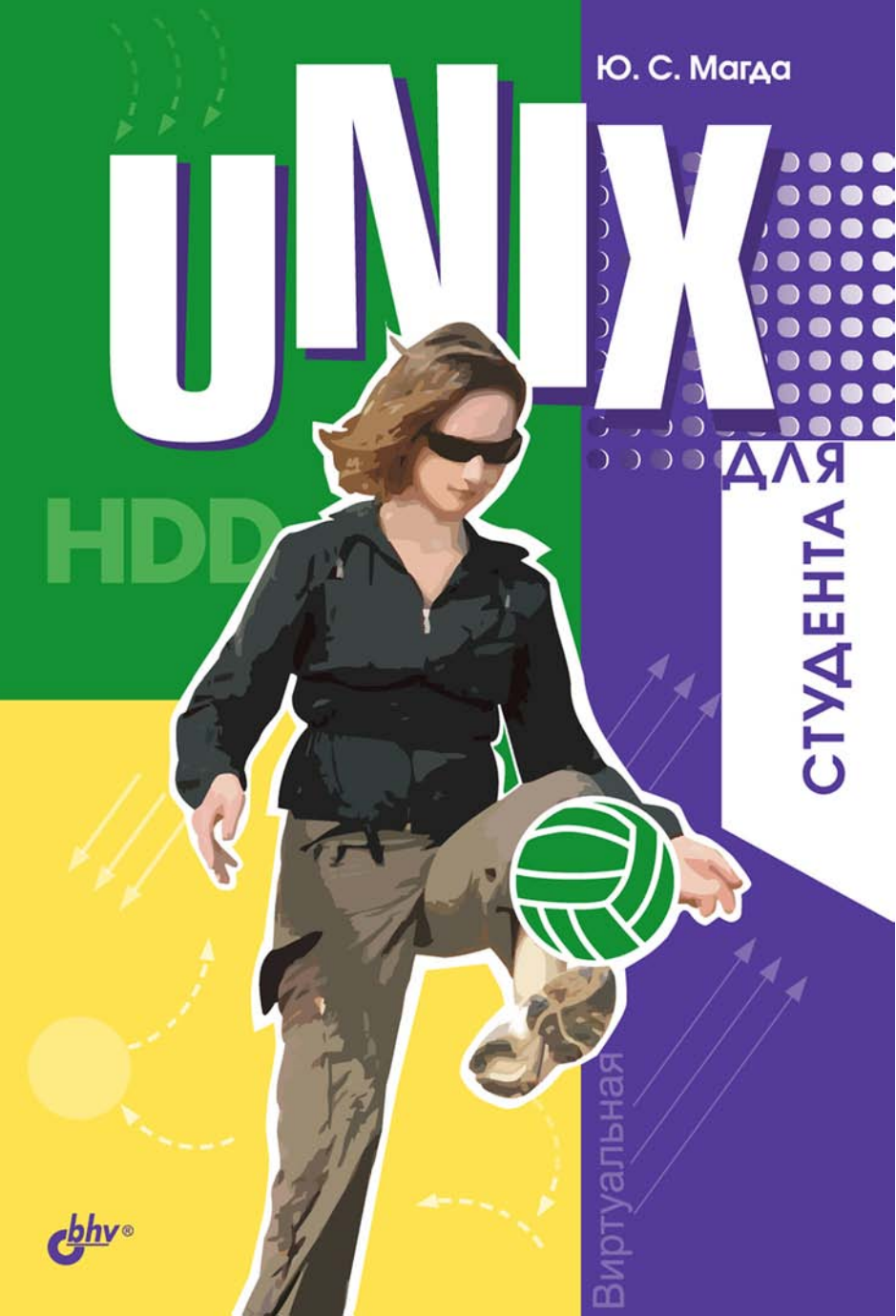
Ю. С. Магда

UNIX для студента



Виртуальная

bhv®

Юрий Магда

UNIX
ДЛЯ СТУДЕНТА

Санкт-Петербург
«БХВ-Петербург»
2007

УДК 681.3.068(075.8)
ББК 32.973.26-018.1я73
М12

Магда Ю. С.

М12 UNIX для студента. — СПб.: БХВ-Петербург, 2007. — 480 с.: ил.

ISBN 978-5-94157-972-3

Рассматривается широкий круг вопросов функционирования операционной системы UNIX, в том числе: базовые вопросы построения операционной системы, принципы организации файловой системы UNIX, учетные записи пользователей, установка, запуск и функционирование популярных операционных систем Linux, FreeBSD и Solaris, взаимодействие пользователя с операционной системой и командные оболочки. С позиции пользователя изложены базовые концепции работы операционной системы UNIX в сетях TCP/IP, настройка Интернета и электронной почты. Рассмотрены вопросы разработки программного обеспечения, в том числе программ на языке C, приложений на Java и командных сценариев на языке Perl. Приведены исходные коды целого ряда программ и примеров. В доступной форме даны принципы построения графических интерфейсов пользователя и работа с ними.

Для широкого круга пользователей UNIX

УДК 681.3.068(075.8)
ББК 32.973.26-018.1я73

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.12.06.
Формат 60×90 1/16. Печать офсетная. Усл. печ. л. 30.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-94157-972-3

Оглавление

Введение **1**

Благодарности **5**

Глава 1. Обзор операционных систем UNIX **7**

1.1. Solaris 11
1.2. FreeBSD 11
1.3. Linux 12

Глава 2. Основы организации UNIX **15**

2.1. Ядро 20
2.1.1. Программы, процессы и потоки 24
2.1.2. Взаимодействие процессов 33
2.2. Системные процессы 39

Глава 3. Файловая система **43**

3.1. Иерархия файловой системы 48
3.2. Функции API для работы с файлами 53
3.3. Операции с файлами. Индексные дескрипторы 58
3.4. Права доступа к файлам 68
3.5. Операции с дисковыми файлами 80
3.5.1. Копирование файлов 80
3.5.2. Перемещение файлов 83
3.5.3. Удаление файлов и каталогов 85
3.5.4. Создание каталогов 86
3.6. Поиск файлов и каталогов 87

Глава 4. Учетные записи пользователей 93

4.1. Команды UNIX для работы с учетными записями 111
4.2. Программное управление учетными записями 123

Глава 5. Установка, запуск и функционирование UNIX 131

5.1. Этапы установки системы 134
5.2. Основы создания файловых систем 138
5.2.1. Файловая система UFS 141
5.2.2. Примеры создания файловых систем 145
5.2.3. Диагностика файловых систем 160
5.3. Особенности установки различных операционных систем 161
5.3.1. Установка операционной системы Solaris 180
5.3.2. Установка Linux 183
5.4. Запуск и остановка UNIX 190
5.4.1. Загрузка FreeBSD 191
5.4.2. Запуск Solaris 9 201
5.4.3. Запуск и останов Linux 209

Глава 6. Взаимодействие пользователя с операционной системой: командные интерпретаторы 217

6.1. Элементы языка shell 220
6.2. Командные файлы 240
6.3. Логические структуры командного интерпретатора 247

Глава 7. Сетевые настройки UNIX 259

7.1. Топология сетей 261
7.2. Модели сетевого взаимодействия 267
7.2.1. Модель OSI 267
7.2.2. Стек протоколов TCP/IP 272
7.3. Сетевые приложения 276
7.4. Адресация в Интернете 282
7.5. Маршрутизация 287
7.6. Электронная почта и Интернет 299
7.6.1. Программа *mail* 301
7.6.2. Программа *sendmail* 306
7.6.3. World Wide Web 307

7.7. Сетевые интерфейсы ... 311
7.8. Статистика работы сети ... 315
7.9. Диагностика сети и поиск неисправностей ... 319
7.10. Основы программирования сетевых приложений ... 325

Глава 8. Разработка программного обеспечения в среде UNIX ... 335

8.1. Разработка приложений на C++ ... 339
8.2. Java ... 347
8.2.1. Первая программа на Java ... 351
8.2.2. Синтаксис языка ... 354
8.2.3. Введение в классы ... 356
8.2.4. Обработка ошибок ... 362
8.2.5. Работа со строками ... 364
8.2.6. Пакеты ... 371
8.2.7. Ввод/вывод в Java ... 372
8.2.8. Апплеты ... 383
8.3. Perl ... 384
8.3.1. Запуск программ ... 387
8.3.2. Скалярные переменные и массивы ... 388
8.3.3. Хэши ... 398
8.3.4. Операции и выражения ... 399
8.3.5. Логические структуры языка ... 403
8.3.6. Регулярные выражения ... 407
8.3.7. Обработка файлов и каталогов ... 411
8.3.8. Сетевое программирование в Perl ... 417

Глава 9. Графический интерфейс пользователя ... 421

9.1. Архитектура системы X Window ... 422
9.2. Команды X Window ... 438
9.3. Оконные менеджеры и графические оболочки ... 448
9.3.1. Оконные менеджеры ... 448
9.3.2. Графические оболочки ... 453

Заключение ... 459

Предметный указатель ... 461

Введение

В данной книге рассматриваются основы функционирования одной из популярных и наиболее динамично развивающихся в последнее время операционных систем — UNIX. Материал книги будет полезен в первую очередь студентам соответствующих специальностей и начинающим пользователям, изучающим эту операционную систему. Изложение материала базируется на принципе "о сложном — доступно", причем основное внимание уделяется теоретическим аспектам построения UNIX. В то же время автор книги старается избегать излишне упрощенного подхода при объяснении принципов организации и работы системы, поскольку при этом в тени остаются очень важные нюансы функционирования UNIX.

Для большей наглядности многие теоретические вопросы подкреплены практическими примерами программного кода, позволяющего закрепить полученные знания и, что очень важно, почувствовать, "как это работает" — при этом от читателя требуются лишь минимальные знания языка программирования C. Основное внимание уделяется описанию функционирования системы на уровне ядра и системных служб, что является основой для дальнейшего углубленного изучения операционной системы UNIX. Поскольку операционная система UNIX является прекрасной средой разработки программного обеспечения, особенно Web-приложений, и во многих случаях используется именно в этом качестве, — в книге приводится описание программных средств разработки, используемых для этих целей.

Отличие этой книги от подобных изданий заключается в том, что простота изложения не сказывается на глубине анализа материала. Здесь делается попытка объединения различных подходов к анализу операционной системы — при этом программная архитектура UNIX, программный интерфейс разработчика и интерфейс пользователя рассматриваются как части единого целого, а именно операционной системы. По этой причине многие аспекты взаимодействия пользователя и операционной системы

иллюстрируются примерами программного кода на языке C, что, по мнению автора, способствует глубокому пониманию функционирования различных подсистем UNIX и осмысленным действиям по конфигурированию и настройке операционной системы пользователем.

Материал книги охватывает следующий круг вопросов:

- анализ архитектуры UNIX, позволяющий понять основные аспекты функционирования операционных систем;
- принципы построения файловых систем UNIX;
- основы программной архитектуры системы X Window, а также настройка и использование графического интерфейса пользователя и графических оболочек;
- сетевые настройки операционной системы UNIX, а также практические аспекты функционирования Интернета и электронной почты;
- принципы разработки программного обеспечения в UNIX с использованием языков C, Java и Perl.

Книга состоит из девяти глав, краткое описание каждой из них приведено далее.

- *Глава 1. "Обзор операционных систем UNIX".*

  В главе рассматриваются стандарты операционных систем UNIX и проводится сравнительный анализ наиболее популярных программных продуктов, таких как Linux, FreeBSD и Solaris. Рассмотрены также и наиболее распространенные аппаратные платформы, на которых работает UNIX.

- *Глава 2. "Основы организации UNIX".*

  Материал главы посвящен обзору архитектуры операционных систем UNIX. Рассматриваются принципы построения UNIX, взаимодействие ядра с другими подсистемами, а также механизмы запуска и выполнения программ пользователя. Здесь же анализируются механизмы реализации многозадачности и многопользовательского режима. Подробно рассматриваются механизмы взаимодействия процессов, выполняющихся в операционной системе, с ядром.

- *Глава 3. "Файловая система".*

  Эта глава посвящена анализу работы файловой системы. Рассмотрена организация и структура файловых систем, операции с файлами и каталогами, а также назначение и изменение прав доступа к объектам файловой системы. Анализируются практические аспекты создания и управления файловыми системами.

- *Глава 4. "Учетные записи пользователей".*

  В главе детально рассматриваются вопросы организации доступа пользователей в UNIX. Основное внимание уделено вопросам создания, удаления и изменения учетных записей пользователя. Проанализированы основные аспекты безопасности системы и политика выбора паролей. Здесь же рассмотрен вход в систему и проверка подлинности учетных записей. Кроме того, приводится методика настройки рабочей среды пользователя.

- *Глава 5. "Установка, запуск и функционирование UNIX".*

  Детально описаны основные этапы установки и настройки операционных систем UNIX. Рассмотрены основные аспекты инсталляции операционных систем, включая выбор и подготовку дисковых накопителей, планирование, создание и монтирование файловых систем, включая примеры создания файловых систем. Здесь же проанализирована последовательность операций при запуске и остановке UNIX.

