачи. Если для функции *t\_connect(3N)* нарушение атомарности допустимо, то ряд функций, таких, например, как связывание (*t\_bind(3N)*), получение информации (*t\_open(3N)*, *t\_getinfo(3N)*) и установка или получение опций протокола (*t\_optmgmt(3N)*) должны быть защищены от возможного нарушения целостности данных по причине прерывания операции. Единственным способом гарантировать атомарность является перевод выполнения критических участков (например, между отправлением примитива и получением подтверждения от поставщика транспортных услуг) в режим ядра. Для этого подсистема STREAMS предлагает механизм обмена управляющими командами с помощью вызова *ioctl(2)*.

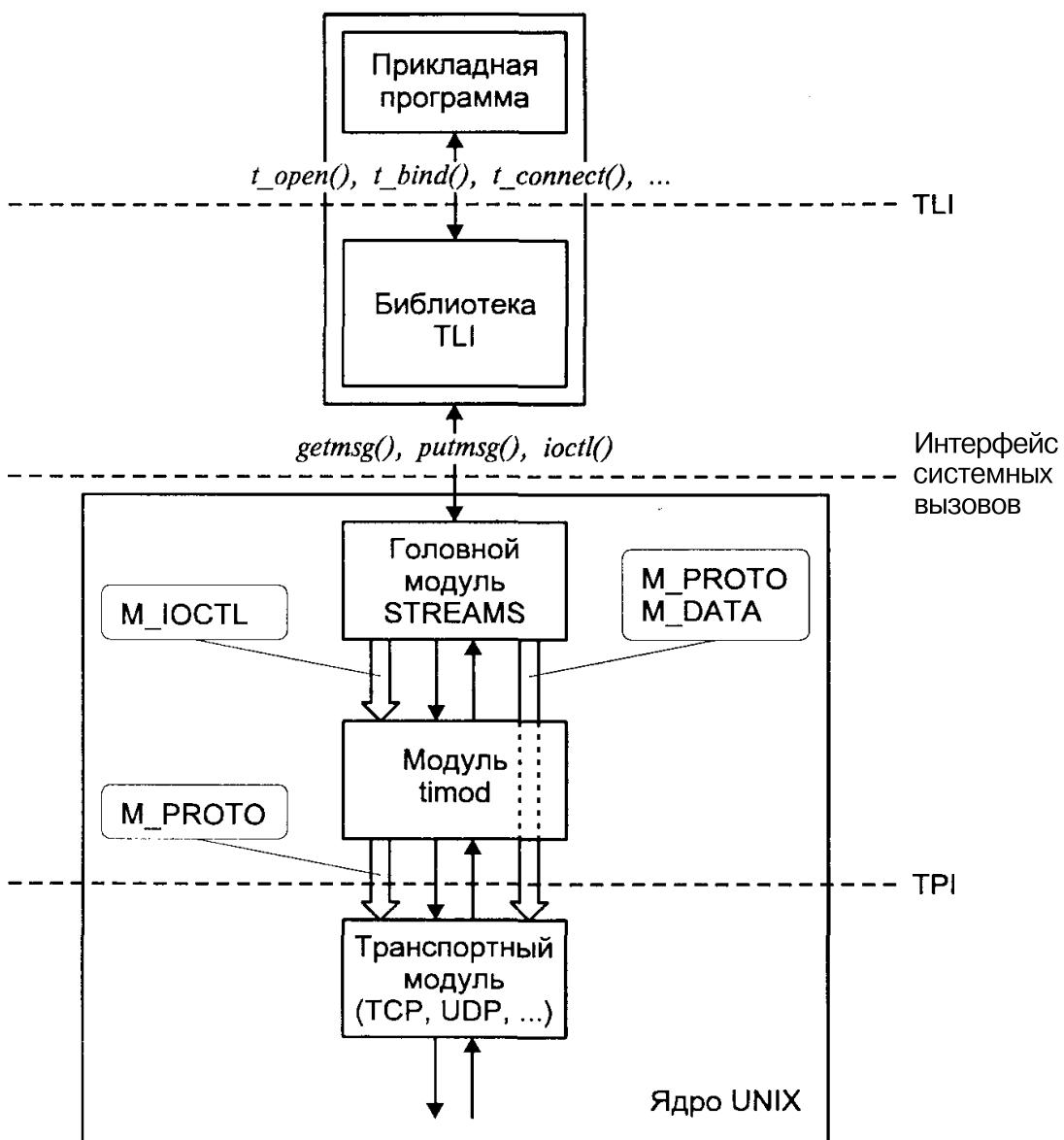
Однако с помощью *ioctl(2)*, как было показано в разделе "Подсистема STREAMS" главы 5, можно формировать лишь сообщения типа M\_IOCTL. Для преобразования этих сообщений в примитивы ТPI служит дополнительный модуль *timod(7M)*, встраиваемый в поток между головным и транспортным модулями. На рис. 6.33 показано местоположение модуля *timod(7M)* и схематически отображены его функции.

Для всех сообщений STREAMS, за исключением сообщений M\_IOCTL, которые генерируются головным модулем в ответ на системный вызов *ioctl(fd, I\_STR, ...)*, модуль *timod(7M)* является прозрачным, т. е. он просто передает эти сообщения следующему модулю вниз по потоку без какой-либо обработки. Несколько сообщений M\_IOCTL обрабатываются модулем и преобразуются в соответствующие примитивы ТPI.

При этом вызов *ioctl(2)* имеет следующий формат:

```
#include <sys/stropts.h>
struct strioctl my_strioctl;

    strioctl.ic_cmd = cmd;
    strioctl.ic_timeout = INFTIM;
    strioctl.ic_len = size;
    strioctl.ic_dp = (char *)buf
    ioctl(fd, I_STR, &my_strioctl);
```



**Рис. 6.33.** Архитектура доступа к транспортным услугам

При вызове *ioctl(2)* поле *size* устанавливается равным размеру соответствующего примитива TPI, определенного полем *cmd* и расположенного в буфере *buf*. При возврате из функции поле *size* содержит размер прими-

тива, возвращенного поставщиком транспортных услуг и расположенного в буфере `buf`.

Модуль `timod(7M)` служит для обработки следующих команд `cmd`:

<b>Значение cmd</b>	<b>Обработка модулем <code>timod(7M)</code></b>
<code>TI_BIND</code>	Команда преобразуется в примитив <code>T_BIND_REQ</code> . При успешном завершении функции <code>ioctl(2)</code> в буфере <code>buf</code> находится примитив <code>T_BIND_ACK</code> .
<code>TI_UNBIND</code>	Команда преобразуется в примитив <code>T_UNBIND_REQ</code> . При успешном завершении функции <code>ioctl(2)</code> в буфере <code>buf</code> находится примитив <code>T_OK_ACK</code> .
<code>TI_GETINFO</code>	Команда преобразуется в примитив <code>T_INFO_REQ</code> . При успешном завершении функции <code>ioctl(2)</code> в буфере <code>buf</code> находится примитив <code>T_INFO_ACK</code> .
<code>TI_OPTMGMT</code>	Команда преобразуется в примитив <code>T_OPTMGMT_REQ</code> . При успешном завершении функции <code>ioctl(2)</code> в буфере <code>buf</code> находится примитив <code>T_OPTMGMT_ACK</code> .

## Интерфейс DLPI

DLPI определяет интерфейс между протоколами уровня канала данных (data link layer) модели OSI, называемыми поставщиками услуг уровня канала данных и протоколами сетевого уровня, называемыми пользователями услуг уровня канала данных. В качестве примера пользователей услуг уровня канала данных можно привести такие протоколы, как IP, IPX или CLNS. С другой стороны, поставщик услуг уровня канала данных непосредственно взаимодействует с различными сетевыми устройствами, обеспечивающими передачу данных по сетям различной архитектуры (например, Ethernet, FDDI или ATM) и использующими различные физические среды передачи.

Для обеспечения независимости DLPI от конкретной физической сети передачи драйвер уровня канала данных состоит из двух частей: верхней аппаратно независимой и нижней аппаратно зависимой. Аппаратно независимая часть драйвера обеспечивает предоставление общих услуг, определенных интерфейсом DLPI, а также поддержку ряда потенциальных пользователей, представляющих семейства протоколов TCP/IP, NetWare и OSI. Аппаратно зависимая часть непосредственно взаимодействует с сетевым адаптером.

На рис. 6.34 приведена структура драйвера поставщика услуг уровня канала данных. Обмен данными между аппаратно независимой частью драйвера и пользователем услуг осуществляется в виде сообщений STREAMS, формат и назначение которых определяется спецификацией DLPI (т. н. примитивы DLPI).

Во время инициализации и последующей передачи данных аппаратно независимая часть драйвера вызывает необходимые функции аппаратно зависимой части. Напротив, при поступлении данных из сети, аппаратно зависимая часть помещает пакеты данных, или кадры, непосредственно в очередь чтения аппаратно независимой части. Обе части совместно используют набор переменных и флагов для взаимной синхронизации и контроля передачи.

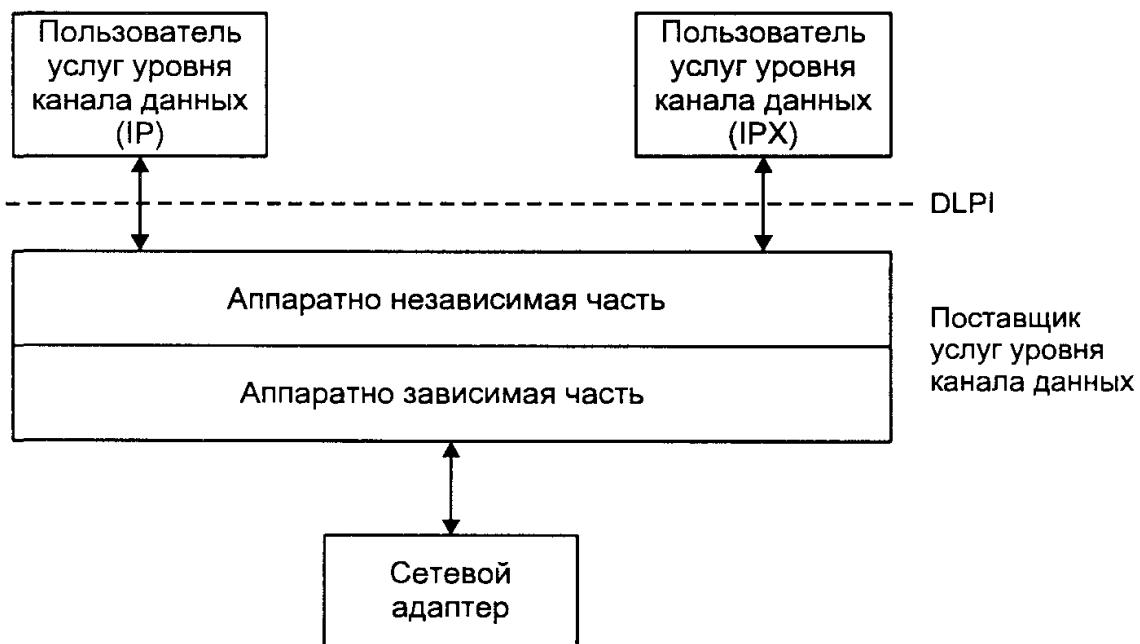


Рис. 6.34. Структура драйвера уровня канала данных

Пользователь получает доступ к услугам поставщика услуг уровня канала данных через *точку доступа к услугам* (Service Access Point, SAP), используя сообщения STREAMS для обмена данными. Поскольку один поставщик может иметь несколько пользователей, например IP и IPX, в его задачу входит маршрутизация данных, полученных от физической сети, к нескольким точкам доступа. Для этого каждый пользователь идентифицирует себя с помощью адреса SAP, который сообщает поставщику, используя примитив связывания (DL\_BIND\_REQ) потока с точкой доступа к услугам уровня канала данных.

Поскольку аппаратно зависимая часть драйвера может обслуживать несколько сетевых адаптеров, каждый сетевой интерфейс идентифицируется *точкой физического подключения* (Physical Point of Attachment, PPA). При этом спецификация DLPI определяет два типа поставщиков услуг. Поставщик услуг первого типа (style 1) производит назначение PPA, исходя из старшего и младшего номеров используемого специального файла устройства (указанного в вызове *open(2)*). Обычно каждый адаптер, обслуживаемый драйвером, ассоциирован со старшим номером, а младший номер используется для создания клонов (см. раздел "Клоны" главы 5). Напр-

тив, поставщик второго типа (style 2) позволяет пользователю явно указать PPA уже после открытия потока с помощью примитива присоединения (DL\_ATTACH\_REQ). Использование поставщиков второго типа является более предпочтительным, например, когда одна физическая сеть поддерживает создание независимых логических, или виртуальных каналов передачи данных (например, каналы ISDN B и D). В этом случае идентификатор PPA, передаваемый примитивом DL\_ATTACH\_REQ, содержит также идентификатор логического канала. Схема описанных точек доступа приведена на рис. 6.35.

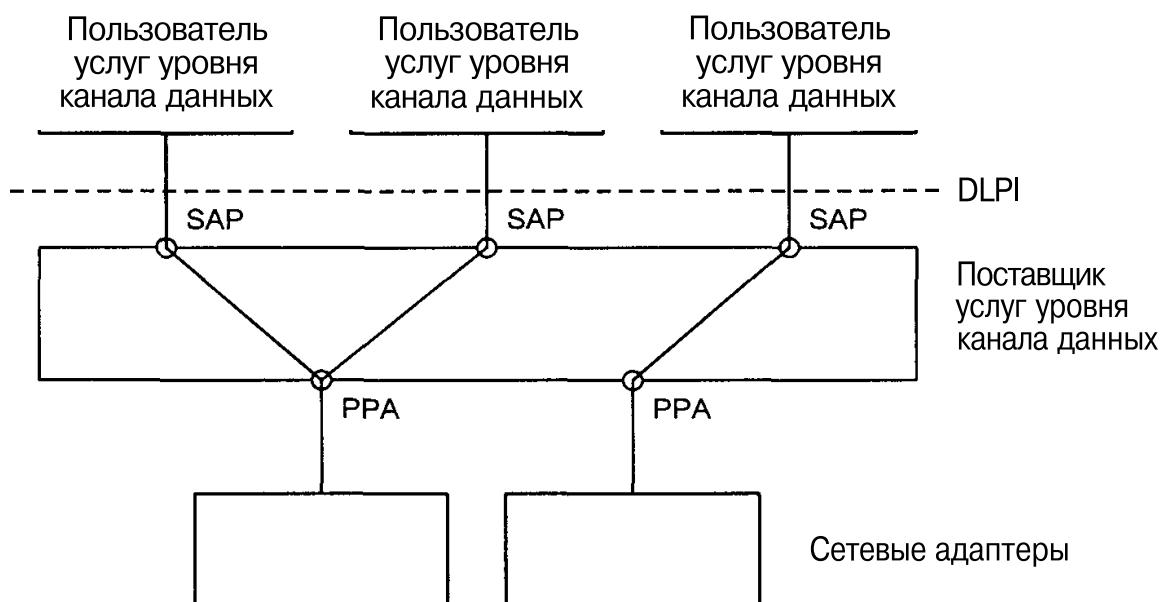


Рис. 6.35. Доступ к услугам поставщика услуг уровня канала данных

DLPI определяет три различных режима передачи данных (или типов услуг), позволяющих обеспечить различные требования протоколов верхнего уровня и поставщиков услуг уровня канала данных:

1. Режим с предварительным установлением связи
2. Режим без предварительного установления связи с подтверждением
3. Режим без предварительного установления связи без подтверждения

В данном разделе мы остановимся только на режиме без предварительного установления связи без подтверждения. Заметим, что для традиционных технологий локальных сетей используется именно этот тип услуг уровня канала данных.

Поскольку дальнейшее обсуждение будет касаться преимущественно коммуникационной инфраструктуры локальных сетей, кратко остановимся на логическом делении уровня канала данных модель OSI в соответствии со стандартом IEEE 802. Применяемые сегодня технологии локальных сетей существенно отличаются друг от друга, как по физической среде и топо-

логии, так и по способу передачи данных в этой физической среде и формату передаваемых данных. Поэтому стандарт IEEE 802 разделяет протоколы локальных сетей на два логических подуровня:

- Верхний независимый от среды передачи подуровень, названный *уровнем управления логическим каналом* (Logical Link Control, LLC), определенный стандартом IEEE 802.2.
- Нижний зависимый от среды передачи подуровень, названный *уровнем управления доступом к среде передачи* (Media Access Control, MAC), определенный стандартами IEEE 802.3 для протокола CSMA/CD, IEEE 802.4 для протокола Token Bus и IEEE 802.5 для Token Ring.

## Доступ к среде передачи

Общим в наиболее распространенных технологиях локальных сетей является то, что несколько сетевых устройств совместно используют одну и ту же среду передачи данных, и соответственно делят между собой полосу пропускания сети. Для корректного и эффективного использования сетевых ресурсов необходим механизм контроля доступа к физической среде передачи, который и обеспечивается протоколами уровня MAC.

Первым по известности в ряду этих протоколов стоит CSMA/CD (Carrier Sense Multiple Access with Collision Detection). При этом методе доступа сетевые устройства конкурируют между собой за право передачи по принципу "кто успел — тот и съел". Основной принцип заключается в том, что сетевое устройство может начать передачу данных, только если сеть свободна. Однако при этом возникают ситуации, называемые *коллизиями*, когда два сетевых устройства начинают передавать данные одновременно. Естественно в этом случае данные не могут быть использованы, и на время коллизии сеть становится недоступной. Время коллизии может быть сокращено, если передающее устройство продолжает "слушать" сеть. Можно сформулировать следующие правила работы CSMA/CD:

1. Если сеть свободна, сетевое устройство может начать передачу, в противном случае, устройство продолжает "слушать" сеть.
2. Если в процессе передачи устройством обнаружена коллизия, устройство должно передать короткий неформатированный сигнал, чтобы гарантировать, что остальными устройствами коллизия также обнаружена, после чего немедленно прекратить передачу.
3. После передачи неформатированного сигнала устройство ожидает случайный промежуток времени, после чего начинает передачу, если сеть свободна.

Передача данных в CSMA/CD осуществляется в виде пакетов, или кадров, для которых существуют два основных формата в соответствии со специ-

фикацией Ethernet 2.0 и стандартом IEEE 802.3. Последний был разработан на основе спецификации Ethernet, однако форматы кадров несколько отличаются, как это показано на рис. 6.36 и 6.37.