- *Глава 6. "Взаимодействие пользователя с операционной системой: командные интерпретаторы".*

  В этой главе рассматривается использование встроенного командного интерпретатора shell. Приводится описание синтаксиса shell, включая описание переменных и логических программных структур интерпретатора. Представлены многочисленные примеры применения командного интерпретатора shell для решения различных задач.

- *Глава 7. "Сетевые настройки UNIX".*

  Глава знакомит читателя с принципами функционирования операционной системы UNIX в сетях TCP/IP. Детально рассматриваются основы построения сетей и базовые протоколы

взаимодействия сетевых приложений, а также вопросы настройки и конфигурирования протокола TCP/IP в операционной системе UNIX. Большое внимание уделено тестированию сетей и поиску неисправностей. Здесь же анализируются основы работы в Интернете, а также описан протокол HTTP, являющийся базисным при обмене информацией между клиентами и Web-узлами.

Часть материала главы посвящена анализу основ функционирования систем электронной почты в UNIX. Здесь рассматриваются теоретические основы построения таких систем, а также анализируются основные аспекты использования наиболее популярных программ электронной почты, включая описание основных протоколов электронной почты.

- *Глава 8. "Разработка программного обеспечения в среде UNIX".*

  В главе обсуждаются наиболее важные аспекты разработки программного обеспечения для систем UNIX с использованием популярных языков программирования C, Java и Perl. Здесь рассмотрены принципы компиляции и сборки приложений с помощью популярного компилятора g++. Значительная часть материала главы посвящена описанию синтаксиса языков Java и Perl с детальным обсуждением примеров программного кода.

- *Глава 9. "Графический интерфейс пользователя".*

  В главе рассматриваются основы построения графических интерфейсов UNIX-систем. Значительная часть материала главы посвящена конфигурированию и настройке графической системы X Window, являющейся основой создания графических приложений для операционных систем UNIX. Кроме этого, рассматриваются основные принципы функционирования и настройки популярных графических оболочек GNOME и KDE.

Материал книги окажет существенную помощь читателям, имеющим определенный опыт работы с операционными системами UNIX, которые хотели бы существенно пополнить свои знания в этой области. Книга может быть полезна также системным администраторам и разработчикам программного обеспечения.

Благодарности

Автор выражает огромную благодарность сотрудникам издательства "БХВ-Петербург" за подготовку материалов книги к изданию. Особая признательность жене Юлии за неоценимую помощь и поддержку при подготовке рукописи.

Глава 1

Обзор операционных систем UNIX

Операционные системы UNIX переживают в настоящее время бум популярности. Будучи фактически единственным конкурентом другой популярной операционной системы — Windows, UNIX отвоевывает с каждым днем все больше и больше позиций в нише наиболее распространенных операционных систем.

Еще с момента своего появления операционная система UNIX стала широко применяться в вычислительных системах различной производительности, начиная персональными компьютерами и заканчивая большими вычислительными комплексами. Популярность этой операционной системы обусловлена несколькими факторами.

Во-первых, это более чем тридцатилетний цикл развития — за этот период операционная система UNIX выдержала проверку временем и проявила себя как очень эффективная программная среда. Во-вторых, программный код системы полностью написан на языке высокого уровня C, что делает ее понятной для пользователей и разработчиков программного обеспечения. Это позволяет относительно легко вносить изменения как в существующие версии UNIX, так и переносить операционную систему на другие аппаратные платформы. Кроме этого, во многих случаях версии этой операционной системы поставляются вместе с исходными текстами, что позволяет легко адаптировать UNIX

под специфические требования, после чего перекомпилировать систему.

Операционная система UNIX создавалась как многопользовательская и многозадачная система, ориентированная в первую очередь на выполнение серверных или управляющих функций для клиентских компьютеров. Такие подходы, а также мощные встроенные средства удаленного администрирования, способствовали тому, что UNIX заняла лидирующие позиции на рынке серверов Интернета и серверов баз данных. Существенным фактором в популяризации операционной системы является и структура ее файловой системы, представляющая единую иерархию объектов с унифицированным доступом как к файлам данных, так и к аппаратным ресурсам, таким как диски, терминалы, принтеры, сеть, память и т. п.

В практическом плане операционная система UNIX предоставляет пользователю целый ряд преимуществ. Перечислим только некоторые из них:

- все популярные приложения (пакеты офисных программ, базы данных и др.) известных производителей программного обеспечения, как правило, реализованы для работы в операционной системе UNIX;
- UNIX поддерживает широкий спектр средств передачи информации, включая многочисленные сетевые и коммуникационные протоколы, что обеспечивает эффективную работу операционной системе в сетях;
- операционная система UNIX имеет весьма развитые средства системного и сетевого управления, включая эффективные инструменты для удаленного администрирования и настройки;
- UNIX является мощной платформой для разработки приложений, в нее включены наиболее популярные языки разработки приложений, средства работы с базами данных, а также другое программное обеспечение;
- операционная система поддерживает все основные аппаратные процессорные архитектуры, включая надежную поддерж-

ку SMP, MMP и кластерных систем. В других серверных средах такая поддержка отсутствует;

- практически все важнейшие промышленные, международные, официально утвержденные и неофициальные стандарты были впервые разработаны для UNIX и только впоследствии распространились и на другие операционные системы. В настоящее время основным стандартом является разработанная консорциумом X/Open Единая спецификация UNIX, содержащая более тысячи интерфейсов прикладных программ и поддерживаемая всеми основными производителями операционной системы UNIX. Несмотря на то, что различные версии UNIX обладают уникальными возможностями, все они отвечают требованиям стандарта POSIX (Portable Operating System Interface, переносимый интерфейс операционной системы), а также стандарта X/Open Portability Guide, Edition 4 (XPG 4) и сертифицированы X/Open на соответствие стандартам UNIX 93;
- операционная система UNIX к настоящему времени утвердилась как платформа для персональных приложений, приложений для рабочих групп и приложений корпоративного класса.

В настоящее время единого стандарта для системы UNIX не существует, и на рынке присутствует множество версий этой операционной системы, имеющих свои названия и особенности. Тем не менее, все без исключения UNIX-системы имеют однотипную архитектуру, интерфейсы и среду программирования, что дает основание считать все UNIX-системы в той или иной степени родственными.

Простота и способность операционных систем UNIX к расширению и модификациям являются весьма серьезными преимуществами по сравнению с другими системами, поэтому UNIX стали переносить на множество платформ. Тем не менее, несмотря на множество реализаций базовой системы, среди них выделяются две основные ветви, берущие начало от System V UNIX и BSD UNIX.

Одна ветвь происходит от версий 4.2, 4.3 или 4.4BSD, другая базируется на системах SVR3 (System V Release 3) или SVR4 (System V Release 4). На протяжении ряда лет версия BSD пользовалась большей популярностью в академических и научных кругах, в то время как версии System V, разработанные компанией AT&T, занимали лидирующие позиции в коммерческих организациях и в промышленности. Несмотря на существующие различия между этими основными типами UNIX-систем, подавляющее большинство пользовательских команд работает одинаково и имеет один и тот же синтаксис во всех версиях операционных систем, независимо от того, используется ли AIX, BSD, HP/UX, Linux или Solaris.

Существующие в настоящее время различия между операционными системами не имеют принципиального значения, и определить, к какой из ветвей принадлежит та или иная реализация операционной системы, иногда бывает довольно сложно. Вот основные различия между операционными системами System V и BSD:

- разные способы установки и настройки терминальных устройств;
- разные способы инициализации, именования конфигурационных файлов и файлов инициализации системы;
- различная настройка параметров файловой системы;
- различные методы получения диагностической информации и ее отображения на консоли

и т. п.

С точки зрения пользователя принципиальных различий между разными ветвями операционной системы UNIX не существует. Справедливости ради следует отметить, что BSD и System V — далеко не единственные реализации операционной системы UNIX. Рассмотрим особенности реализации некоторых, наиболее популярных версий операционной системы UNIX и начнем с Solaris.

1.1. Solaris

Исходный код UNIX System V Release 4 был доработан компанией Sun, в результате чего появилась реализация операционной системы UNIX, названная Solaris. Эта операционная система имеет несколько основных отличий от базовой операционной системы. В частности, в Solaris были добавлены следующие возможности:

- симметричная многопроцессорная обработка;
- режим реального времени.

В настоящее время Solaris является одной из самых распространенных версий операционной системы UNIX и работает на платформах SPARC и Intel86. Для развития продукта фирма Sun Microsystems предоставила более открытый доступ к кодам операционной среды Solaris.

1.2. FreeBSD

Другая ветвь UNIX — FreeBSD, берет свое название от "Berkeley Software Distribution". Эта операционная система основана на версии 4.4BSD-Lite и сохраняет специфичные черты модели развития BSD-систем.

На базе версии 4.4BSD-Lite было создано несколько операционных систем с открытыми исходными кодами, среди которых особо следует выделить проект GNU. Операционная система FreeBSD позволяет:

- выполнять одновременно несколько задач с динамическим регулированием приоритетов, что распределяет ресурсы компьютера между приложениями и пользователями оптимальным образом;
- одновременно работать многим пользователям и использовать систему совместно для решения ряда задач. Это означает, что системные ресурсы, такие, например, как принтеры и накопители на магнитных лентах, могут распределяться между

пользователями в системе или сети, при этом для каждого пользователя или группы пользователей могут быть установлены определенные ограничения на использование того или иного ресурса. Это позволяет избежать перегрузок в работе операционной системы;

- работать с распространенными сетевыми протоколами и стандартами, такими как SLIP, PPP, NFS, DHCP и NIS. Это позволяет операционной системе FreeBSD эффективно функционировать совместно с другими операционными системами, например, Windows. Кроме того, FreeBSD может использоваться в качестве интернет-сервера, предоставляя полный спектр сервисов (WWW, FTP, маршрутизация);
- использовать стандарт X Window System (X11R6), предоставляющий графический интерфейс пользователю.

Операционная система FreeBSD обеспечивает совместимость на уровне программного кода с большинством программ, разработанных для Linux и System V Release 4, обладая полным комплектом инструментальных средств для разработки программ (языки C, C++, Fortran и Perl). Исходные тексты FreeBSD свободно распространяются через Интернет, так что систему можно оптимизировать для специальных приложений или проектов. Для коммерческих операционных систем такая возможность отсутствует.

FreeBSD очень часто используется как платформа для высокопроизводительных рабочих станций, при этом она оказывается более эффективной по сравнению с другой, не менее популярной операционной системой Linux. Системы на основе BSD могут демонстрировать бóльшую по сравнению с Linux производительность, обеспечивая при этом более высокую надежность. Наконец, операционная система FreeBSD может выполнять код, разработанный для Linux, но не наоборот.

1.3. Linux

Еще одной, очень популярной реализацией UNIX является Linux. Эта версия UNIX обладает большинством свойств, при-

сущих другим реализациям, и, кроме того, включает некоторые дополнительные возможности. Linux — это полная многозадачная многопользовательская операционная система, допускающая одновременную работу многих пользователей.

Операционная система Linux очень популярна среди миллионов пользователей. GNU/Linux вместе с набором инструментальных средств по оценкам экспертов охватывает около 40% рынка UNIX. Многие компании выпускают дистрибутивы Linux — пакеты, включающие ядро, множество утилит, приложений и программное обеспечение для установки ОС. GNU/Linux получила поддержку у таких компаний, как Sun и IBM.

Для разработки программного обеспечения, работающего в Linux, был создан специальный фонд под названием Free Software Foundation (FSF), цель которого заключается в поиске источников финансирования разработки программного обеспечения GNU. Несмотря на относительно короткую историю существования, под эгидой проекта GNU было создано и адаптировано огромное количество программ, среди которых наиболее известными являются утилиты Emacs, gcc (компилятор GNU C) и bash (командная оболочка).

Необходимо отметить, что эта операционная система хорошо совместима с рядом стандартов для UNIX на уровне исходных текстов, включая IEEE POSIX.1, System V и BSD, поскольку создавалась с расчетом на такую совместимость.

Большинство из свободно распространяемого через Интернет программного обеспечения для UNIX может быть откомпилировано для работы в Linux с минимальными изменениями. Более того, все исходные тексты для Linux, включая ядро, драйверы устройств, библиотеки, пользовательские программы и инструментальные средства, также распространяются свободно.

К специфическим особенностям Linux следует отнести контроль работ по стандарту POSIX (используемый оболочками, такими как csh и bash), работу с псевдотерминалами, поддержку национальных и стандартных клавиатур с динамически загружаемыми драйверами клавиатур.

Операционная система Linux поддерживает различные типы файловых систем. Некоторые из них, например, файловая система ext2fs, были созданы специально для Linux. Кроме того, поддерживаются и другие типы файловых систем, такие, например, как Minix и Xenix. Linux включает реализацию файловой системы MS-DOS, позволяющей прямо обращаться к файлам в формате FAT на жестком диске. Наконец, для работы с CD-ROM поддерживается стандарт файловой системы ISO 9660.