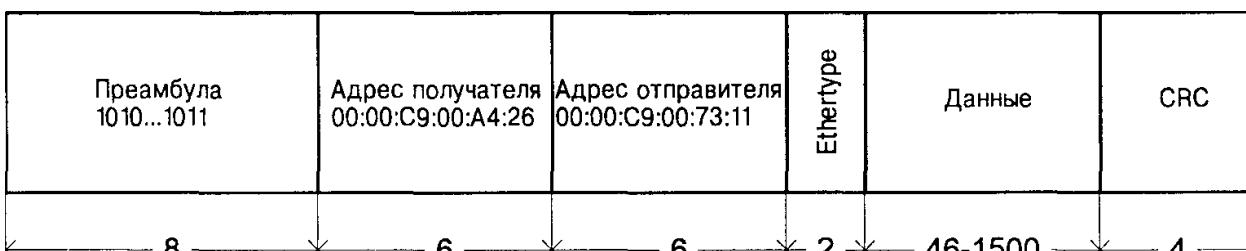


Рис. 6.36. Формат кадра Ethernet

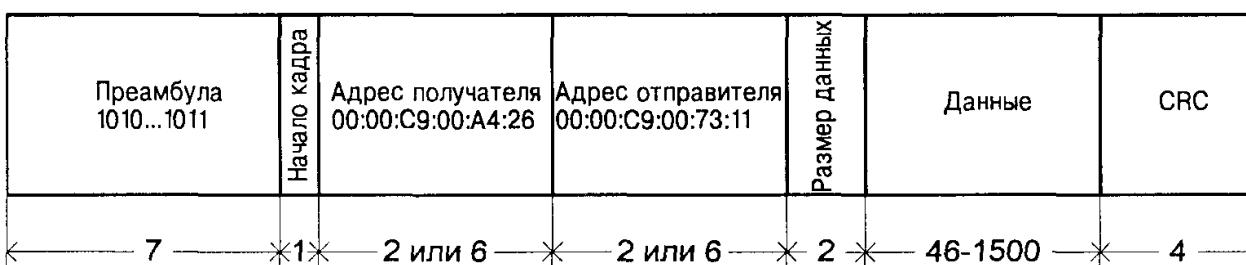


Рис. 6.37. Формат кадра IEEE 802.3

Существенным различием между двумя форматами является то, что поле "тип пакета" (EtherType) кадра Ethernet используется для обозначения размера кадра в случае IEEE 802.3. В кадре Ethernet это поле идентифицирует сетевой протокол, использующий данный кадр. К счастью, значения идентификаторов протоколов превышают 1500 — максимальный размер данных кадра, поэтому драйвер может легко определить используемый формат.

Другой, также часто используемый метод доступа, используемый в кольцевых топологиях сетей, заключается в передаче между сетевыми устройствами, подключенными к кольцу, маркера — небольшого пакета, играющего роль эстафетной палочки (например, в сетях Token Ring). Пока ни одно из устройств не передает данные, маркер, циркулирующий в кольце, имеет флаг "свободный". При необходимости передачи устройство дожидается свободного маркера, изменяет его флаг на "занятый" и передает пакет данных сразу же за маркером. Поскольку теперь в сети отсутствует свободный маркер, все остальные устройства должны воздержаться от передачи. При этом устройство, которому адресованы данные, при получении скопирует их в свой буфер. Занятый маркер совершает круг и возвращается к передавшему пакет устройству. Последнее извлекает из сети маркер и пакет данных, изменяет флаг маркера на "свободный" и вновь передает его в кольцо. Таким образом, ситуация возвращается к исходной.

Технология FDDI, также использует метод передачи маркера, правда, несколько отличающийся от только что описанного. Основное отличие заключается в том, что устройство сразу же после передачи пакета помещает

свободный маркер. Если какое-либо устройство желает передать данные, оно может воспользоваться этим маркером, также поместив новый свободный маркер вслед за переданным пакетом. Таким образом, в кольце может одновременно существовать несколько пакетов, что повышает эффективность использование пропускной способности сети.

Формат кадров в сетях Token Ring определяется двумя стандартами — IEEE 802.5 и FDDI. Однако за исключением октета контроля доступа эти форматы не отличаются друг от друга. Формат кадра IEEE 802.5 приведен на рис. 6.38.

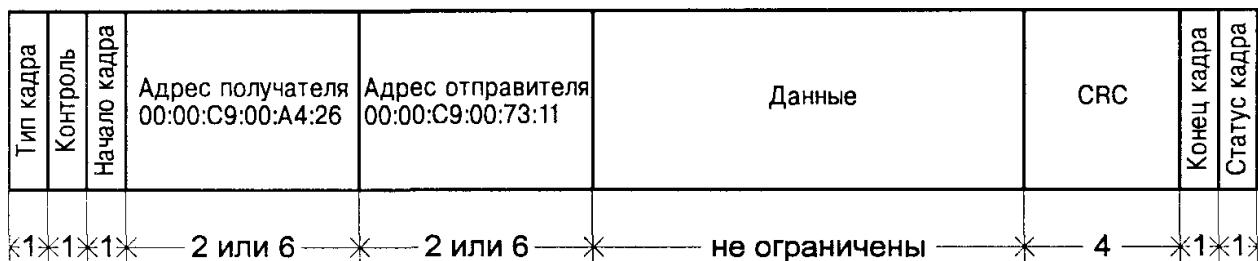


Рис. 6.38. Формат кадра IEEE 802.5

## Протокол LLC

Протокол LLC обеспечивает большую часть услуг уровня канала данных. Этот протокол был разработан на основе другого протокола уровня канала данных — HDLC, однако обладает меньшей функциональностью по сравнению со своим родителем.

Формат кадра LLC представлен на рис. 6.39. Основными полями заголовка кадра являются DSAP и SSAP, которые определяют адреса точек доступа (SAP) соответственно отправителя и получателя данных. Кадр LLC также может содержать дополнительный заголовок SNAP (Sub-Network Access Point), также называемый адресом логической точки доступа (Logical SAP, LSAP).

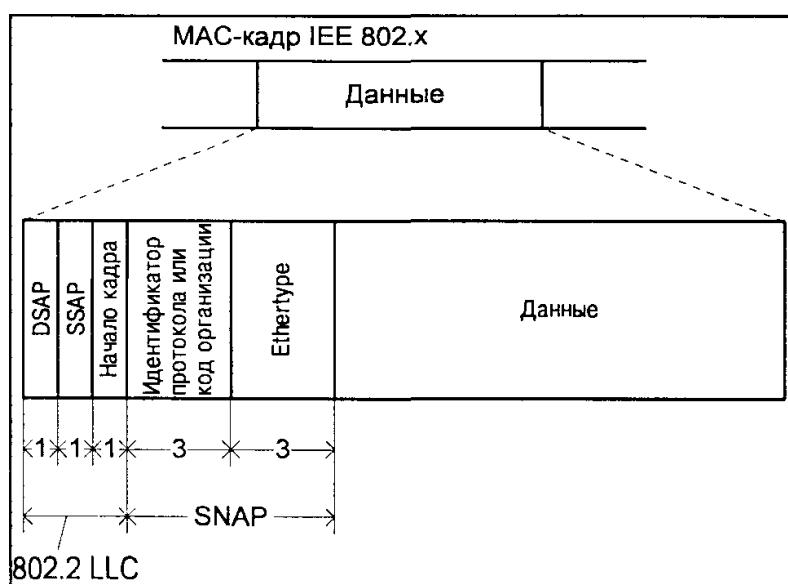


Рис. 6.39. Формат кадра LLC

## Инкапсуляция IP

При работе в локальной сети на базе технологии CSMA/CD возможны два варианта инкапсуляции датаграмм IP в кадры уровней LLC и MAC.

Первый заключается в использовании кадров Ethernet 2.0. В этом случае поле данных (1500 октетов) полностью принадлежит IP-датаграмме, а SAP адресуется полем "тип пакета", которое содержит значение параметра Ethertype — индекса протокола верхнего уровня. В случае IP это значение равно 0x0800. Значения Ethertype для других протоколов приведены в табл. 6.11.

**Таблица 6.11.** Значение Ethertype для некоторых протоколов

0x0000–0x05DC	Поле Length IEEE802.3
0x0800	Internet IP (Ipv4)
0x0806	ARP
0x6003	DEC DECNET Phase IV Route
0x8137	Novell IPX

Второй вариант предполагает использование формата IEEE 802.3. В этом случае IP-датаграмма инкапсулируется в кадр LLC, а адресация SAP осуществляется в заголовке SNAP с помощью идентификатора Ethertype. При этом поля DSAP и SSAP не используются, и их значения устанавливаются равными 0xAA. Заметим, что в этом случае максимальный размер IP-датаграммы составляет 1492 октета.

При передаче данных TCP/IP в сетях Token Ring используется формат кадра IEEE 802.5, инкапсулирующий кадр LLC с заголовком SNAP, как описано выше.

## Внутренняя архитектура

Как уже говорилось, драйвер, реализующий поставщика услуг уровня канала данных, состоит из двух частей: аппаратно зависимой и аппаратно независимой. Соответственно драйвер хранит отдельные структуры данных, необходимые для работы этих частей. Архитектура драйвера приведена на рис. 6.40.

Для каждого обслуживаемого драйвером сетевого адаптера создается отдельная структура данных `DL_bdconfig_t`, описывающая характеристики адаптера и содержащая необходимую для управления адаптером информацию, а также статистику, являющуюся частью MIB (Management Information Base). Эта структура используется аппаратно независимой и зависимой частями совместно, в том числе и для передачи определенной информации между ними.