Как и другие операционные системы, Linux обеспечивает полный набор протоколов TCP/IP для сетевой работы. Сюда входят драйверы устройств для многих сетевых карт Ethernet, SLIP (Serial Line Internet Protocol), обеспечивающий пользователям доступ по TCP/IP при последовательном соединении, PLIP (Parallel Line Internet Protocol), PPP (Point-to-Point Protocol), NFS и т. д. В систему включена поддержка всего спектра сервисов TCP/IP (FTP, telnet, NNTP и SMTP).

Глава 2

Основы организации UNIX

Материал этой главы посвящен анализу основных принципов, на которых базируется операционная система UNIX. Эти принципы просты и субъективно понятны.

- Операционная система должна быть максимально надежной — это означает, в первую очередь, что система должна сохранять работоспособность при возникновении каких-либо неисправностей (отказы в работе аппаратных средств, ошибки в функционировании программного обеспечения). В более узком смысле надежность означает, что система должна по возможности обеспечивать некоторый минимальный уровень функционирования при возникновении неисправностей, а также способность к быстрому восстановлению. Например, при отказе одного из жестких дисков система должна выполнить ряд восстановительных операций, чтобы оставаться в работоспособном состоянии и обеспечивать работу пользователей — при этом самым существенным является минимизация потерь критически важных данных пользователей.

  Принципиально избежать отказов нельзя — аппаратные средства имеют свойство давать сбои, а приложения пользователя, впрочем, как и программные модули самой системы, могут содержать ошибки. Важным является то, с какими "потерями" система выходит из этой ситуации. Самым неприятным следствием сбоя в работе может быть потеря критически важных данных пользователей, поэтому способность быстрого восстановления системы после сбоев, включая восстанов-

ление целостности данных, является наиболее существенным критерием при оценке надежности.

Операционная система UNIX изначально проектировалась как отказоустойчивая система. Несмотря на некоторые различия в программной архитектуре, во всех современных UNIX-системах предусмотрены базовые механизмы, гарантирующие высокий уровень надежности. Вот основные из них:

- разделение программного кода операционной системы и пользовательских приложений. В практическом плане это означает, что бóльшая часть программного кода операционной системы, включая драйверы устройств и системные сервисы, а также системные области данных и стека, изолирована от пользовательского программного кода и реализована в форме *ядра*. Если, например, пользовательский процесс попытается напрямую, минуя стандартные вызовы операционной системы, обратиться к жесткому диску, то механизмы защиты ядра не позволят выполнить эту операцию, поскольку могут быть разрушены важные дисковые структуры, что сделает систему неработоспособной. Доступ пользовательских приложений к ресурсам системы возможен только посредством определенных стандартных функций, предоставляемых системой UNIX специально для этих целей. В этом случае используется механизм системных вызовов, который мы рассмотрим далее;
- использование механизмов безопасности для доступа пользовательских процессов к ресурсам операционной системы. Каждый процесс может обращаться только к тем ресурсам (файлам и устройствам), для которых система предоставляет ему доступ. При этом каждому пользовательскому процессу назначается индивидуальный уровень привилегий, определяющий, какие операции и с какими ресурсами процесс может выполнять;
- использование механизмов резервного копирования и восстановления данных, включая полное восстановление системы.

- UNIX по своей природе является многопользовательской ОС, обеспечивая возможность одновременной работы нескольких пользователей в одной системе. Поскольку пользователи, работающие в системе, могут использовать один и тот же ресурс одновременно, например, дисковый файл или принтер, система предусматривает механизмы разделения доступа к общим ресурсам, обеспечивая эффективную работу. При доступе к общим ресурсам, кроме разделения доступа, применяется и механизм безопасности, поскольку одни пользователи могут читать и записывать данные в файл, в то время как другие могут только читать данные из файла или вообще не иметь доступа к данным. Операционные системы, допускающие работу в таком режиме, называются *многопользовательскими.*

- Операционная система UNIX позволяет выполняться множеству процессов одновременно, включая как системные, так и пользовательские процессы. Само понятие *"процесс"* является комплексным, но в первом приближении его можно представить как двоичный образ исполняемого дискового файла (программы), загруженный в память и содержащий инструкции процессора, подлежащие выполнению. Выполнение процессов обеспечивается механизмами синхронизации процессов, выполняющимися на уровне ядра системы. Операционная система управляет процессом в течение всего жизненного цикла, от начала создания и до его уничтожения.

- Единый подход, касающийся доступа к ресурсам операционной системы. В операционной системе UNIX все ресурсы, к которым может иметь доступ процесс, являются *объектами файловой системы.* Этот подход лежит в основе построения и функционирования файловой системы UNIX. UNIX рассматривает дисковые файлы программ или данных, принтеры, жесткие диски, терминальные линии и т. д. как объекты файловой системы, доступ к которым осуществляется с помощью системных вызовов. Такой механизм очень удобен, поскольку позволяет использовать единый программный интерфейс как для работы с файлами данных, так и с устройствами, независимо от физической природы объектов. Кроме этого, можно

использовать единые подходы для установки атрибутов безопасности для объектов файловой системы.

Перечисленные особенности UNIX делают ее очень удобной средой для функционирования пользовательских приложений и позволяют легко переносить ее на разные аппаратные платформы.

Посмотрим, как отображены вышеперечисленные возможности операционной системы UNIX в ее программной архитектуре. В упрощенном виде структура операционной системы UNIX показана на рис. 2.1.

Рис. 2.1. Архитектура UNIX

Основой операционной системы UNIX является *ядро*, которое взаимодействует, с одной стороны, с аппаратными средствами, а с другой стороны обеспечивает работу пользовательских программ.

Часть функций операционной системы выполняется *системными процессами*, реализованными, как правило, в форме *процессов-серверов*. Большинство таких процессов запускается обычно при начальной загрузке системы и функционирует непрерывно вплоть до ее останова. Процесс-сервер более известен под названием *"демон"* (daemon), и его основная функция — выполнение запросов на обслуживание клиентских процессов. Например, сетевой демон `inetd` управляет сетевыми сервисами, демон

`telnetd` обслуживает клиентские запросы по протоколу `telnet` и т. д. Процессы-демоны будут рассмотрены далее в этой главе.

Следует сказать, что в ядре операционной системы часть процессов также функционирует подобно демонам, хотя бóльшая часть функций ядра реализована в виде отдельных потоков (kernel threads), а не процессов.

Уровень интерфейса системных вызовов отвечает за взаимодействие программ пользователя и операционной системы. Ни одна программа пользователя не может получить доступ к ресурсам системы иначе, как посредством системных вызовов. *Системные вызовы* (system calls) реализованы как функции интерфейса прикладного программирования API (Application Programming Interface), и их чаще называют "функциями API".

Системные вызовы инициируют смену контекста выполнения процесса: процесс, работающий в режиме пользователя, переключается на выполнение в защищенном режиме (режим ядра). Такое переключение позволяет процессу вызывать защищенные процедуры ядра для выполнения системных функций. Таким образом, системные вызовы обеспечивают программный интерфейс для доступа к управлению системными ресурсами, например, памятью, дисковым пространством и периферийными устройствами. Системные вызовы реализованы в виде "библиотеки времени выполнения" (run-time library), а многие из них используются командными интерпретаторами shell. Следует заметить, что системные функции ядра для корректной работы требуют упаковки аргументов специальным образом.

В качестве примеров системных вызовов можно привести низкоуровневые функции ввода/вывода, такие как `open()`, `read()`, `write()` и `close()`.

Системные вызовы обеспечивают выполнение целого ряда операций:

- трансляции операций пользователя в запросы к драйверам устройств;
- создания, запуска и уничтожения процессов;
- ввода/вывода;

- доступа к файлам и дисковым устройствам;
- поддержки терминальных устройств.

Наличие интерфейсов в форме системных вызовов предоставляет определенные преимущества разработчикам программ. Во-первых, процесс разработки программ становится намного легче, поскольку программисту нет необходимости изучать особенности программных интерфейсов низкого уровня для каждого из устройств.

Во-вторых, повышается надежность системы в целом, поскольку ядро UNIX может проверить корректность запроса программы пользователя на уровне интерфейса, прежде чем ответить на такой запрос.

Наконец, наличие таких интерфейсов позволяет легко переносить программы на другие реализации UNIX.

На самом верхнем уровне модели, показанной на рис. 2.1, находятся программы пользователей. К таким программам относятся как программы, созданные самим пользователем, так и различные офисные приложения, текстовые редакторы, графические оболочки, программы отправки и получения электронной почты и т. д. Большинство пользовательских программ взаимодействует с системой посредством системных вызовов, хотя в некоторых случаях такие программы вообще не обращаются к ядру.

Проанализируем более детально функциональные компоненты операционной системы UNIX и начнем с ядра.

2.1. Ядро

Ядро операционной системы UNIX отвечает за выполнение базовых функций системы, таких как управление процессами, файловой системой и устройствами ввода/вывода, а также безопасностью системы. Функционирование операционной системы в основном определяется ядром, поэтому остановимся на этом более подробно.

Во многих версиях операционной системы UNIX ядро реализовано в виде большого исполняемого дискового файла, который

запускается в момент инициализации системы — так называемое *монолитное ядро*. Монолитное ядро включает все необходимые модули для работы, в том числе и драйверы устройств. Для изменения функциональности такого ядра, например, при необходимости включить новый драйвер устройства, исходный текст ядра нужно полностью перекомпилировать.

Модульное ядро содержит базовые компоненты, необходимые для загрузки системы, при этом дополнительные модули программного кода (обычно это драйверы устройств) загружаются в оперативную память и подключаются к ядру динамически, по мере необходимости, например, при включении аппаратного устройства. Такие дополнительные модули реализованы в виде загружаемых модулей ядра.

Преимуществом модульного ядра является то, что базовый модуль ядра имеет небольшой размер, быстрее загружается и требует меньше ресурсов операционной системы. В то же время монолитное ядро работает чуть быстрее, поскольку не требуется переключений контекста выполняемых процессов (что имеет место в случае загрузки/выгрузки дополнительных модулей) и дополнительной синхронизации, как при использовании отдельных модулей.

В большинстве современных операционных систем UNIX используется комбинированный тип ядра, которое, обладая широким диапазоном функциональности, допускает загрузку дополнительных модулей во время работы операционной системы.

Еще одна особенность функционирования ядра современных UNIX-систем — возможность одновременного выполнения нескольких потоков, функционирующих относительно независимо друг от друга в контексте какого-либо процесса (kernel threads). В первом приближении поток можно представить себе как отдельный выполняющийся фрагмент программного кода в рамках одного процесса, при этом ядро управляет синхронизацией выполнения потоков. В этом случае говорят о *многопоточности ядра*, которая позволяет повысить эффективность функционирования как самого ядра, так и операционной системы в целом.

Высокая эффективность многопоточности объясняется тем, что синхронизация отдельных потоков одного и того же процесса выполняется с минимальными издержками и требует значительно меньше времени, чем запуск отдельного процесса, поскольку не требуется выделение системных ресурсов и обновление системных таблиц. Потоки выполняются в контексте процесса, их создавшего, и не требуют выделения отдельного адресного пространства, в то время как для каждого вновь созданного процесса требуется копия таблиц дескрипторов родительского процесса.

Большинство современных операционных систем UNIX поддерживает выполнение не только отдельных потоков ядра, но и многопоточность на уровне пользовательских программ, повышая тем самым их производительность. В этом случае многопоточные приложения выполняются как совокупность *элементарных* (lightweight) *процессов*, которые используют общее адресное пространство, общие страницы памяти и открытые файлы. Естественно, что для получения выигрыша в производительности выполняющаяся программа должна поддерживать реализацию многопоточности. Еще одной особенностью ядра UNIX является и то, что оно функционирует в *режиме невытесняющей многозадачности* (non-preemptive multitasking). Это означает, что система не может прерывать выполнение процесса, выполняющегося в режиме ядра.

Ядро UNIX обеспечивает выполнение таких функций:

- синхронизация процессов (создание, выполнение, остановка и завершение процессов);
- планирование приоритетов выполнения процессов посредством выделения им времени процессора — в этом случае процесс выполняется в течение определенного ядром интервала (кванта) времени, после чего он приостанавливается, и ядро возобновляет или начинает выполнение другого процесса. Через определенный интервал времени приостановленный процесс возобновляет выполнение и т. д.;
- распределение памяти выполняемым процессам — при этом каждому процессу выделяется определенный объем памяти,

причем адресное пространство процесса недоступно другим процессам. Кроме того, в функции ядра входит управление общим адресным пространством нескольких процессов и областью подкачки;

- создание, изменение и удаление файловых систем, расположенных на устройствах постоянного хранения информации (жесткие диски и магнитные ленты), а также управление данными пользователей. Функции ядра управляют выделением дискового пространства, выполняют отображение физической структуры файловой системы в логическую форму, доступную для работы пользователей, а также устанавливают атрибуты доступа к объектам файловой системы, защищая пользовательские файлы от несанкционированного доступа;
- управление доступом процессов к периферийным устройствам ввода/вывода, таким как терминалы, накопители на магнитных лентах и сетевое оборудование.

Ядро операционной системы является прозрачным для пользовательской программы. Это означает, что детали взаимодействия программы пользователя и операционной системы скрыты от пользователя. Например, если программа пользователя обращается к какому-либо файлу и записывает в него данные, то ядро системы выполняет приблизительно такую последовательность действий:

1. Определяет местоположение файла на носителе.
2. Получает информацию о расположении требуемых данных в физических секторах накопителя.
3. Определяет место записи блока данных в физическую область дискового пространства и т. д.
4. Наконец, ядро вызывает драйвер устройства и передает ему параметры и код операции (записи данных), после чего и выполняется запись данных.

Кроме вышеперечисленных, ядро реализует ряд необходимых функций по обеспечению выполнения процессов пользовательского уровня, за исключением функций, которые реализуются на самом пользовательском уровне.

Например, ядро выполняет определенные действия, необходимые для работы командного интерпретатора shell. Такие функции командного интерпретатора как чтение вводимых с терминала данных, динамическое создание процессов, синхронизация выполнения процессов, открытие программных конвейеров и переадресация ввода/вывода — все они реализуются через системные вызовы ядра.

Рассмотрим более подробно, что же означают некоторые термины и понятия, которые очень часто используются при анализе работы UNIX. Эти термины являются ключевыми, и понимание их смысла очень важно для уяснения работы всей операционной системы и анализа работы основных подсистем UNIX. К таким терминам относятся "программа", "процесс" и "поток".

2.1.1. Программы, процессы и потоки

Программа или, по-другому, приложение представляет собой дисковый исполняемый файл, вызываемый операционной системой для выполнения. *Процесс* в UNIX, как и в других операционных системах, — это выполняющийся образ исполняемого файла. Не следует смешивать эти понятия, поскольку они имеют разный смысл в контексте анализа операционной системы.
В обычном смысле, когда мы говорим, что "программа выполняется" или "программа завершилась", имеется в виду процесс или группа процессов.

Термин "процесс" — это ключевое понятие в UNIX. В первом приближении процессом называют загруженный в память двоичный образ дискового файла, содержимое которого интерпретируется центральным процессором как совокупность машинных инструкций, данных и стековых структур. Работу самой операционной системы можно представить как функционирование совокупности процессов. Например, при входе пользователя в операционную систему начинает выполняться процесс shell, при запуске какой-либо команды, например, `ls -l`, также порождается процесс. Обобщая, можно сказать, что всякий раз, когда вызывается команда UNIX или запускается пользовательская программа, порождается новый процесс. Из этого правила

есть исключения, например, запуск команды `cd`, но подобные случаи мы рассматривать не будем.

Операции, выполняемые при запуске приложения, проиллюстрированы на рис. 2.2.

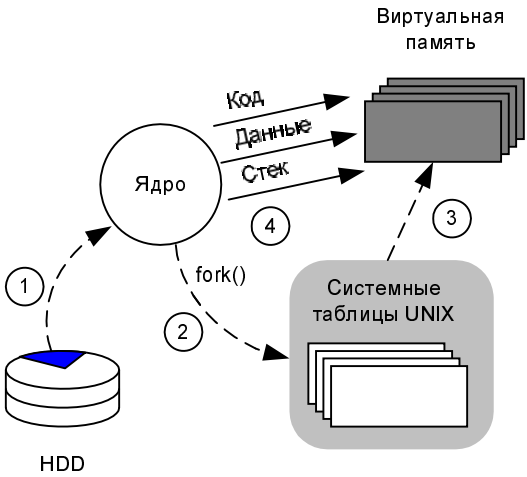


Рис. 2.2. Запуск программы

Операционная система считывает исполняемый файл программы с диска (1) и, если формат файла программы является корректным, выполняет системный вызов `fork()` (2). При этом ядро заносит информацию, относящуюся к созданному процессу, в специальные системные таблицы (таблицу процессов, таблицу дескрипторов файлов, таблицу текущего процесса и таблицу ядра), а также выделяет виртуальную память для кода, данных и стека текущего процесса (3, 4), после чего управление передается первой инструкции процесса.

В простейшем варианте исполняемый файл программы выполняется как один процесс, хотя в большинстве других случаев таких процессов может быть множество. Следует сказать, что и само ядро системы функционирует как совокупность многих взаимосвязанных процессов.

Ядро системы идентифицирует каждый процесс по его номеру, который называется *идентификатором процесса* (Process ID, PID). Каждому процессу присваивается уникальный идентификатор PID, позволяющий ядру различать процессы. В момент создания нового процесса ядро присваивает ему следующий больший по числовому значению свободный идентификатор. Если достигнуто максимальное значение, равное 65 737, то очередной процесс получит минимальный свободный PID.

Завершение выполнения процесса инициируется системным вызовом `exit()`. Родительский процесс при необходимости может проанализировать состояние завершения порожденного процесса, поскольку в системных таблицах сохраняется статистика выполнения процесса. Ядро операционной системы отслеживает окончание работы процесса, освобождая использовавшийся им идентификатор.

Все процессы, выполняющиеся в среде операционной системы UNIX, могут функционировать либо в пользовательском режиме (User Mode), либо в режиме ядра (Kernel Mode). Подобное разделение обусловлено архитектурой системы и возможностью доступа процессов к ресурсам системы (об этом мы поговорим далее). Пользовательский режим не позволяет напрямую обращаться к аппаратным ресурсам системы и системным структурам данных, в то время как процесс, выполняющийся в режиме ядра, имеет такую возможность.

Пространство памяти ядра представляет собой область памяти, в которой процессы ядра или, по-другому, процессы, работающие в контексте ядра, реализуют функции ядра. *Пространство ядра* — это защищенная область, и пользователь получает к ней доступ только через интерфейс системных вызовов. Пользовательский процесс не имеет прямого доступа ко всем инструкциям и физическим устройствам — их имеет процесс, выполняющийся в режиме ядра. Такой процесс может менять карту памяти, что необходимо для переключения процессов (смены контекста). Пользовательский процесс начинает работать в режиме ядра в момент, когда выполняется программный код ядра посредством системного вызова.

Поскольку пользовательские процессы и ядро не имеют общего адресного пространства памяти, необходим механизм передачи данных между ними. При выполнении системного вызова аргументы вызова и соответствующий идентификатор процедуры ядра передаются из пользовательского пространства в пространство ядра. Идентификатор процедуры ядра передается либо через регистры процессора, либо через стек, а аргументы системного вызова передаются через область памяти вызывающего процесса. Область памяти пользовательского процесса содержит информацию о процессе, необходимую ядру, причем пользовательский процесс не может обращаться к пространству ядра, но ядро может обращаться к пространству процесса.

До сих пор мы рассматривали механизм создания и выполнения одного процесса. В большинстве случаев работающий процесс может порождать другие процессы, например, при необходимости параллельной обработки данных или клиентских запросов (если это процесс-сервер наподобие `telnet`, `ftp` и т. д.).

Как и в рассмотренной ранее схеме запуска процесса, здесь используется системный вызов `fork()`. На рис. 2.3 показан пример иерархии нескольких процессов.

При запуске приложения создается основной процесс, который создает (порождает) процессы 1 и 2, процесс 1 создает процесс 3, а процесс 2 создает процесс 4. В этом случае говорят, что основной процесс является *родительским* для процессов 1 и 2, процесс 2 является родительским для процесса 4, а процесс 1 является родительским для процесса 3. В свою очередь, процессы 1 и 2 называют *дочерними* или *порожденными* основным процессом, а процессы 3 и 4 — порожденными для процессов 1 и 2 соответственно.

Каждый порожденный процесс наследует от родительского процесса копию *таблицы дескрипторов* (описателей) открытых файлов. Это означает, что если в родительском процессе открыт файл для записи или чтения, то он будет доступен для записи или чтения и в порожденном процессе. Более того, если порожденный процесс закроет дескриптор файла, то в родительском процессе файл по-прежнему останется доступным.

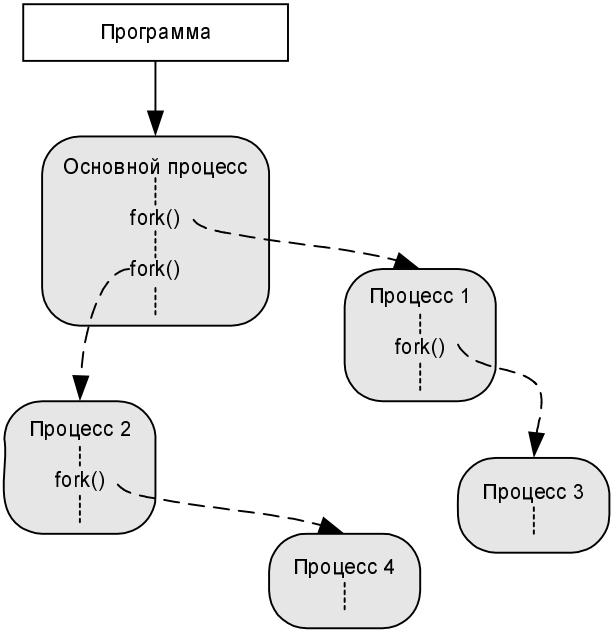


Рис. 2.3. Пример иерархии процессов

Сказанное иллюстрирует пример исходного текста приложения, записывающего строку символов в файл из родительского и порожденного процессов (листинг 2.1).

Листинг 2.1. Запись данных в файл родительским и порожденным процессами

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/wait.h>
```

```
int main(void)
{
  int fd, status;
  char *pstr = "Parents' string\n";
  char *chstr = "Childs' string.\n";
  pid_t pid;

  fd = open("demo1.log", O_RDWR | O_CREAT | O_APPEND);
  write(fd, pstr, strlen(pstr));
  if (pid = fork() == 0)
   {
    write(fd, chstr, strlen(chstr));
    close(fd);
    exit(0);
   }
  else if (pid > 0)
   {
     close(fd);
     wait(&status);
   }
  return 0;
}
```

В этой программе, написанной на C, объявлены две строки `pstr` и `chstr`. Строка `pstr` записывается в файл с именем demo1.log в родительском процессе, а строка `chstr` записывается в этот же файл из порожденного (дочернего) процесса. Для выполнения программы нам понадобится переменная `pid` типа `pid_t`, в которой будет помещено значение идентификаторов родительского и порожденного процессов.

Переменная `fd` будет хранить дескриптор файла, а переменная `status` — информацию о состоянии завершившегося дочернего процесса.

Файл demo1.log открывается с помощью системного вызова `open()` в режиме чтения/записи, при этом, если файл не существует, он будет создан, а новые данные будут добавлены в конец файла:

```
fd = open("demo1.log", O_RDWR | O_CREAT | O_APPEND);
```

Результатом выполнения функции `open()` (если успешно) будет дескриптор файла `fd`. Следующий оператор записывает строку `pstr` в дескриптор `fd`:

```
write(fd, pstr, strlen(pstr));
```

Начиная со следующего оператора, будут выполняться два процесса:

```
if (pid = fork() == 0)
```

На этом моменте я хочу остановиться более подробно. Системный вызов `fork()` создает дочерний процесс, родительскому процессу возвращается идентификатор PID порожденного процесса, а порожденному процессу возвращается 0. Эту особенность можно использовать для ветвления в операторе `if`, что позволяет выполнять код родительского и порожденного процессов отдельно.

Если предположить, что анализ кода завершения функции `fork()` не выполняется (отсутствует оператор `if`), то после завершения `fork()` будут выполняться две одинаковые копии одного и того же процесса!

Все операторы, находящиеся в фигурных скобках после оператора `if`, выполняются в порожденном процессе:

```
write(fd, chstr, strlen(chstr));
close(fd);
exit(0);
```

Поскольку при создании дочернего процесса ему передаются все дескрипторы открытых файлов, то можно выполнить запись данных в открытый дескриптор (функция `write()`), после чего закрыть дескриптор (системный вызов `close()`). Закрытие дескриптора файла не влияет на файловые операции в родительском

процессе, который сможет выполнять операции с файлом без его повторного открытия.

Для ожидания завершения порожденного процесса в родительском процессе используется системный вызов `wait()`.

В результате выполнения программы в файл demo1.log будут записаны строки

```
Parents' string
Childs' string
```

Выполнение программного кода в операционной системе UNIX не ограничивается на уровне процессов. Во многих случаях использование порожденных процессов может оказаться невыгодным как с точки зрения потребляемых ресурсов системы, особенно памяти, так и быстродействия. В UNIX предусмотрен еще один механизм, обеспечивающий параллельное выполнение программного кода, но уже в рамках одного и того же процесса — *механизм потоков* (threads). Потоки широко используются как функциями ядра, так и системными сервисами UNIX.

Существенным отличием потоков от процессов является то, что поток выполняется в контексте данного процесса, и для него не выделяется отдельное адресное пространство и не создаются копии таблиц текущего процесса. Файловые дескрипторы и переменные в памяти являются общими для процесса и потока, поэтому при закрытии, например, файлового дескриптора в потоке он закрывается и для процесса.

Для создания потоков во всех версиях операционной системы UNIX используется функция `pthread_create()` библиотеки C, одним из параметров которой является адрес функции потока. Функция потока, созданная с помощью `pthread_create()`, собственно и выполняет всю работу.

Схематически модель использования потоков можно представить так, как показано на рис. 2.4.

Процесс может ожидать завершения выполнения потока, выполнив функцию `pthread_join()`, а сама функция потока может завершаться обычным образом — при этом инструкция `return`

или просто достижение конца функции приводит к неявному завершению работы потока.

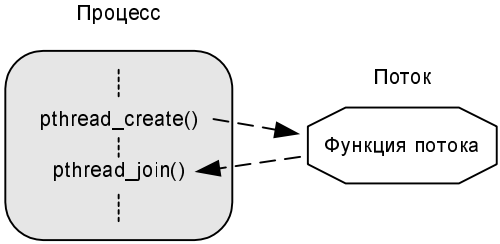


Рис. 2.4. Создание и выполнение потока

Работа с потоком продемонстрирована в следующем примере (листинг 2.2).

Листинг 2.2. Пример создания и выполнения потока

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *tFunc(void *arg)
{
 printf("Hello from thread!\n");
}

int main(void)
{
  pthread_t tchild;
  pthread_create(&tchild, NULL, &tFunc, NULL);
  pthread_join(tchild, NULL);
  return 0;
}
```

Исходный текст примера очень прост. Основной процесс создает поток с помощью функции `pthread_create()`, в качестве первого параметра которой задан адрес дескриптора нового потока `tchild`, а в качестве третьего параметра — адрес функции потока `tFunc`. После создания потока функция `tFunc` выводит сообщение на экран дисплея. Процесс ожидает завершения потока, вызвав функцию `pthread_join()`, первым параметром которой является дескриптор созданного потока.