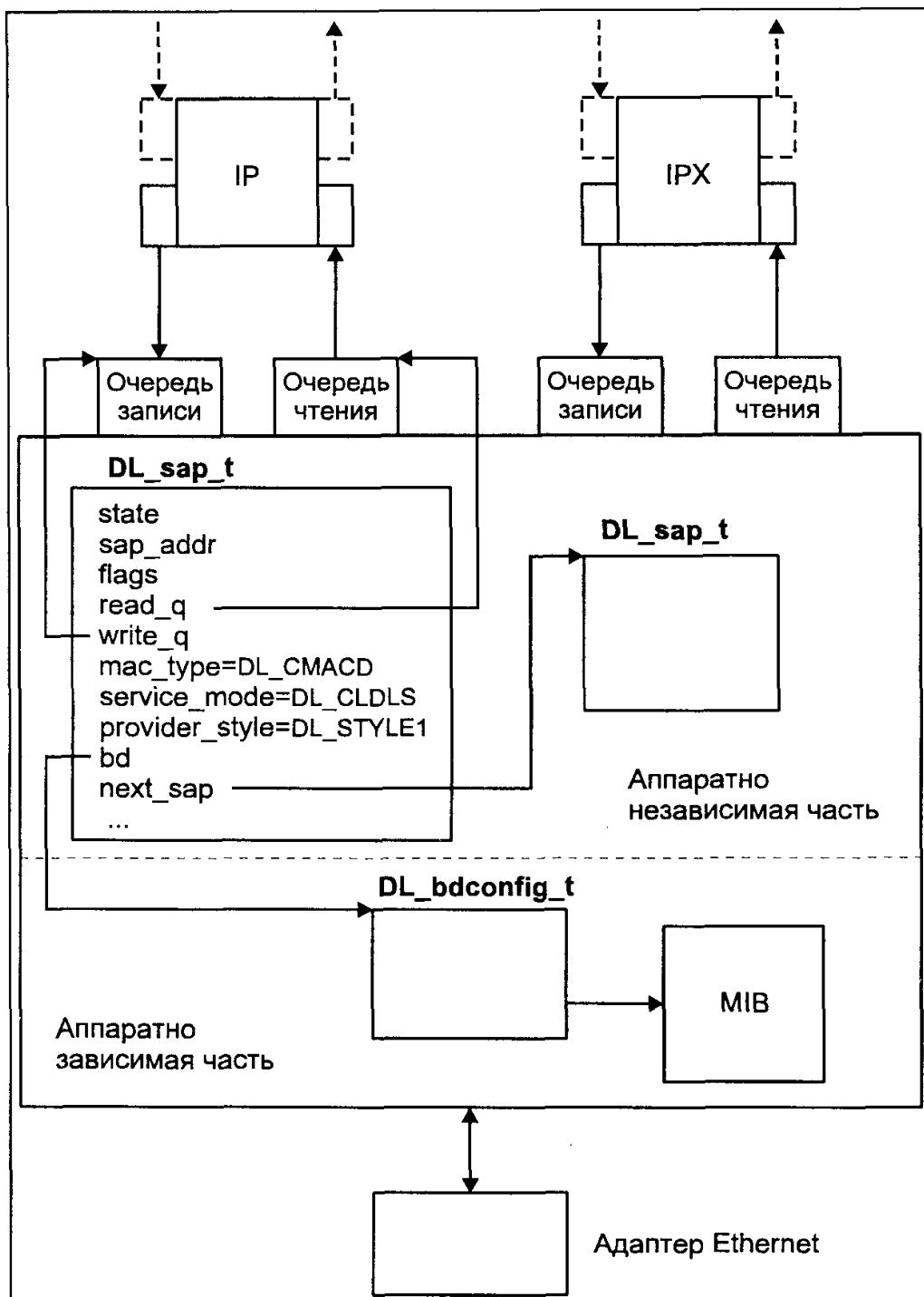


Рис. 6.40. Архитектура драйвера DLPI

В частности, эта структура содержит следующие поля:

major	Старший номер устройства, связанного с данным сетевым адаптером
io_start	Адрес начала области ввода/вывода
io_end	Адрес конца области ввода/вывода
mem_start	Адрес начала базовой памяти
mem_end	Адрес конца базовой памяти

irq_level	Уровень прерывания
max_saps	Максимальное число точек доступа (SAP)
flags	Флаги состояния адаптера
mib	Список статистических данных

Поле `flags` может включать следующие флаги:

BOARD_PRESENT	Устанавливается драйвером после успешной инициализации адаптера
BOARD DISABLED	Устанавливается драйвером при неудачной инициализации адаптера. Этот флаг также может быть установлен, если драйвер определит нарушения в функционировании адаптера
TX_BUSY	Указывает на отсутствие ресурсов, например отсутствие необходимых буферов для передачи кадра
TX_QUEUED	Указывает на наличие кадров, ожидающих передачи

Для каждого подключенного пользователя услуг, или, другими словами, для каждой активной SAP драйвер создает структуру данных `DL_sap_t`, описывающую тип и характеристики точки доступа. Приведем описание некоторых полей этой структуры:

state	Состояние SAP. Возможные состояния определены интерфейсом DLPI. Исходное состояние точки доступа <code>DL_UNBOUND</code>
sap_addr	Уникальный адрес (идентификатор) SAP
flags	Флаги, определяющие дополнительные характеристики SAP
read_q	Указатель на очередь чтения потока, связанного с SAP
write_q	Указатель на очередь записи потока, связанного с SAP
mac_type	Тип используемого протокола доступа и формат используемого кадра. Возможные значения включают: <code>DL_CMACD</code> IEEE 802.3 <code>DL_ETHER</code> Ethernet 2.0 <code>DL_TPB</code> IEEE 802.4 <code>DL_TPR</code> IEEE 802.5 <code>DL_HDLC</code> ISO HDLC <code>DL_FDDI</code> FDDI
service_mode	Режим передачи данных. В локальных сетях обычно используется режим без установления связи без подтверждения DL CLDLS
provider_style	Тип поставщика услуг: <code>DL_STYLE1</code> или <code>DL_STYLE2</code>
bd	Указатель на структуру <code>DL_bdconfig_t</code> , связанную с сетевым адаптером
next_sap	Указатель на следующую точку доступа в списке активных SAP
max_spdu	Максимальный размер данных, которые могут быть переданы в кадре
min_spdu	Минимальный размер данных, которые могут быть переданы в кадре

Дополнительные характеристики SAP хранятся в поле `flags`, которое может включать следующие флаги:

<code>RAWCSMACD</code>	Указывает, что через SAP передаются только кадры формата IEEE 802.3
<code>SNAPCSMACD</code>	Указывает, что через SAP передаются кадры формата LLC SNAP
<code>PROMISCUOUS</code>	Указывает, что SAP работает в режиме <i>отсутствия фильтрации кадров</i> ( <i>promiscuous mode</i> ), при котором SAP получает копии всех кадров независимо от адреса точки доступа, которой они предназначаются. Данный режим применяется, например, при создании приложений мониторинга уровня канала данных
<code>PRIVILEGED</code>	Указывает, что управление точкой доступа требует привилегий суперпользователя

Кроме того, драйвер хранит и обновляет статистическую информацию о сетевом интерфейсе, представляющую собой набор счетчиков, связанных с работой адаптера, и ассоциированных с ним точек доступа. Пользователь может получить интересующую его статистику с помощью соответствующей команды `ioctl(2)`. Приведем в качестве примера описание некоторых из этих счетчиков:

<code>ifInOctets</code>	Общее число октетов, полученных адаптером
<code>ifOutOctets</code>	Общее число октетов, переданных адаптером
<code>ifOutUcastPkts</code>	Число переданных однокомандных (unicast) пакетов
<code>ifOutNUcastPkts</code>	Число переданных групповых (multicast) и широковещательных (broadcast) пакетов
<code>ifInDiscards</code>	Число полученных, но отброшенных правильных пакетов
<code>ifInUcastPkts</code>	Число полученных однокомандных (unicast) пакетов
<code>ifInNUcastPkts</code>	Число полученных групповых (multicast) и широковещательных (broadcast) пакетов
<code>ifInErrors</code>	Число пакетов, полученных с ошибкой
<code>ifUnknownProtos</code>	Число полученных пакетов, которые были отброшены из-за неправильной SAP адресата
<code>ifOutQlen</code>	Число пакетов, находящихся в очереди на передачу
<code>ifOutErrors</code>	Число пакетов, переданных с ошибкой
<code>etherCollisions</code>	Число коллизий

Аппаратно независимая часть драйвера обрабатывает все запросы, поступающие от пользователя услуг уровня канала данных. Для этого в драйвере определены следующие функции (часть из них являются стандартными точками входа STREAMS):

<code>DLopen()</code>	Точка входа <code>xxopen()</code> . Эта функция инициализирует SAP, связанную с данным потоком. Функция проверяет наличие флага <code>BOARD_PRESENT</code> и в случае его отсутствия возвращает ошибку.
<code>DLclose()</code>	Точка входа <code>xxclose()</code> . Эта функция сбрасывает текущее состояние SAP и устанавливает его равным <code>DL_UNBOUND</code> .

(продолжение)

- DLwput () Точка входа `xxput()` для очереди записи. Эта функция интерпретирует примитивы DLPI и вызывает соответствующие процедуры драйвера. В случае, если примитив содержит команду уровня канала данных, например, запрос на передачу датаграммы, вызывается функция `DLcmds()`, которая производит формирование кадра и вызов функции передачи кадра аппаратно зависимой части драйвера. В случае, когда примитив содержит команду `ioctl(2)`, вызывается функция `Dlioctl()`.
- DLrsrv () Точка входа `xxservice()` для очереди чтения. Функция `DLrecv()` помещает каждый кадр, полученный от аппаратно зависимой части драйвера, в очередь чтения потока, ассоциированного с адресуемой SAP. В зависимости от формата кадра (протокола MAC) вызывается соответствующая процедура, извлекающая данные и помещающая их в сообщение `DL_UNITDATA_IND` (для услуги без предварительного установления связи и без подтверждения), которое направляется вверх по потоку пользователю услуг. Кроме того, `DLrsrv()` просматривает список активных SAP для возможного копирования сообщения в очередь потоков, имеющих тот же адрес точки доступа. Поскольку функция `DLrecv()` помещает кадр в очередь первого найденного потока с требуемым адресом SAP (см. описание функции ниже), описанное поведение `DLrsrv()` гарантирует, что все пользователи услуг уровня канала данных, зарегистрировавшие один и тот же адрес SAP, получат свою копию пакета данных.
- DLrecv () Функция обработки полученного пакета. Эта функция определяет формат пакета и помещает его в очередь потока, ассоциированную с адресуемой SAP. Обычно эта функция вызывается функцией обработки прерывания при получении очередного кадра данных от сетевого адаптера.

## Примитивы DLPI

Как и в случае предоставления транспортных услуг, обмен данными между пользователем и поставщиком происходит в виде сообщений, несущих примитивы DLPI. Ниже рассмотрены некоторые из этих примитивов, относящиеся к режиму передачи без предварительного установления связи и без подтверждения. Именно такой режим обычно используется в традиционных локальных сетях.

Несмотря на то что рассматриваемая услуга не предусматривает установления связи, фактической передаче данных предшествует обмен примитивами для инициализации потока и подключения его к поставщику услуг уровня канала данных. Во-первых, пользователь должен создать точку доступа к поставщику услуг, для чего необходимо произвести операцию связывания. Во-вторых, в случае использования поставщика услуг второго типа (style 2), пользователь также должен подключиться к требуемой РРА. Наконец, пользователю может потребоваться произвести ряд действий, включающих получение информации о созданном потоке, регистрацию

специфического группового адреса для потока или включение режима отсутствия фильтрации кадров, при котором пользователь сможет получать копии всех пакетов, полученных поставщиком услуг<sup>29</sup>.

После этого пользователь может передавать данные, учитывая, однако, что в обсуждаемом режиме поставщик не гарантирует надежную доставку данных адресату (удаленному пользователю услуг уровня канала данных). Например, отсутствие управления передачей может привести к переполнению буферов, и, как следствие, к потере кадров. Неправильные кадры, полученные из сети, также будут отбрасываться без уведомления передающей стороны. Однако преимуществом является отсутствие необходимости установления связи и связанных с этим накладных расходов.

Итак, приведем некоторые управляющие сообщения **DLPI**, используемые в режиме без предварительного установления связи и без подтверждения. В табл. 6.12 приведено их краткое описание.

**Таблица 6.12. Примитивы DLPI**

<b>Примитив DLPI</b>	<b>Тип сообщения</b>	<b>Значение</b>
DL_BIND_REQ	M_PROTO	<p>Запрос на связывание.</p> <p>Этот примитив инициируется пользователем услуг и запрашивает связывание потока с точкой доступа и его активизацию. Следует иметь в виду, что активным считается поток, для которого поставщик услуг может передавать или принимать пакеты данных. Таким образом, PPA, ассоциированная с данным потоком, должна быть инициализирована до завершения обработки запроса на связывание (другими словами, поставщик гарантирует, что при получении пользователем подтверждения связывания DL_BIND_ACK инициализация PPA завершилась успешно). Сообщение состоит из одного блока M_PROTO, который содержит значение адреса SAP, тип услуги и ряд других параметров, обсуждение которых выходит за рамки данной книги.</p>
DL_BIND_ACK	M_PCPROTO	<p>Подтверждение получения запроса на связывание.</p> <p>Этот примитив отправляется пользователю услуг и означает, что поток был связан с адресом SAP и был активирован. Сообщение состоит из одного блока M_PCPROTO, в частности, содержащего значение адреса SAP.</p>

<sup>29</sup> Включение этого режима требует привилегий суперпользователя и используется преимущественно в приложениях мониторинга уровня канала данных.

Таблица 6.12 (продолжение)

Примитив DLPI	Тип сообщения	Значение
DL_ATTACH_REQ	M_PROTO	<p>Запрос на подключение к РРА.</p> <p>Этот примитив инициируется пользователем услуг уровня канала данных и запрашивает у поставщика ассоциализацию потока с указанной РРА. Этот запрос является необходимым для поставщика второго типа (style 2) для указания физической среды, по которой будут передаваться данные. Сообщение состоит из одного блока M_PROTO, в котором пользователь передает значение идентификатора РРА. Формат этого идентификатора определяется поставщиком. Пользователь должен указать, как минимум, физическую среду передачи. Для сетей, где несколько независимых каналов передачи мультиплексируются в одном физическом носителе, идентификатор также должен содержать информацию о конкретном канале передачи данных. Примером технологий, обеспечивающих такое мультиплексирование являются ISDN (каналы B и D) и ATM (коммутируемые и постоянные виртуальные каналы — SVC и PVC).</p>
DL_INFO_REQ	M_PCPROTO	<p>Запрос на получение параметров потока.</p> <p>Этот примитив служит для запроса пользователем значений размеров различных параметров потока, активизированного поставщиком DLPI, а также информации о текущем состоянии интерфейса. Сообщение состоит из одного блока M_PCPROTO.</p>
DL_INFO_ACK	M_PCPROTO	<p>Параметры транспортного протокола.</p> <p>Этот примитив служит для передачи пользователю ранее запрошенных с помощью DL_INFO_REQ параметров. Сообщение состоит из одного блока M_PCPROTO, содержащего информацию, часть из которой приведена ниже:</p> <p>d1_max_sdu — определяет максимальное число октетов данных пользователя, которое может быть передано в одном кадре. (Максимальный размер SDU поставщика услуг.)</p> <p>d1_min_sdu — определяет минимальный размер SDU.</p>

Таблица 6.12 (продолжение)

Примитив DLPI	Тип со-общения	Значение
DL INFO ACK	M_PCPROTO	<p><code>dl_addr_length</code> — определяет максимальную длину адреса DLSAP поставщика. Этот адрес, помимо адреса SAP может также включать физический адрес интерфейса и ряд других полей (иерархический адрес).</p> <p><code>dl_addr_offset</code> — указывает смещение адреса DLSAP в блоке M_PCPROTO.</p> <p><code>dl_mac_type</code> — указывает тип среды передачи, поддерживаемой потоком DLPI. См. значение поля <code>mac_type</code> структуры <code>DL_sap_t</code> ранее в этой главе.</p> <p><code>dl_current_state</code> — указывает текущее состояние потока.</p> <p><code>dl_service_mode</code> — определяет тип услуги, обеспечивающей потоком DLPI.</p> <p><code>dl_provider_style</code> — определяет тип поставщика услуг (style 1 или style 2).</p> <p><code>dl_brdcst_addr_length</code> — определяет размер физического широковещательного адреса.</p> <p><code>dl_brdcsr_addr_offset</code> — указывает смещение значения адреса DLSAP в блоке M_PCPROTO.</p>
DL UNITDATA REQ M_PROTO		<p>Запрос на передачу данных.</p> <p>Этот примитив применим только для услуг уровня канала данных без предварительного установления связи и отправляется пользователем услуг в качестве запроса на передачу кадра. Сообщение состоит из одного блока M_PROTO, за которым может следовать один или несколько блоков типа M_DATA, содержащих данные пользователя. Блок M_PROTO содержит значения размера адресов и сам адрес получателя кадра, а также приоритет из диапазона, определенного поставщиком.</p>
DL UNITDATA IND M_PROTO		<p>Индикация получения данных.</p> <p>Этот примитив применим только для услуг уровня канала данных без предварительного установления связи и указывает пользователю, что поставщиком услуг получен кадр от удаленного узла. Сообщение состоит из одного блока M_PROTO, за которым может следовать один или несколько блоков типа M_DATA, содержащих данные пользователя. Блок M_PROTO содержит значения адресов отправителя и получателя кадра.</p>

Таблица 6.12 (окончание)

Примитив DLPI	Тип сообщения	Значение
DL_OK_ACK	M_PCPROTO	Положительное подтверждение. Этот примитив сообщает пользователю услуг уровня канала данных, что предшествующий примитив, инициированный им, был успешно принят поставщиком услуг. Примитив DL_OK_ACK передается только для примитивов, нуждающихся в подтверждении.
DL_ERROR_ACK	M_PCPROTO	Сообщение об ошибке. Этот примитив сообщает пользователю услуг, что последний примитив, инициированный им, вызвал ошибку. Получение этого примитива может рассматриваться как отрицательное подтверждение, свидетельствующее, что никаких действий, связанных с ошибочным примитивом, не было предпринято. Сообщение состоит из одного блока M_PCPROTO, содержащего тип примитива, вызвавшего ошибку, код DLPI и, если возможно, код системной ошибки UNIX.
DL_UDERROR_IND	M_PROTO	Сообщение об ошибке кадра. Этот примитив применим только для услуг уровня канала данных без предварительного установления связи и указывает пользователю, что его запрос на передачу DL_UNITDATA_REQ вызвал ошибку и не может быть выполнен. Сообщение состоит из одного блока M_PROTO, содержащего размер адреса и сам адрес получателя, а также код ошибки.

## Заключение

В этой главе описана организация сетевой поддержки UNIX. Рассмотрение не выходило за рамки обсуждения семейства протоколов TCP/IP, хотя архитектура сетевого доступа операционной системы позволяет обеспечить поддержку практически любых протоколов. В этом отношении большей гибкостью обладает сетевая подсистема UNIX System V, основанная на архитектуре STREAMS.

Хотя стандартная спецификация протоколов гарантирует совместимость между системами различных разработчиков и производителей, на эффективность и производительность сетевой подсистемы оказывает существенное влияние конкретная реализация алгоритмов. Этот аспект особенно

актуален для протокола транспортного уровня — TCP. Безусловно, работа сетевой подсистемы также существенным образом зависит от оптимальной настройки, но этот вопрос, к сожалению, находится за пределами этой книги. Однако сегодня уже недостаточно просто связи с удаленным хостом, и материал этой главы может помочь обеспечить требуемое качество этой связи.

В главе также описан программный интерфейс сетевого доступа. В частности, был рассмотрен пример использования сокетов для межпроцессного взаимодействия не только в рамках одного компьютера, но и в распределенной сетевой инфраструктуре.