Для компиляции данного примера требуется указать в командной строке для компилятора C++ один или несколько параметров (обычно одним из параметров является `-lpthread`), которые нужны для подключения библиотеки потоковых функций. Эти параметры зависят от конкретной реализации операционной системы, поэтому перед использованием потоковых функций следует внимательно ознакомиться с опциями используемого компилятора C++.

2.1.2. Взаимодействие процессов

Ранее мы рассмотрели, каким образом создается порожденный процесс, но при этом возникает закономерный вопрос: каким образом данные одного процесса могут быть доступны другому процессу? В операционной системе UNIX для этого предусмотрен механизм обмена, известный под названием *неименованного канала*.

Неименованный или, как часто его называют, анонимный канал создается с помощью системного вызова `pipe()`, обеспечивая коммуникации между родительским и порожденным процессами, а также между порожденными процессами одного и того же родительского процесса.

Функция `pipe()` создает файл канала, который является временным буфером и используется для того, чтобы вызывающий процесс мог записывать и считывать данные другого процесса. Канал освобождается после того, как все процессы закрывают файловые дескрипторы, которые ссылаются на данный канал.

Системный вызов `pipe()` используется в командных интерпретаторах UNIX для реализации программного канала (обозначается как `|`) с целью соединения стандартного вывода одного процесса со стандартным вводом другого (мы рассмотрим программные конвейеры при изучении командных интерпретаторов shell). Коммуникационный канал обычно реализуется как однонаправленный и устанавливается только между двумя процессами, причем один процесс является отправителем, а другой — получателем данных. Если, например, два процесса A и B нуждаются в двунаправленном коммуникационном канале, то они создадут два канала: один для записи данных процесса A в процесс B, а другой — для обратной операции.

Взаимодействие родительского и порожденного процессов посредством неименованного канала показано на рис. 2.5.

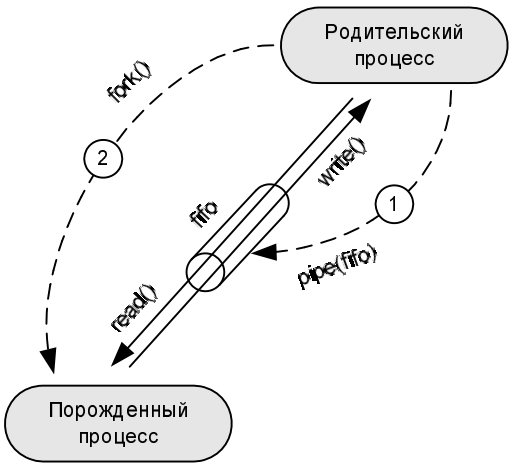


Рис. 2.5. Взаимодействие процессов посредством неименованных каналов

Вначале родительский процесс создает неименованный канал при помощи системного вызова `pipe()` (1). Созданный таким образом канал является двунаправленным, т. е. процесс может

как записывать в него данные, так и считывать их из него. Неименованный канал характеризуется двумя дескрипторами — для входных и выходных данных.

Если родительский процесс порождает новый процесс (2), то созданный процесс получает копии дескрипторов созданного канала. На рис. 2.5 показан двунаправленный канал передачи данных, в который оба процесса могут одновременно как записывать данные, так и читать их из него. Такая конфигурация, как я уже упоминал, практически не используется из-за сложностей синхронизации передачи/приема данных. Более приемлемой является конфигурация, показанная на рис. 2.6.

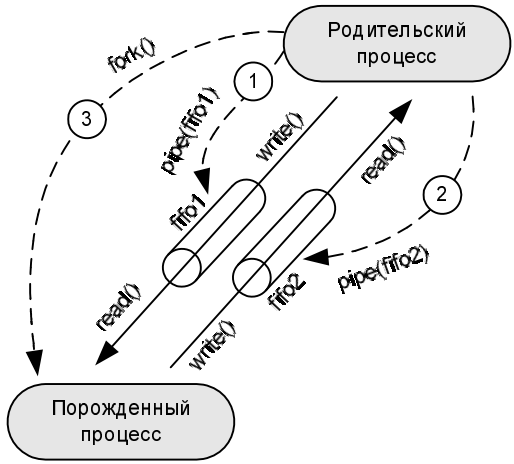


Рис. 2.6. Обмен данными при помощи двух однонаправленных каналов

Здесь основной процесс создает два однонаправленных канала с дескрипторами `fifo1` и `fifo2` при помощи двух вызовов функции `pipe()` (1, 2). При этом канал `fifo1` можно настроить так, чтобы в него мог записывать данные родительский процесс, а читал их дочерний процесс. Канал `fifo2` можно настроить для

работы в обратном направлении: записи данных дочерним и чтения данных родительским процессом.

Следует сказать, что прием/передача данных в канале синхронизируется следующим образом:

- если происходит попытка записи данных со стороны процесса в уже заполненный канал, то ядро заблокирует его до тех пор, пока другой процесс не произведет считывание из канала достаточного количества данных, чтобы заблокированный процесс смог возобновить операцию записи;
- если канал пуст, а процесс пытается прочитать из него данные, то он будет блокироваться до тех пор, пока другой процесс не запишет данные в канал.

В листинге 2.3 приведен пример исходного текста приложения, использующего передачу данных от родительского процесса дочернему.

Листинг 2.3. Передача данных по неименованному каналу

```
#include <stdio.h>
#include <sys/types.h>
#include <strings.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
  int fds[2];
  int status;
  char buf[128];
  char str[] = "Parents' process string\n";
  pid_t pid;
```

```
  if (pipe(fds) == -1)
   {
    printf("Error of creating pipe!");
    exit(0);
   }
  pid = fork();
  if (pid == 0)
   {
    bzero(buf, sizeof(buf));
    close(fds[1]);
    read(fds[0], buf, sizeof(buf));
    printf("CHILD: %s\n", buf);
    close(fds[0]);
    exit(0);
   }
  else if (pid > 0)
   {
    close(fds[0]);
    write(fds[1], str, strlen(str));
    close(fds[1]);
    wait(&status);

   }
  return 0;
}
```

Здесь основной процесс создает неименованный канал, определяемый набором дескрипторов `fds`:

```
if (pipe(fds) == -1)
```

Дескриптор `fds[0]` используется для чтения, а дескриптор `fds[1]` — для записи данных.

Затем создается дочерний процесс, реализованный в блоке операторов

```
if (pid == 0)
 {
   . . .
 }
```

который будет читать данные из канала. Поскольку запись данных в дочернем процессе не производится, мы закрываем канал для записи:

```
close(fds[1]);
```

Далее дочерний процесс будет ожидать поступления данных для чтения, вызвав функцию `read()`:

```
read(fds[0], buf, sizeof(buf));
```

По окончанию приема данных дочерний процесс закрывает дескриптор для чтения и завершает работу.

Родительский процесс выполняется в блоке операторов

```
else if (pid > 0)
 {
  . . .
 }
```

При этом канал для чтения закрывается, после чего данные записываются в канал посредством системного вызова `write`:

```
write(fds[1], str, strlen(str));
```

После вывода данных родительский процесс закрывает дескриптор канала для записи и ожидает завершения дочернего процесса.

В этом примере, как вы видите, используется один канал. Для того чтобы дочерний процесс мог записывать данные, а родительский — их читать, нужно создать еще один канал, поменяв при этом функции чтения на запись и обратно.

До сих пор мы рассматривали взаимодействие родительского и порожденного процессов. Во многих случаях процессы должны

обмениваться данными с другими процессами, независимыми от них. Для этого может использоваться несколько механизмов, которые мы вкратце рассмотрим.

Процессы могут обмениваться данными, используя общие (разделяемые) области памяти (рис. 2.7).

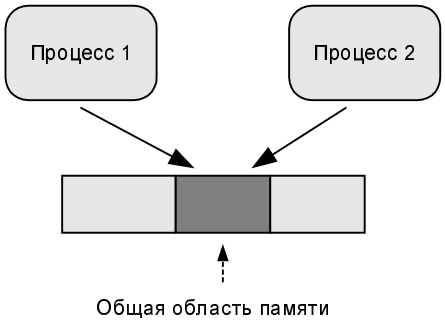


Рис. 2.7. Схема использования общей области памяти

При этом данные одного процесса могут быть доступны другому процессу. При использовании такого механизма следует учитывать то, что оба процесса могут запросить одни и те же данные, поэтому следует предусмотреть механизмы синхронизации доступа (можно использовать стандартные средства операционной системы UNIX).

Для межпроцессного взаимодействия в UNIX очень широко используется механизм так называемых *сокетов* (гнезд), который мы подробно рассмотрим при анализе сетевых возможностей операционной системы.

2.2. Системные процессы

Следующий уровень функциональности, который мы рассмотрим, — *системные процессы* (см. рис. 2.1), которые создаются при запуске системных программ. К системным программам

или, по-другому, к системному программному обеспечению относят командные оболочки shell (интерпретаторы), команды и утилиты системного администрирования, драйверы и протоколы коммуникаций. Как известно, операционная система UNIX включает ряд стандартных системных программ для выполнения задач администрирования, конфигурирования и поддержки файловой системы. Кроме того, к этой группе программ следует отнести утилиты:

- настройки параметров конфигурации системы;
- перекомпоновки ядра (если она необходима) и добавления новых драйверов устройств;
- создания и удаления учетных записей пользователей;
- создания и подключения физических файловых систем;
- установки параметров контроля доступа к файлам.

В качестве пользовательских программ могут выступать командные файлы (скрипты), написанные с помощью командного интерпретатора shell, или разработанные на одном из языков высокого уровня (C, Pascal, Fortran) приложения. Командные интерпретаторы представляют собой высокоуровневую среду программирования и очень удобны для создания собственных командных файлов (по аналогии с MS-DOS), позволяющих разрабатывать довольно сложные программы. Возможности командных интерпретаторов намного превышают те, которые имеет, например, интерпретатор командной строки MS-DOS, что делает их полноценным инструментом разработки программ.

Системные процессы являются частью ядра и всегда расположены в оперативной памяти. Они не имеют выполняемых файлов и запускаются особым образом при инициализации ядра системы. Исполняемые инструкции и данные таких процессов постоянно находятся в ядре системы, поэтому они могут вызывать другие функции и обращаться к данным, недоступным для остальных процессов. К системным процессам относится и процесс начальной инициализации `init`, являющийся прародителем всех остальных процессов. Несмотря на то, что `init` не является ча-

стью ядра и запускается из выполняемого файла, его функционирование критически важно для всей системы в целом.

Программа `/sbin/init`, запускающая процесс `init`, порождает процессы для запуска системы на основе записей, находящихся в файле /etc/inittab. При этом процесс `init` анализирует записи в файле /etc/inittab и определяет последовательность запуска, остановки и перезапуска остальных процессов. Мы рассмотрим процесс `init` более подробно в следующих главах.

Еще одна группа процессов, выполняющихся в системе, относится к *прикладным процессам*. Как правило, это процессы, созданные в контексте пользовательского сеанса работы, важнейшим из которых является командный интерпретатор shell, обеспечивающий выполнение команд пользователя в системе UNIX. Пользовательские процессы могут выполняться как в интерактивном (приоритетном), так и в фоновом режимах. Интерактивные процессы монопольно владеют терминалом, и пока такой процесс не завершит свое выполнение, пользователь не имеет доступа к командной строке.

На этом анализ основных аспектов функционирования операционной системы UNIX можно закончить и приступить к более детальному рассмотрению отдельных подсистем.

Глава 3

Файловая система

Организация данных в операционных системах определяется структурой *файловой системы*. Файловая система является одной из важнейших функциональных частей UNIX и обеспечивает:

- хранение данных, принадлежащих операционной системе и пользователям, а также обеспечение их целостности;
- эффективный доступ к данным, находящимся на запоминающих устройствах длительного хранения (накопителях на жестких и оптических дисках, магнитных лентах и т. д.);
- эффективное выполнение операций восстановления данных в случае их повреждения;
- единообразный механизм доступа ко всем объектам файловой системы.

Файловая система UNIX устроена таким образом, чтобы соответствовать одной из основных концепций этой операционной системы — представлению всех объектов операционной системы, независимо от их природы, в виде файлов. Здесь привычное для многих пользователей понятие файла приобретает более широкий смысл.

В привычных для многих операционных системах MS-DOS и в ранних версиях Windows термином "файл" обозначался двоичный образ данных, записанных на диск. В UNIX к файлам относятся не только дисковые файлы или каталоги, но также и программные объекты — именованные и неименованные кана-

лы, сокеты, терминальные линии, а также физические устройства ввода/вывода, такие, например, как накопители на гибких и жестких дисках, параллельный и последовательный порты и т. д. При этом устройства ввода/вывода представлены специальными файлами, которые имеют название *файлов устройств*.

Подобное представление означает, что ко всем объектам файловой системы можно обращаться, используя стандартный программный интерфейс, предоставляемый UNIX. Например, к дисковому файлу, именованному каналу или параллельному порту можно обращаться, используя системные вызовы `open()`, `read()`, `write()` и `close()`. Сказанное касается всех пользовательских и части системных программ, для которых собственно и создана такая модель файловой системы. Операционная система UNIX на уровне ядра и драйверов устройств обрабатывает запросы к разным устройствам, дифференцируя их типы и используя различные подходы.

Есть еще одна причина, по которой выбрана именно такая архитектура файловой системы, — это необходимость обеспечить надежность системы. Напомню, что в UNIX пользовательские приложения не могут обращаться напрямую к аппаратным устройствам иначе как через системные вызовы, поэтому разработчики операционной системы использовали один и тот же программный интерфейс как для дисковых файлов, так и для устройств ввода/вывода.

Лучше понять механизм взаимодействия пользовательского приложения и объекта файловой системы UNIX позволяет рис. 3.1.

Здесь показана схема обращения из программы пользователя к двум объектам файловой системы:

- параллельному порту компьютерной системы (файл устройства `/dev/lp0`), к которому подсоединен принтер, для чего используется системный вызов `open()`:

```
open("/dev/lp0", O_RDWR);
```

- дисковому файлу `file`:

```
open("file", O_RDONLY);
```

Рис. 3.1. Схема взаимодействия приложения пользователя и объектов файловой системы

Для программы пользователя обращение к разным по природе типам устройств (параллельный порт и жесткий диск) прозрачно, т. е. программно они различаются лишь именами устройств. В то же самое время для операционной системы оба этих системных вызова могут обрабатываться различными способами,

поскольку оба устройства управляются различными драйверами, оперирующими как различными типами данных (символьными и блочными), так и различными аппаратными интерфейсами.

Подобная структура файловой системы очень удобна для разработчиков программного обеспечения, поскольку она обеспечивает унифицированный программный интерфейс для работы с объектами файловой системы.

Рассмотрим более подробно основные типы файлов, используемые в UNIX. К ним относятся бинарные файлы, содержащие двоичные данные (например, программы или текст), записанные на жесткий диск, специальные файлы устройств, сокеты и именованные каналы, а также символические и жесткие ссылки.

Бинарные файлы содержат наборы двоичных битов, в которых в закодированном виде находится та или иная информация, например, текст, рисунки, программы, аудиоданные и т. д. В большинстве случаев, когда используют термин "файл", то имеют в виду именно такие файлы. Данные, записанные в такие файлы, должны интерпретироваться или операционной системой (если это программный файл), или другими приложениями.

Файлы устройств позволяют операционной системе UNIX и другим программам взаимодействовать с аппаратными средствами и периферийными устройствами системы. Файлы устройств не эквивалентны драйверам устройств — драйверы обеспечивают доступ к устройству на уровне аппаратно-программного интерфейса, преобразуя пакеты запросов, поступающие от ядра в соответствующие инструкции процессора. Файлы устройств можно представить как шлюзы, через которые драйвер получает запросы. Когда ядро получает запрос к файлу устройства, оно просто передает этот запрос соответствующему драйверу.

Структура файла устройства отличается от той, которую имеет файл данных. Сами файлы устройств обрабатываются базовыми средствами файловой системы, а их характеристики записываются на диск. Взаимодействие пользовательской программы, файла устройства и драйвера было показано на рис. 3.1 (обращение к параллельному порту, которому соответствует файл устройства `/dev/lp0`).

Файлы устройств могут иметь один из двух типов: файл байт-ориентированного ("символьного") или блок-ориентированного ("блочного") устройств. Файлы байт-ориентированных устройств позволяют связанным с ними драйверам выполнять собственную буферизацию ввода/вывода, в то время как файлы блок-ориентированных устройств обрабатываются драйверами, манипулирующими большими блоками данных и возлагающими буферизацию на ядро. Некоторые типы аппаратных средств, например, накопители на жестких дисках и магнитных лентах, представляются файлами обоих типов.

Поскольку в системе могут присутствовать устройства одного типа, то файлы устройств распознаются по двум номерам — старшему и младшему. Старший номер устройства информирует ядро, к какому драйверу относится данный файл, а младший номер сообщает драйверу, к какому физическому устройству следует обращаться. Так, например, старший номер устройства 6 в Linux обозначает драйвер параллельного порта. Первый параллельный порт `/dev/lp0` будет иметь старший номер 6 и младший номер 0.

Некоторые драйверы используют младший номер устройства нестандартным способом. Например, драйверы накопителей на магнитных лентах часто руководствуются им при выборе плотности записи, а также определяют, нужна ли перемотка ленты после закрытия файла устройства. В некоторых системах драйвер терминала, управляющий последовательными устройствами, использует младшие номера устройств для идентификации модемов.

Еще один тип файла UNIX — *сокет* (гнездо). Чаще всего сокеты используются для взаимодействия между независимыми процессами, выполняющимися на одной и той же (локальные сокеты) или на разных системах (сокеты протокола TCP). Сокеты TCP позволяют взаимодействовать процессам, выполняющимся на разных машинах в сетях TCP/IP, хотя могут применяться для обмена данными между процессами, работающими на одном и том же хосте (в этом случае используется так называемый *интерфейс обратной связи*, который часто называют loopback interface, он имеет IP-адрес 127.0.0.1).

Взаимодействие процессов посредством сокетов очень широко применяется в самой операционной системе. Например, система печати, система X Window и система Syslog активно используют сокеты для обмена данными. Более детально взаимодействие процессов посредством сокетов мы рассмотрим при анализе сетевого взаимодействия, сейчас же замечу, что сокеты создаются с помощью системного вызова `socket()`, а закрываются только при закрытии соединения с обеих сторон.

К объектам файловой системы UNIX относят *именованные каналы*. Так же как и локальные сокеты, именованные каналы позволяют взаимодействовать процессам, выполняющимся на одной машине. Именованные каналы создаются командой `mknod`, а удаляются командой `rm`.

К отдельному типу объектов файловой системы можно отнести *символические* и *жесткие ссылки*, которые служат для обеспечения альтернативных способов обращения к файлам.

Большинство операционных систем UNIX поддерживает несколько типов файловых систем. Кроме базовой версии, берущей начало от FreeBSD 4.3, поддерживаются и другие файловые системы, например, обладающие повышенной надежностью или упрощенными средствами восстановления после сбоев (HP-UX), а также системы, поддерживающие другую семантику (Solaris). Большинство современных файловых систем имеет определенные отличия от базовой версии, поэтому в данной главе описывается обобщенный тип файловой системы.

3.1. Иерархия файловой системы

Файловые системы UNIX располагаются, как правило, на жестких дисках, каждый из которых состоит из одной или нескольких логически связанных групп цилиндров, называемых разделами (partitions). Физическое расположение и размер раздела устанавливаются при форматировании диска. В UNIX-системах разделы являются независимыми, доступ к ним осуществляется как к различным носителям данных, при этом

один раздел содержит, как правило, только одну физическую файловую систему.

Файловую систему можно рассматривать, с одной стороны, как логическую структуру в виде дерева каталогов и файлов с четко установленной иерархией. Именно в таком виде и представляется файловая система UNIX пользователю. С другой стороны, файловая система — это совокупность расположенных на физическом носителе упорядоченных и неупорядоченных двоичных данных. Пользователь обычно не имеет доступа непосредственно к блокам данных на носителях, хотя и может управлять ими при помощи программ, в которых используются соответствующие системные вызовы. Управление физической файловой системой — прерогатива функций ядра, поэтому вмешательство в этот процесс нежелательно, даже если вы хорошо представляете себе, что делаете.

В операционных системах UNIX используются различные типы файловых систем, каждая из которых имеет свои особенности, но операционная система и программы пользователей имеют дело не с физическими блоками на жестком диске, а с логическими структурами данных, которые образуют логическую файловую систему. В дальнейшем, если особо не оговорено, будет использоваться термин "файловая система" — при этом будет подразумеваться именно логическая файловая система.

Основой любой файловой системы является *корневой каталог* (обозначается как /). Корневой каталог, как и другие каталоги, является обычным файлом, содержащим информацию о других файлах, позволяя четко структурировать объекты файловой системы. Все остальные каталоги и файлы располагаются в рамках структуры, порожденной корневым каталогом (в нем и в его подкаталогах), независимо от их физического местонахождения. Структуру файловой системы UNIX можно представить себе так, как показано на рис. 3.2.

Совокупность имен каталогов, через которые проходит путь к заданному файлу, образует, вместе с именем этого файла включительно, *путевое имя*. Путевые имена могут быть *абсолютными* (например, /tmp/myfile) или *относительными* (например, local/

filesystem), при этом относительное имя указывается по отношению к текущему каталогу. Максимальный размер имени каталога не должен превышать 255 символов, а отдельное путевое имя не должно быть более 1023 символов.

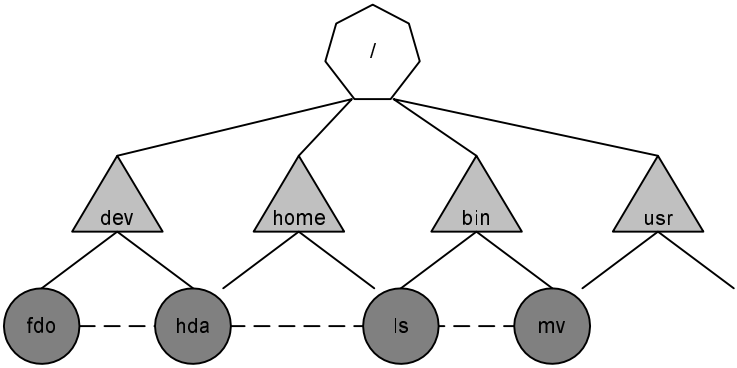


Рис. 3.2. Структура файловой системы UNIX

Для корневого каталога обязательно создается отдельная физическая файловая система, а сам он является точкой ее монтирования, о чем свидетельствует наличие подкаталога lost+found. Корневой каталог должен быть всегда доступен и монтируется автоматически при запуске системы, поэтому с формальной точки зрения все остальные физические файловые системы для функционирования UNIX не нужны.

Большинство операционных систем придерживается более-менее стандартной структуры каталогов файловой системы, устанавливая для них предопределенные назначения, хоть это и не обязательно:

- /bin — содержит программы пользователей — в большинстве современных систем обычно является символической ссылкой на каталог /usr/bin;
- /dev — каталог для специальных файлов устройств — обычно содержит подкаталоги для различных классов и типов устройств, например, dsk, rdsk, rmt, inet (в System V);

- /etc — каталог для конфигурационных файлов — может включать подкаталоги для различных компонентов и сервисов;
- /home — каталог, в котором размещаются начальные каталоги пользователей — во многих случаях к нему монтируется (присоединяется) отдельная физическая файловая система;
- /lib — каталог для библиотек — обычно является символической ссылкой на /usr/lib;
- /lost+found — этот подкаталог содержится в каждом каталоге, являющемся точкой монтирования физической файловой системы;
- /mnt — точка монтирования для файловых систем на съемных носителях или дополнительных дисках (например, CD-ROM или флоппи-диски);
- /opt — каталог для дополнительного программного обеспечения (может быть пустым или отсутствовать);
- /proc — каталог псевдо-файловой системы, предоставляющей в виде каталогов и файлов информацию о ядре, памяти и процессах, работающих в системе;
- /sbin — каталог для системных программ, необходимых для системного администрирования UNIX;
- /tmp — каталог для временных файлов;
- /usr — в этом каталоге находятся программы, библиотеки, заголовочные файлы, справочные руководства (/usr/share/man), исходные тексты ядра и утилит системы (в Linux), файлы очереди печати (/usr/spool в BSD-системах) и т. д. Часто каталог является точкой монтирования отдельной физической файловой системы. В этом каталоге могут находиться следующие подкаталоги:
  - /usr/bin — основные программы и утилиты операционной системы;
  - /usr/include — заголовочные файлы библиотек функций. Здесь же могут находиться подкаталоги;

- /usr/lib — статически и динамически компонуемые библиотеки; может содержать подкаталоги;
- /usr/local — каталог для дополнительного свободно распространяемого программного обеспечения. Данный подкаталог содержит иерархию подкаталогов, аналогичную корневому (bin, etc, include, lib и т. д.);

❒ /var — в UNIX System V и Linux этот каталог является аналогом каталога, используемого для хранения файлов различных системных сервисов, например, файлов журналов системы.