Во второй части главы была описана внутренняя архитектура сетевых подсистем в BSD UNIX и UNIX System V. Хотя эти вопросы наиболее интересны разработчикам драйверов и других подсистем ядра, более пристальный взгляд на взаимодействие компонентов операционной системы может помочь и администраторам в решении их проблем, и пользователям в оценке качества работы их систем для уверенного обсуждения этой темы с системным администратором.

## Приложение А

### Электронный справочник *man(1)*

Многообразие команд и утилит UNIX, обилие ключей или опций к ним, особенности применения и формат тех или иных системных вызовов и функций могут озадачить неискушенного пользователя. К счастью, в UNIX имеется электронный справочник, позволяющий быстро получить исчерпывающую информацию по интересующей команде или функции, формату файла или типам данных. Воспользоваться этим справочником так же просто, как и любой другой утилитой UNIX. Для этого в командной строке нужно ввести *man* с названием команды или функции, о которой вы хотели бы получить информацию. Например, введя \$ *man man*, вы узнаете как пользоваться справочником.

Весь справочный материал разбит на разделы, порядок и названия которых различны для разных версий операционной системы. В таблице, приведенной ниже, перечислены традиционные разделы и их названия (чаще всего, это просто номер) для двух основных ветвей UNIX: BSD и System V.

<b>Содержимое раздела</b>	<b>BSD</b>	<b>UNIX</b>	<b>UNIX System V</b>
Прикладные утилиты	1		1
Системные вызовы	2		2
<u>Библиотечные функции</u>	3		3
Специальные файлы, драйверы устройств и аппаратное обеспечение	4		7
Форматы различных конфигурационных и системных файлов	5		4
Всякая всячина, например, типы файловых систем, определение типов данных и т.д.	7		5
<u>Административные утилиты</u>	8		1M

Некоторые разделы справочника могут иметь подсекции, содержимое которых можно уточнить, прочитав введение к соответствующему разделу. Например, в операционной системе Solaris 2.x для вывода содержания раздела "Библиотечные функции" необходимо задать следующую команду:

```
$ man -z3 intro
```

Явное указание раздела понадобится и в том случае, когда по заданному ключевому слову имеются статьи в различных разделах. Например, для слова *passwd* имеются статьи в разделе 1 (команда *passwd(1)* для изменения пароля пользователя) и в разделе 4 (формат файла паролей *passwd(4)*). Просматривая статью, обязательно прочитайте абзац с названием "SEE ALSO" в котором приведены названия статей справочника, имеющие отношение к интересующей вас тематике.

## **Приложение Б**

### **Дополнительная информация об операционной системе UNIX**

#### **Книги**

**Б. Кернigan, Р. Пайк. UNIX-универсальная среда программирования.** //Пер. с англ. М.: Финансы и статистика, 1992. Прекрасная книга, написанная людьми, принимавшими непосредственное участие в создании этой операционной системы. Книга окажется интересной как для пользователей, так и для программистов и администраторов системы. Несмотря на относительно небольшой объем, книга позволяет достаточно основательно изучить эту операционную систему.

**Bach M. The Design of the UNIX Operating System.** Englewood Cliffs, NJ: Prentice-Hall, 1986. Долгое время эта книга являлась практически единственным полным описанием внутренней архитектуры UNIX. Хотя материал, представленный в книге, основан на системе UNIX System V Release 2, большинство положений остаются справедливыми и сегодня.

**М. Банахан, Э. Раттер. Введение в операционную систему UNIX.** //Пер. с англ. М.: Радио и связь, 1986. Одна из немногих книг по UNIX на русском языке. Изданная на английском языке в 1982 году, книга в значительной степени устарела. В основном предназначена тем, кто приступает к использованию UNIX.

**Р. Готье. Руководство по операционной системе UNIX.** //Пер. с англ. М.: Финансы и статистика, 1985. Книга во многом копирует электронный справочник man, хотя и содержит ряд любопытных примеров по использованию тех или иных утилит. В книге также содержится ряд практических рекомендаций для системного администратора. Несомненным достоинством является то, что книга написана на русском языке.

**Leffler S., McKusick M.K., Karels M.J., Quarterman J.S. The Design and Implementation of the 4.3BSD UNIX Operating System.** Reading, MA: Addison-Wesley, 1989. Книга написана группой разработчиков этой версии операционной системы. В книге детально обсуждается внутренняя архитектура ядра и принцип работы системы 4.3BSD UNIX. Прекрасная возможность получить информацию о UNIX "из первых уст".

**Vahalia, U. UNIX Internals: the New Frontiers.** UpperSaddle River, NJ: Prentice Hall, 1996. В книге сравниваются принципы организации и функционирования нескольких современных версий UNIX (SVR4.X, Solaris, Digital UNIX, 4.4BSD, Mach и OSF/1). В книге нашли свое отражение последние достижения в разработке операционных систем семейства UNIX.

**Pate S.D. UNIX Internals. A Practical Approach.** Addison Wesley Longman Ltd., 1996. Книга написана сотрудником компании Santa Cruz Operation, Inc. и посвящена архитектуре ядра операционной системы UNIX. Хотя весь материал основан на версии UNIX SCO OpenServer, большая часть положений справедлива и для других современных систем. Большое количество иллюстраций и практических примеров позволяет проникнуть в тайны ядра UNIX.

**Stevens, W.R. Advanced Programming in the UNIX Environment.** Reading, MA: Addison-Wesley, 1992. Наиболее полное описание программного интерфейса операционной системы UNIX. Книга содержит много примеров и служит прекрасным пособием разработчикам программного обеспечения для этой операционной системы.

**Stevens, W.R. UNIX Network Programming.** Englewood Cliffs, NJ: Prentice Hall, 1990. Прекрасное руководство для программистов в области сетевых технологий. Многочисленные примеры позволяют проверить положения книги на практике.

## Информация в Internet

**Официальный сервер UNIX (<http://www.rdg.opengroup.org/unix/>).** В этом разделе сервера группы The Open Group, собственностью которой является зарегистрированный знак UNIX, вы можете ознакомиться с различными спецификациями, имеющими отношение к этому семейству операционных систем. В частности, здесь вам предложат программное обеспечение **CI Report**, позволяющее проверить насколько тексты написанной вами программы удовлетворяют стандартам и требованиям переносимости.

**Компания Santa Cruz Operation, Inc. (<http://www.sco.com/>).** На сервере этой фирмы-производителя коммерческих версий операционной системы UNIX — SCO OpenServer SCO и UnixWare, вы можете ознакомиться с каталогом предлагаемого программного обеспечения, новинками и предложениями от SCO. Здесь же вы узнаете, как получить бесплатную версию систем OpenServer и UnixWare для некоммерческого использования. Обширный раздел сервера посвящен разработчикам программного обеспечения.

**Сервер компании Silicon Graphics, Inc. (<http://www.sgi.com/>).** Если вы счастливый обладатель рабочей станции, сервера или суперкомпьютера фир-

мы Silicon Graphics, вы наверняка частый гость на этом сервере. Здесь представлен материал о собственной версии операционной системы UNIX, получившей название IRIX. Сегодня эта операционная система работает на компьютерах Silicon Graphics и считается одной из самых мощных.

**Информационный центр DIGITAL UNIX (<http://www.unix.digital.com/>).** Этот сервер компании Digital Equipment Corporation посвящен операционной системе DIGITAL UNIX, предназначеннй для рабочих станций и серверов на базе процессоров Alpha. Здесь вы ознакомитесь с новейшими достижениями компании, найдете полезный материал по операционной системе DIGITAL UNIX, а также сможете скопировать ряд программных продуктов.

**Программное обеспечение для серверов RS/6000 (<http://www.rs6000.ibm.com/software/>).** Крупнейший производитель компьютеров и программного обеспечения к ним, фирма IBM на этом сервере представляет собственную версию операционной системы UNIX — AIX 4.x и разнообразные приложения, разработанные для нее.

**Сервер компании Berkeley Software Design, Inc. (<http://www.bsdi.com/>).** Эта компания является поставщиком операционной системы BSDI, изначально разработанной в Калифорнийском университете Беркли. На этом сервере вам предложат новинки программного обеспечения, часть из которых можно скопировать на собственный сервер. Здесь вы имеете возможность лучше ознакомиться с этой версией UNIX, по праву являющейся одним из лидеров в области сетевых операционных систем.

**FreeBSD (<http://www.freebsd.org/>).** На этом сервере вы найдете ответы на все вопросы, связанные с операционной системой FreeBSD. Эта система разработана и поддерживается большой группой энтузиастов, познакомиться с которыми вы также сможете на этом сервере. Система включает все возможности BSD UNIX и к тому же является совершенно бесплатной. Если у вас дома имеется персональный компьютер, подключенный к Internet, вы сможете установить систему по сети. После этого возвращайтесь на сервер и расширяйте возможности вашей рабочей станции, устанавливая дополнительное программное обеспечение.

**OpenBSD (<http://www.openbsd.org/>).** Здесь вы познакомитесь с проектом OpenBSD, направленным на разработку свободно распространяемой системы OpenBSD, в основе которой лежит версия UNIX 4.4BSD. Последняя версия системы OpenBSD 2.1 была выпущена участниками проекта 2 июня 1997 года. Посетите этот сервер и вы узнаете, как получить и установить эту систему.

**NetBSD (<http://www.netbsd.org/>).** Еще один проект, посвященный разработке свободно распространяемой системы ветви BSD UNIX. Здесь вы узнаете о целях проекта, его истории и участниках, а также получите прак-

тические советы, как получить дистрибутив операционной системы, установить ее и использовать в своей работе.

**UNIXhelp for Users (<http://www.winterweb.com/UNIX/>).** Полезный справочный материал по UNIX, созданный в Университете Эдинбурга, Великобритания. Здесь вы можете найти советы по работе в операционной системе, например по настройке пользовательского окружения или управлению задачами. Вы также ознакомитесь с основными концепциями UNIX, а также с правилами применения основных команд и утилит. Если online-доступ к этой информации кажется вам слишком медленным, вы сможете переписать и установить электронную версию справочника на собственном компьютере.