Следует сказать, что использование других каталогов верхнего уровня и подкаталогов зависит от версии операционной системы UNIX, установленного системного и прикладного программного обеспечения, а также от изменений, внесенных в базовую конфигурацию пользователем.

В операционной системе UNIX путевое имя файла принято называть *жесткой ссылкой*, причем для большинства файлов имеется всего лишь одна жесткая ссылка, хотя при помощи команды `ln` пользователь может создать дополнительные жесткие ссылки для каждого из файлов.

Например, для файла с путевым именем /home/user1/text можно создать жесткую ссылку /home/user1/text.new, выполнив команду

```
ln /home/user1/text /home/user1/text.new
```

После этого при любых операциях с данным файлом к нему можно обращаться по этой ссылке.

Другой тип ссылки — *символическая*, или, как ее называют по-другому, "мягкая" ссылка — позволяет вместо путевого имени файла указывать псевдоним. При обнаружении символической ссылки, например, во время поиска файла, ядро извлекает из нее путевое имя. Жесткая ссылка отличается от символической тем, что она указывает непосредственно на индексный дескриптор файла, т. е. на физическое расположение первого блока файла, в то время как символическая ссылка указывает на файл по его имени.

Файл, адресуемый символической ссылкой, и сама ссылка являются физически разными объектами файловой системы.

Символическая ссылка создается командой `ln -s`, а удаляется командой `rm`. Такая ссылка может указывать на файл, находящийся в другой файловой системе, или даже на несуществующий файл. В рассмотренном ранее примере можно жесткую ссылку заменить символической:

```
ln -s /home/user1/text /home/user1/text.new
```

3.2. Функции API для работы с файлами

В операционные системы UNIX включен целый ряд функций прикладного интерфейса программирования API для работы с объектами файловой системы или, по-другому, системных вызовов для выполнения файловых операций. В этой и последующих главах при использовании функций API мы будем применять только те функции, которые являются общими для всех реализаций UNIX, т. е. придерживаться общих спецификаций, изложенных в стандарте POSIX. Стандарт POSIX предлагает стандартный интерфейс прикладного программирования операционных систем, в котором определены функции API для манипулирования файлами и процессами. Например, системный вызов `fork()`, создающий новый процесс и рассмотренный нами ранее, также включен в этот стандарт.

Если особо не оговорено, все дальнейшие рассуждения будут проводиться применительно к спецификациям POSIX.

Системные вызовы для работы с объектами файловой системы сгруппированы в табл. 3.1.

Таблица 3.1. *Системные вызовы UNIX для манипуляций файлами*

Функция API	Назначение
`open()`	Открывает файл для доступа к данным
`read()`	Чтение данных из файла
`write()`	Запись данных в файл

Таблица 3.1 *(окончание)*

Функция API	Назначение
`lseek()`	Позиционирование в открытом файле
`close()`	Закрытие дескриптора открытого файла
`stat()`, `fstat()`	Получение атрибутов файла
`chmod()`	Изменение прав доступа к файлу
`chown()`	Изменение идентификатора пользователя (uid) или группы (gid) для файла
`utime()`	Изменение даты и времени последнего изменения содержимого файла и последнего доступа к нему
`link()`	Создание жесткой ссылки на файл
`unlink()`	Удаление жесткой ссылки на файл
`umask()`	Установка маски

В большинстве команд операционной системы UNIX использованы эти функции, причем иногда мнемонические обозначения совпадают. Например, на основе функций API `link()` и `unlink()` созданы одноименные команды UNIX. То же самое касается, например, команд `chmod` и `chown`, в основе которых лежат одноименные функции API.

Вкратце рассмотрим синтаксис наиболее часто используемых функций API: `open()`, `read()`, `write()` и `close()`. Начнем с системного вызова `open()`.

Функция `open()` устанавливает соединение между процессом и файлом, позволяя как создавать новые, так и открывать существующие файлы для выполнения операций чтения и/или записи. Открытие файла означает получение дескриптора (описателя) на данный файл, который будет в дальнейшем использоваться во всех операциях с файлом. *Дескриптор* (handle) представляет собой беззнаковое целое число и используется операционной системой для доступа к файлу. Нельзя работать с данными файла, не открыв его и не получив дескриптор.

Очень часто при определении операции записи или чтения говорят "запись в дескриптор файла" или "чтение из дескриптора файла", имея в виду открытый файл, для которого получен дескриптор. По завершению всех операций с файлом дескриптор следует закрыть (в принципе, операционная система по завершению процесса закрывает все открытые дескрипторы, принадлежащие ему, но это плохая практика — оставлять открытыми дескрипторы файлов).

Функция `open()` имеет прототип:

```
#include <sys/types.h>
#include <fcntl.h>

int open(const char* path, int access_mode, mode_t
permission);
```

Смысл параметров функции следующий:

- *path* — имя файла, которое может быть абсолютным (символьная строка, начинающаяся символом /), относительным (символьная строка, не начинающаяся символом /) или символической ссылкой;
- *access_mode* — целочисленное значение, показывающее, какие типы доступа к файлу разрешены вызывающему процессу. Все типы доступа определены как макросы в файле fcntl.h и могут иметь одно из следующих значений:
  - `O_RDONLY` — файл доступен только для чтения;
  - `O_WRONLY` — файл доступен только для записи;
  - `O_RDWR` — файл доступен для чтения и для записи.

  Кроме того, можно задавать один или несколько указанных далее модификаторов, логически складывая их с указанными флагами доступа, что расширяет или изменяет механизм доступа:
  - `O_APPEND` — позволяет добавить данные в конец файла;
  - `O_CREAT` — позволяет создать файл, если он не существует;

- O_TRUNC — отбрасывает содержимое файла, устанавливая его размер равным 0;

- *permission* — необходим только в том случае, если в параметре *access_mode* присутствует флаг O_CREAT. Этот параметр задает права доступа к файлу для владельца, группы-владельца и остальных пользователей.

Пример вызова функции open() приведен далее:

```
int fd = open("/home/user1/text", O_RDWR | O_CREAT |
O_APPEND);
```

Здесь функция open() открывает файл /home/user1/text для чтения/записи, при этом, если файл не существует, он создается, а для существующего файла данные будут добавлены в его конец.

Функция возвращает дескриптор fd, который помещается в таблицу дескрипторов файлов процесса и будет использоваться при последующих операциях.

Функция read() читает блок данных указанного размера из файла с заданным дескриптором. Прототип функции выглядит так:

```
#include <sys/types.h>
#include <unistd.h>

ssize_t read(int fd, void* buf, size_t size);
```

Параметры функции означают следующее:

- *fd* — дескриптор открытого файла, полученный ранее с помощью системного вызова open() или иным способом;
- *buf* — буфер данных, куда помещаются данные, считанные из файла;
- *size* — количество байтов, которое необходимо прочитать из файла (тип этого параметра эквивалентен unsigned int).

Функция read() возвращает количество байтов, прочитанных в буфер памяти *buf*.

Вот пример использования функции:

```
char buf[128];
int bytesRead;
bytesRead = read(fd, buf, sizeof(buf));
```

В этом примере функция `read()` пытается прочитать 128 байтов в буфер `buf` с открытого файла, указанного дескриптором `fd`. В переменную `bytesRead` будет помещено количество считанных байтов.

Функция `write()` записывает блок данных фиксированного размера в файл, дескриптор которого задается в качестве первого параметра функции.

Прототип функции выглядит так:

```
#include <sys/types.h>
#include <unistd.h>

ssize_t write(int fd, const void* buf, size_t size);
```

Параметры функции имеют следующий смысл:

- *fd* — дескриптор открытого файла, полученный при вызове функции `open()` или иным способом;
- *buf* — буфер данных, откуда выбираются данные;
- *size* — количество записываемых байтов.

Следующий пример показывает использование функции `write()`:

```
char *str = "String to write";
write(fd, str, strlen(str));
```

Здесь в открытый файл с дескриптором `fd` записывается строка `str`, размер которой определяется функцией `strlen(str)`.

Функция `close()` закрывает открытый дескриптор файла, принимая значение дескриптора в качестве единственного параметра.

Анализ работы файловых функций, который мы только что провели, поможет лучше понять, как выполняются операции с объектами файловой системы в UNIX, которые будут рассмотрены в следующем разделе.

3.3. Операции с файлами. Индексные дескрипторы

Любой файл операционной системы UNIX описывается информационным блоком, который называется i-node (индексный дескриптор). Это исключительно важная структура, поскольку повреждение или некорректная информация, помещенная в индексный дескриптор, фактически означает уничтожение файла. Все операции с объектами файловой системы осуществляются только через индексные дескрипторы, поэтому рассмотрим эту информационную структуру подробно.

Если файл открыт, то операционная система создает копию индексного дескриптора в памяти, в то время как исходный дескриптор хранится на диске. В индексном дескрипторе находится вся информация о файле, за исключением его имени, которое хранится в каталоге, где размещен файл. Индексный дескриптор содержит следующие характеристики файла:

- атрибуты доступа и тип файла;
- информацию о владельце файла;
- время последнего изменения, последнего доступа и последней модификации;
- счетчик жестких ссылок;
- размер файла в байтах;
- адреса физических блоков на жестком диске.

Структура информационных полей в индексном дескрипторе одинакова для всех операционных систем UNIX и состоит из 16-ти 32-разрядных значений (рис. 3.3).

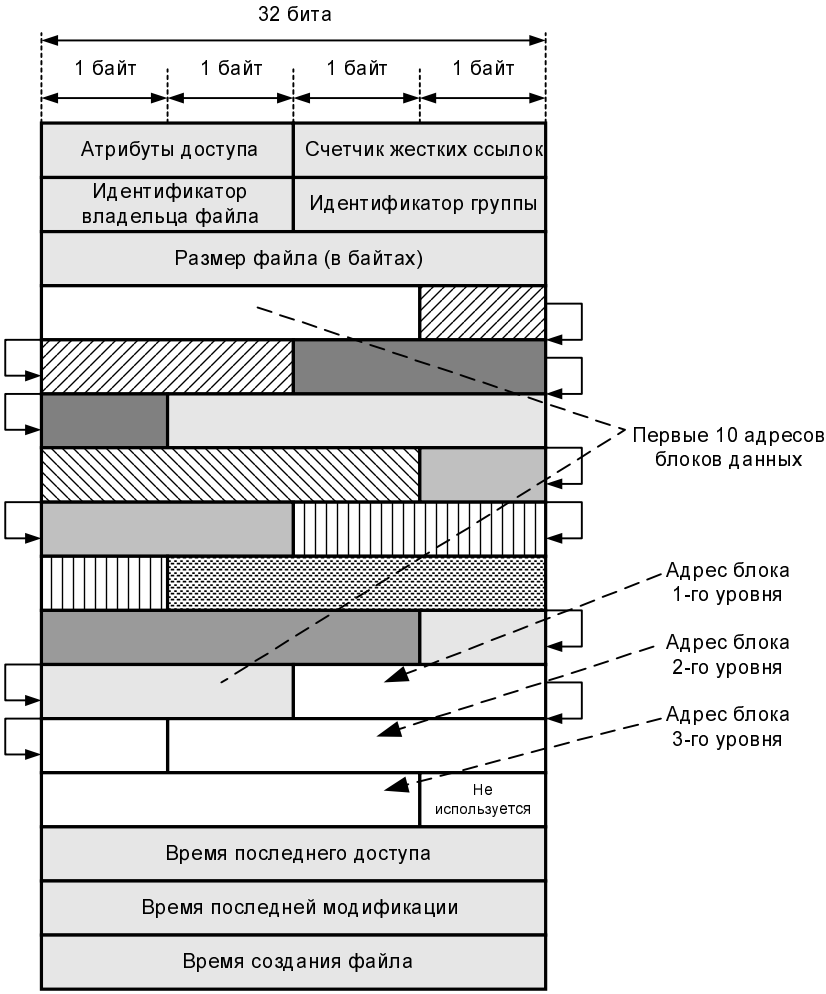


Рис. 3.3. Индексный дескриптор файла

Индексный дескриптор содержит 13 адресов физических блоков, каждый из которых занимает 3 байта (в последних версиях FreeBSD и Linux операционной системы UNIX для адресации используется 4 байта). Первые 10 адресов блоков непосредст-

венно ссылаются на блоки данных, а оставшиеся 3 содержат адреса индексных блоков, которые, в свою очередь, ссылаются на следующие блоки данных.