**Sun World Online (<http://www.sun.com/sunworldonline/>).** В этом разделе сервера компании Sun Microsystems вы найдете электронные версии журнала SunWorld. Краткие обзоры и аналитические статьи, посвященные различным аспектам, связанным с операционной системой Solaris и UNIX вообще, рекомендации экспертов и советы для начинающих, все это вы встретите на страницах журнала. Вы можете оформить подписку и получать по электронной почте уведомления о новых номерах журнала (на англ. яз.).

**Журнал UnixWorld (<http://www.unixworld.com/unixworld/>).** Здесь вы сможете ознакомиться с электронным журналом UnixWorld, материалы которого содержат практические рекомендации для начинающих пользователей, экспертов и системных администраторов различных версий операционной системы UNIX.

**Вопросы и ответы по операционной системе Solaris (<http://zaphod.cs.uni-sb.de/Corner/solaris2.html>).** Если у вас возникла проблема, загляните в раздел "Вопросы и ответы" сервера.

**Сервер FreeBird (<http://www.freebird.org>).** Сервер в основном посвящен операционной системе SCO UnixWare, и конечно содержит информацию полезную для пользователей других версий UNIX. На этом сервере вы можете заглянуть в онлайн страницы электронного справочника man, скопировать разнообразное программное обеспечение, познакомиться с телеконференциями, посвященными UNIX и многое другое.

**Unix Guru Universe (<http://www.ugu.com/>).** Этот сервер по праву называется официальным сервером системных администраторов UNIX. По количеству справочного материала, практических рекомендаций, программного обеспечения, ссылок на другие ресурсы, имеющие отношение к UNIX, этот сервер не имеет себе равных.

**Ресурсы UNIX (<http://wwwhost.cc.utexas.edu/cc/services/unix/index.html>).** Прекрасный сервер Университета штата Техас, созданный в рамках проекта ACITS (Academic Computing and Instructional Technology Services). На этом сервере вы сможете воспользоваться электронными версиями доку-

ментации и руководств, найти различные ресурсы, связанные с вопросами безопасности, сетевой поддержки, программного обеспечения.

**Защита данных (<http://voyager.crrel.usace.army.mil/~pete/security.html>).** На сервере вы найдете информацию по различным аспектам защиты данных в операционной системе UNIX, начиная с советов по выбору паролей и заканчивая практическими рекомендациями по защите вашей системы от несанкционированного доступа.

**Координационный центр CERT (<http://www.cert.org/>).** Это официальный сервер Координационного центра группы быстрого реагирования по компьютерной безопасности CERT (Computer Emergency Response Team). Здесь вы получите исчерпывающие ответы на вопросы об истории CERT, целях и задачах этой организации, а также узнаете, как улучшить защищенность вашей системы от несанкционированного доступа.

**Электронная библиотека документации по SCO UNIX (<http://www2.sco.com:1996/dochome.html>).** На этом сервере расположены электронные гипертекстовые версии документации по операционным системам SCO UNIX. Прекрасный источник информации для пользователей SCO UNIX.

**Jeff's UNIX Vault (<http://www2.shore.net/~jblaine/vault/>).** Прекрасная коллекция ссылок на ресурсы Internet, связанные с операционной системой UNIX.

**Домашняя страница Ричарда Стевенса (<http://www.kohala.com/~rstevens/>).** Если вы хотите познакомиться с автором замечательных книг по программированию в UNIX Ричардом Стевенсом, посетите его домашнюю страницу. Здесь вы найдете полную библиографию его книг с аннотациями и содержанием, а также большое количество ссылок на другие источники информации по операционной системе UNIX.

**Зеркальный сервер компании SCO (<ftp://ftp.olly.ru/>).** На сервере Санкт-Петербургской фирмы OLLY представлен большой объем справочной информации, драйверов, условно бесплатного программного обеспечения для операционных систем компании Santa Cruz Operation, Inc., включая зеркальные копии многих разделов сервера [ftp.sco.com](ftp://ftp.sco.com).

**Книги и документация по UNIX на русском языке (<http://pluto.xTech.RU/Russian/Unix-Doc/>).** На сервере Новосибирского института систем информатики представлены переводы книг М. Баха "Архитектура операционной системы UNIX", М. Уэлша "Инсталляция Linux и первые шаги", а также некоторые разделы электронного справочника *man*.

# Предметный указатель

Абсолютное имя файла, 143

Адрес

сокета, 269

Класс, 398

IP, 389; 398

MAC, 389

линейный, 201

Адресное пространство

процесса, 204

ядра, 204

Алармы, 219

Атрибуты пользователя, 51

Блоки хранения данных, 281; 290

Брейк-адрес, 151

Буферный кэш, 312

Диспетчер, 316

## В

Ввод/вывод

Подсистема, 18

Потоки, 66

Буферизация, 132

scatter-gather, 128

Версии системы UNIX

BSD UNIX, 7

System III, 6

System V, 6

System V Release 4, 7

OSF/1, 8

Виртуальная память, 18; 197

Виртуальная файловая система, 293

Владение файлами, 28; 140

Внутренняя структура, 313

Временной квант, 216

Вторичная память, 199

Уровни выполнения системы, 187

## Г

Группа

цилиндров, 288

процессов, 173; 237

пользователей, 51

## Д

Датаграмма, 394

Демон (пример), 180

Дескриптор сегмента, 199

Диспетчер буферного кэша, 316

Диспозиция сигнала, 161

Дочерний процесс, 42; 154

Драйверы, 323; 369

Встраивание в ядро, 338

Клоны, 335

Типы, 323

## З

Задание, 80

Запуск новой программы, 230

## И

Идентификатор

пользователя, 51; 53

первичной группы, 53

процесса, 147

Индексный дескриптор, 20; 282

Виртуальный, 293

Массив (ilist), 281

Интерфейс

DLPI, 487

Примитивы, 497

Точка физического подключения, 488

Точка доступа к услугам, 488

TLI, 426

TP1, 472

Транспортные примитивы, 473

доступа низкого уровня, 47

системных вызовов, 16

## К

Каналы, 128; 242

Каталог, 21; 291

Корневой, 20; 26; 142

Текущий, 142

файловой системы s5fs, 285

Клоны, 335

Код возврата, 119

Командный интерпретатор, 56

Запуск команд

условный, 73

в фоновом режиме, 73

Команды, 82  
 Переменные, 60  
   экспортируемые, 64  
   внутренние, 64  
 Подстановки, 71  
 Приглашение  
   первичное, 63  
   вторичное, 63  
   Пример, 184  
 Система управления заданиями, 80  
 Скрипт, 57  
   Код возврата, 65  
 Условные выражения, 74  
 Функции  
   определенные пользователем, 69  
   встроенные, 69  
 Циклы, 77  
 Коммуникационный домен, 264  
 Коммутатор  
   протокола, 454  
   устройств, 325  
   файловых систем, 298  
 Контекст процесса, 221

## М

### Маршрут

по умолчанию, 459  
 прямой и косвенный, 460  
 Маршрутизация, 386; 394; 458  
   Таблица, 458  
 Маска сети, 399  
 Метаданные файла, 20; 144; 282  
 Микроядро, 8  
 Модель OSI, 391  
   Уровень  
     приложений, 387  
     сетевой, 387  
     транспортный, 387  
     управления доступом к среде  
       передачи, 490  
     управления логическим каналом,  
       490

## Н

Наследование атрибутов, 154; 156; 227

Область памяти, 111; 207  
 Обработка ошибок, 95  
 Обработчик выхода, 119  
 Ограничение ресурсов, 177; 287  
   изменяемое, 177  
   жесткое, 177

Окно, 411  
   переполнения, 416; 418  
 Операции, 295; 297  
 Отложенный вызов, 218

## П

Память  
   Брейк-адрес, 151  
   виртуальная, 197  
   вторичная, 199  
   Выделение, 150  
   Область, 111; 207  
   разделяемая, 258  
   Сегмент, 198; 404  
   Страница, 202; 198  
 Пароль пользователя, 52  
   Требования, 54  
 Переменные окружения, 115  
 Подсистема  
   STREAMS, 350  
     Головной модуль, 370  
     Модули, 356  
       IP, 466  
       TCP, 469  
       *timod(7M)*, 485  
       UDP, 468  
     Мультиплексирование, 378  
     Создание потока, 373  
     Сообщения, 357  
       типы, 361  
     ввода/вывода, 18  
     файловая, 18  
     управления процессами, 18  
 Пользователи системы  
   стандартные, 55  
   Псевдопользователи, 51  
   Суперпользователь, 51  
 Порт, 389; 401  
 Права доступа, 140  
   для каталогов, 32  
   к файлу, 30  
 Приоритет процесса, 222  
 Протокол  
   IP, 389; 393  
     Инкапсуляция, 493  
     LLC, 492  
   TCP, 404  
     Состояния, 406  
     Синдром "глупого окна", 414  
     Медленный старт, 416  
     Устранение затора, 417  
     Быстрая повторная передача, 420

- UDP, 402  
 Семейство TCP/IP, 383; 464  
     архитектура, 386  
     скользящего окна, 405  
     Управляющий блок, 464  
 Процесс, 38  
     Адресное пространство, 197; 204  
     Атрибуты, 41  
         EUID, 41  
         Nice Number, 41  
         PID, 41  
         PPID, 41  
         RUID, 41  
         TTY, 41  
     Выполнение, 189  
         в режиме ядра, 189; 233  
         в режиме задачи, 189  
         Планирование, 18; 216  
     Группа процессов, 173; 237  
     дочерний, 42; 154  
     зомби, 193; 236  
     Идентификаторы, 147  
     Контекст, 221  
         переключение, 194; 221  
     Межпроцессное взаимодействие, 241  
     Ограничения ресурсов, 177  
     Переменные окружения, 115  
     Приоритет, 222  
         относительный, 86; 223  
         сна, 223  
         текущий, 223  
         родительский, 42; 154  
     Создание, 154; 226  
     Сон, 194  
     Состояния, 191  
     Структуры данных,  
         типы, 39  
     Управление, 86  
 Псевдотерминалы, 348  
     Основной драйвер, 348  
     Подчиненное устройство, 348  
 Путь, 20  
  
 Раздел диска, 280  
 Разделяемая память, 258  
 Родительский процесс, 42; 154  
  
 Свопинг, 199; 211  
     Область, 199  
 Связь  
     символическая, 22; 134  
     жесткая, 22  
 Сеанс, 173  
 Сегмент, 198  
     селектор сегмента, 199  
 Семафор, 253  
     Пример использования, 260  
 Сетевой интерфейс, 456  
 Сигналы  
     SIGBUS, 138  
     SIGALRM, 220  
     SIGCHLD, 158  
     SIGHUP, 175  
     SIGINT, 160; 175  
     SIGKILL, 46; 161  
     SIGQUIT, 160; 175  
     SIGSTOP, 161; 195  
     SIGSTP, 175  
     SIGTERM, 46  
     SIGTTIN, 175; 195  
     SIGTTOU, 175; 195  
     SIGXCPU, 217  
     Диспозиция, 161  
     Доставка и обработка, 238  
     Маска, 167  
     Набор, 166  
     надежные, 160; 166  
     Отправление, 237  
 Система межпроцессного  
     взаимодействия  
         FIFO, 22; 243  
         Идентификаторы, 245  
         Каналы, 128; 242  
         Пространство имен, 265  
         Разделяемая память, 258  
         Семафоры, 253  
         Сообщения, 248; 357  
         Сравнение различных средств, 277  
 Системный журнал, 183  
 Системный вызов, 16; 94  
     *accept(2)*, 272; 421  
     *alarm(2)*, 220  
     *bind(2)*, 269; 421  
     *chdir(2)*, 143  
     *chmod(2)*, 141  
     *chown(2)*, 140  
     *chroot(2)*, 143  
     *close(2)*, 125  
     *connect(2)*, 271  
     *creat(2)*, 124  
     *dup(2)*, 125  
     *dup2(2)*, 125  
     *exec(2)*, 42, 230  
     *exit(2)*, 118; 235  
     *fchdir(2)*, 143

*fchmod(2)*, 141  
*fchown(2)*, 140  
*fchroot(2)*, 143  
*fcntl(2)*, 129  
*fork(2)*, 42, 226  
*getmsg(2)*, 363; 371  
*getpgid(2)*, 173  
*getpgrp(2)*, 173  
*ioctl(2)*, 380; 463; 485  
*kill(2)*, 161  
*lchown(2)*, 140  
*link(2)*, 134  
*listen(2)*, 421  
*lseek(2)*, 126  
*mknod(2)*, 243  
*mmap(2)*, 317  
*msgctl(2)*, 250  
*msgget(2)*, 249  
*msgrecv(2)*, 250  
*msgsnd(2)*, 250  
*open(2)*, 122  
*pipe(2)*, 128; 242  
*putmsg(2)*, 363; 370  
*read(2)*, 126; 314; 363  
*ready(2)*, 126  
*readlink(2)*, 135  
*recv(2)*, 273; 421  
*recyfrom(2)*, 273  
*semget(2)*, 255  
*semop(2)*, 255  
*send(2)*, 273; 421  
*sendto(2)*, 273  
*setegid*, 149  
*seteuid*, 149  
*setgid*, 149  
*setsid(2)*, 174  
*setuid*, 149  
*shmat(2)*, 259  
*shmdt(2)*, 260  
*shmget(2)*, 259  
*sigaction(2)*, 166  
*socket(2)*, 266  
*stat(2)*, 144  
*sync(2)*, 316  
*sysconf(2)*, 116  
*symlink(2)*, 134  
*unlink(2)*, 134  
*vfork(2)*, 229  
*wait(2)*, 158  
*waitid(2)*, 158  
*waitpid(2)*, 158  
*write(2)*, 127; 314; 363  
*writev(2)*, 127

для работы с файлами, 121  
 ошибки, 97  
 Скрипт, 57  
 Сокет, 25; 264; 453  
 Пример использования, 274  
 Программный интерфейс, 420  
 Типы, 265  
 Сон процесса, 234  
 Сообщения, 248  
 Пример использования, 251  
 Стандарты  
 ANSI, 11  
 IEEE 802.3, 491  
 POSIX, 10  
 SVID, 11  
 XPG3, 11  
 Страница памяти, 202  
 Страницочное замещение, 211  
 Структура данных  
*clist*, 345  
*ilist*, 281  
*netbuf*, 427  
*passwd*, 149  
*preigion*, 207; 229  
*proc*, 190  
*queue*, 356  
*region*, 207; 229  
*sigaction*, 167  
*siginfo\_t*, 168  
*snode*, 334  
 общий, 335  
*stat*, 144  
*user*, 190  
*vfs*, 297  
*vnode*, 294  
 Суперблок, 281  
 Суперпользователь, 51

**Т**

Таблица  
 дескрипторов, 200  
 маршрутизации, 458  
 Таймер, 469  
 Тик, 217  
 Текущий рабочий каталог, 142  
 Терминальная линия, 346  
 Неканонический режим, 347  
 Канонический режим, 347  
 Типы процессов, 39  
 Точка  
 входа, 325  
 доступа к услугам, 488

монтирования, 298  
физического подключения, 488

## Удаленный вызов процедур (RPC), 440

Заглушка, 441  
Обработка особых ситуаций, 444  
Передача данных, 410  
Передача параметров, 442  
Представление данных, 445  
Семантика вызова, 444

## Управление

заданиями, 80  
передачей данных, 364  
процессами, 86

## Управляющий терминал, 174; 237

## Устройства, 342

блочные, 22; 340  
Номер, 48; 325  
символьные, 22

## Утилиты

*at(1)*, 87  
*cat(1)*, 84  
*chgrp(1)*, 86  
*chgrp(1)*, 29  
*chmod(1)*, 86  
*chown(1)*, 29; 86  
*cmp*, 82  
*cp*, 83  
*cut*, 85  
*diff*, 82  
*egrep*, 83  
*fgrep*, 83  
*file*, 86  
*find*, 85  
*fsck(1M)*, 321  
*getty(1M)*, 57; 118; 349  
*grep*, 83  
*head*, 84  
*kill*, 87  
*ld(1)*, 106  
*In*, 83  
*login(1)*, 57; 149; 349  
*ls*, 83  
*mkdir*, 83  
*more*, 84  
*mv*, 83  
*nice*, 86  
*pg*, 84  
*ps*, 87  
*re nice*, 86  
*rm*, 83  
*rmdir*, 83

*sort*, 84  
*to//*, 84  
*wc*, 85

## Ф

Файловая система, 20  
BSD UNIX, 288  
System V, 280  
specfs, 333  
Виртуальная, 293  
Монтирование, 296  
Структура, 26  
Целостность, 317  
Файловая таблица, 307  
Файловый дескриптор, 123; 306  
Файловый указатель, 130  
Файлы, 20

Атрибуты  
Дополнительные, 35; 141  
Sticky bit, 36  
SGID, 36  
SUID, 36  
Блокирование доступа, 309  
обязательное, 311  
рекомендательное, 310  
Блокирование записи, 130  
Владение, 28; 140  
Имена, 285  
абсолютное, 143, 303  
относительное, 143; 303  
трансляция, 303  
Класс доступа, 30  
Метаданные, 20; 144; 282  
Отображаемые в памяти, 137  
специальные блочных устройств, 47  
специальные символьных устройств, 22; 47  
Типы, 21

## Форматы

COFF, 112; 205  
ELF, 108; 205  
исполняемых файлов, 107  
кадров, 491

## Фрагментация, 394

Функции стандартных библиотек  
*atexit(3C)*, 120  
*calloc(3C)*, 152  
*free(3C)*, 153  
*getenv(3C)*, 116  
*malloc(3C)*, 152  
*mktemp(3C)*, 275  
*openlog(3)*, 184  
*putenv(3C)*, 116

- realloc(3C)***, 152  
***signal(3C)***, 164  
***syslog(3)***, 183  
***t\_accept(3N)***, 432  
***t\_bind(3N)***, 431  
***t\_close(3N)***, 433  
***t\_connect(3N)***, 431  
***t\_listen(3N)***, 432  
***t\_look(3N)***, 439  
***t\_open(3N)***, 429  
***t\_recv(3N)***, 433  
***t\_rcvrel(3N)***, 438  
***t\_rcvudata(3N)***, 433  
***t\_snd(3N)***, 433  
***t\_sndrel(3N)***, 438  
***t\_sndudata(3N)***, 433  
Функция *main()*, 114
- Ш**  
Шлюзы, 387
- А**  
ANSI, 11
- С**  
COFF, 112; 205  
Computer Research Group, 5
- Е**  
ELF, 108; 205  
Ethernet, 491
- Ф**  
FIFO, 22; 243  
Пример использования, 244
- І**  
IEEE 802.3, 491  
IP, 389
- М**  
MULTICS, 3
- О**  
OSF/1, 8
- Р**  
PDU, 389  
POSIX, 10  
Programmer's WorkBench, 5
- Р**  
RPC, 440
- С**  
Sticky bit, 36  
STREAMS, 426; 471  
SVID, 11
- Т**  
TCP, 404  
TCP/IP, 383; 464
- У**  
UDP, 402  
UNIX System Group, 4
- Х**  
X/Open, 10  
XPG3, 11