Предполагается, что все данные файла, размещенные по физическим адресам, указанным в индексном дескрипторе, находятся на одном и том же физическом диске.

Можно подсчитать, какой максимальный размер может иметь файл при использовании такого индексного дескриптора.

Предположим, что размер физического блока данных файла составляет 512 байтов. В этом случае первые 10 адресов обеспечат хранение 512 × 10 = 5120 байтов данных. Каждый из 3-х индексных блоков также имеет размер 512 байтов и может содержать 170 3-байтовых адресов. Таким образом, 1-й, 2-й и 3-й индексные блоки дадут 170 × 170 × 170 × 512 = 2 515 456 000 байтов, т. е. максимальный размер файла может составлять приблизительно 2,5 Гбайт (2 530 344 960 байтов).

Если файл открывается для выполнения операции, ядро помещает в память копию индексного дескриптора из таблицы индексных дескрипторов файлов. Такая копия "в памяти" помимо стандартной информации, рассмотренной ранее, содержит несколько дополнительных полей:

- счетчик ссылок (reference count), показывающий количество одновременно открытых копий данного файла;
- статусную информацию, указывающую на такие состояния:
  - индексный дескриптор блокирован;
  - процесс ожидает разблокирования;
  - индексный дескриптор, находящийся в памяти, отличается от версии на диске (dirty);
  - выполнены какие-то модификации файла, не сохраненные к настоящему моменту на диске;
- номер дискового устройства, где расположен файл.

Обратите внимание на то, что в индексном дескрипторе не указывается имя файла — операционная система помещает имя

файла вместе с номером индексного дескриптора (i-number) в каталог, где располагается файл.

Для более ранних версий операционных систем UNIX каждая запись каталога состоит из 16 байтов, первые два из которых указывают на номер индексного дескриптора, а остальные 14 содержат имя файла. Поскольку имя файла было ограничено 14-ю символами, в современных версиях UNIX запись в каталоге имеет следующий формат (рис. 3.4).

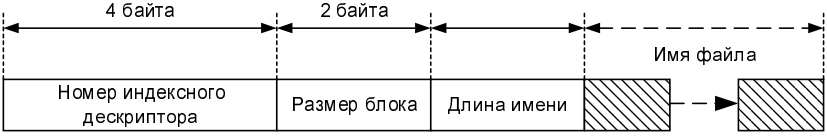


Рис. 3.4. Структура записи в каталоге

В операционной системе UNIX для каждой файловой системы создается *таблица индексных дескрипторов*, в которой хранится информация обо всех файлах. Каждая запись в такой таблице содержит индексный дескриптор и его номер. Например, если ядру понадобится получить доступ к информации о файле, индексный дескриптор которого имеет номер 69, то оно будет просматривать все записи таблицы индексных дескрипторов в поисках записи, содержащей индексный дескриптор с номером 69.

Так как номер индексного дескриптора уникален только в пределах одной файловой системы, то запись в этой таблице идентифицируется как по номеру индексного дескриптора, так и по идентификатору файловой системы, который присваивается файловой системе при ее монтировании командой `mount`.

Взаимосвязь всех структур данных лучше всего показать на примере создания файла. Предположим, что программа должна создать файл с именем test в каталоге /home/user. Последовательность шагов для выполнения этой задачи и действия операционной системы показаны на рис. 3.5.

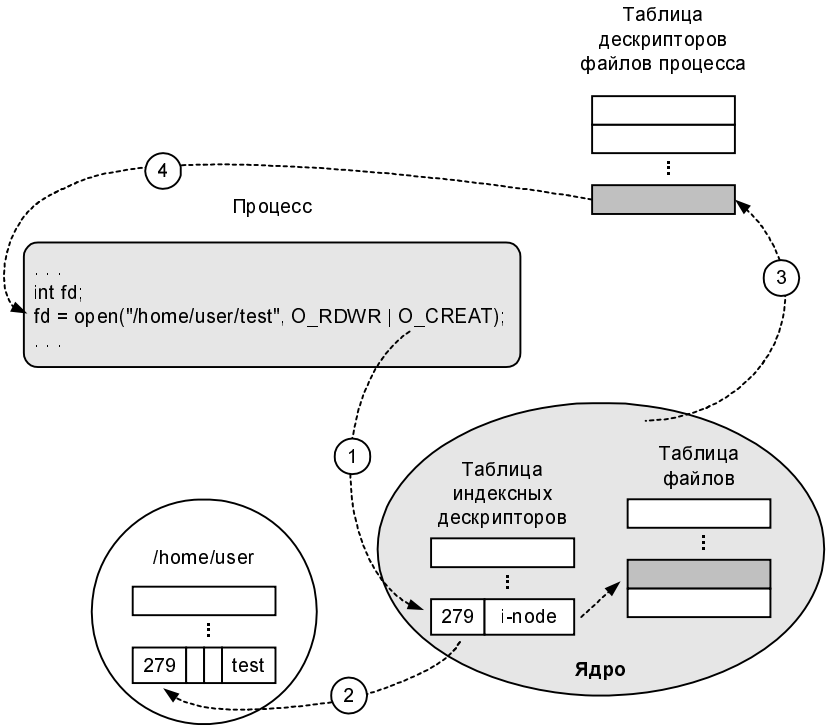


Рис. 3.5. Схема операций при создании файла в операционной системе UNIX

Для лучшего понимания, как выполняются файловые операции в UNIX, проанализируем схему, показанную на рис. 3.5 (с целью упрощения изложения предполагаем, что все операции выполняются без ошибок):

1. Выполняющийся процесс делает попытку создания файла test в каталоге /home/user при помощи системного вызова

   ```
   fd = open("/home/user/test", O_RDWR | O_CREAT);
   ```

   Ядро UNIX создает новую запись в таблице индексных дескрипторов и присваивает вновь созданному дескриптору уни-

кальный номер (в данном примере номер индексного дескриптора равен 279). Эти действия выполняются на этапе (1).

2. Ядро добавляет номер дескриптора и имя файла в каталог /home/user (2).
3. Ядро ищет в таблице дескрипторов файлов процесса первую незадействованную позицию. Если такая позиция есть, то она будет использована для обращения к файлу (3). Кроме этого, в системную таблицу файлов заносится ссылка на запись в таблице индексных дескрипторов, содержащая данные по открытому файлу.
4. Ядро возвращает процессу номер (индекс) позиции в таблице дескрипторов файлов процесса в качестве дескриптора открытого файла (4). Значение этого дескриптора присваивается переменной `fd`.

Кроме этих действий, ядро выполняет и целый ряд других операций. Так, значение счетчика ссылок в индексном дескрипторе файла, точнее, в его копии, загруженной в память, увеличивается на 1. Аналогично увеличивается и значение счетчика ссылок в системной таблице файлов. Кроме этого, в таблицу файлов заносится информация о режиме, в котором открыт файл (в нашем случае файл открыт в режиме чтения/записи и если отсутствует на диске, то создается — это определяется вторым параметром функции `open()`, который равен `O_RDWR | O_CREAT`).

Наконец, в записи таблицы файлов формируется указатель текущей позиции в открытом файле. Этот указатель представляет собой смещение относительно начала файла позиции, начиная с которой будет происходить чтение/запись данных.

Рассмотренный нами процесс создания файла, естественно, является весьма упрощенным, тем не менее, он дает некоторое представление о принципах выполнения файловых операций в UNIX-системах.

Каждый файл в операционной системе UNIX характеризуется набором *свойств* или, по-другому, *атрибутов*. Мы уже сталкивались с ними при рассмотрении индексных дескрипторов, а сейчас проанализируем атрибуты файлов более подробно.

Набор общих атрибутов файлов представлен в табл. 3.2.

Таблица 3.2. *Атрибуты файлов*

Атрибут	Значение
file type	Тип файла
access permission	Права доступа к файлу для владельца, группы и остальных пользователей
hard link count	Количество жестких ссылок на файл
uid	Идентификатор владельца файла
gid	Идентификатор группы, к которой принадлежит владелец файла
file size	Размер файла в байтах
last access time	Время последнего доступа
last modify time	Время последней модификации
last change time	Время последнего изменения
inode number	Номер индексного дескриптора
file system ID	Идентификатор файловой системы

Перечисленные в таблице атрибуты используются ядром для управления файлами. Например, при попытке доступа какого-либо пользователя к файлу ядро сравнивает его идентификаторы с идентификаторами uid и gid файла, чтобы установить какая категория разрешений должна быть задействована.

Все файлы имеют атрибуты, перечисленные в табл. 3.2, хотя, например, для файлов устройств атрибут "file size" не имеет смысла и всегда установлен в 0. Дополнительно для файлов устройств устанавливаются такие атрибуты, как старший и младший номера устройств.

Атрибуты назначаются файлу ядром в момент его создания и могут оставаться неизменными в течение всего времени его существования. Некоторые атрибуты (тип файла, номер индексно-

го дескриптора, идентификатор файловой системы, старший и младший номера устройств (для файлов устройств)) остаются неизменными, другие можно изменить либо программно, либо используя команды UNIX.

Для получения информации о файле программным способом можно воспользоваться системным вызовом `stat()`. В листинге 3.1 приведен исходный текст программы на C, в которой на экран дисплея выводится размер файла, номер индексного дескриптора и значение счетчика жестких ссылок.

Листинг 3.1. Получение атрибутов файла при помощи функции `stat()`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
  struct stat statv;
  if (argc != 2)
   {
     printf("Usage: %s path\n", argv[0]);
     exit(0);
   }
  if (!stat(argv[1], &statv))
   {
     printf("---------- %s attributes----------\n",
argv[1]);
     printf("Size: %d\n", (int)statv.st_size);
     printf("i-number: %d\n", (int)statv.st_ino);
```

```
    printf("Ref.count: %d\n", (int)statv.st_nlink);
  }
 return 0;
}
```

В качестве единственного параметра программа принимает путевое имя файла (`argv[1]`). После вызова функции `stat()` на экран дисплея выводятся размер файла в байтах, номер индексного дескриптора и количество жестких ссылок на данный файл. Все атрибуты файла помещаются в структуру `statv`, адрес которой является вторым параметром функции `stat()`.

Например, для файла test, находящегося в каталоге приложения, были получены такие результаты (исходный текст скомпилирован в исполняемый файл stat_demo):

```
# ./stat_demo test
---------- test attributes-----------
Size: 30
i-number: 123880
Ref.count: 1
```

Обратите внимание на количество жестких ссылок — оно равно 1, т. е. для обычного файла без жестких ссылок счетчик ссылок всегда равен 1. Если задать для файла test жесткую ссылку, например:

```
# ln test test_hard_link
```

то после запуска программы stat_demo получим следующий результат:

```
# ./stat_demo test
---------- test attributes-----------
Size: 30
i-number: 123880
Ref.count: 2
```

Как видно из результата, количество жестких ссылок равно 2 — фактически мы имеем два имени для одного файла: test и test_hard_link. Если бы мы создали символическую ссылку, счетчик ссылок остался бы равным 1, поскольку символическая ссылка представляет собой объект файловой системы со своим индексным дескриптором.

Содержимое каталога можно просмотреть программным способом, используя две библиотечные функции C — `opendir()` и `readdir()`. Кроме этого, следует объявить в программе структуру типа `dirent` (для систем, совместимых с System V) или `direct` (для систем, совместимых с FreeBSD). В этом конкретном случае программа работает в операционной системе Solaris, поэтому используется `dirent`.

Структура (`dirent` или `direct`) содержит поля "Номер индексного дескриптора" и "Имя файла", которые присутствуют в каждой записи каталога (см. рис. 3.4).

Следующая программа (назовем ее opendir_demo), исходный текст которой представлен в листинге 3.2, выводит на экран дисплея имена файлов текущего каталога и номера их индексных дескрипторов.

Листинг 3.2. Получение содержимого записей текущего каталога

```
#include <stdio.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/types.h>

int main(void)
{
 struct dirent *dp;
 DIR* dir;
 const char* path_dir = ".";
```

```
 dir = opendir(".");
 while (dp = readdir(dir))
  {
    printf("%15s:\t i-number = %d\n", dp->d_name, dp->d_ino);
  }
 closedir(dir);
 return 0;
}
```

Здесь для работы с открытым каталогом служит указатель `DIR*`, возвращаемый функцией `opendir()`, единственным параметром которой является путь к текущему каталогу.

Далее в цикле `while` читается содержимое записей каталога, которое помещается в структуру `dp` функцией `readdir()`. Цикл заканчивается, когда `readdir()` возвращает нулевое значение. Поле `d_name` структуры `dp` содержит имя файла, а поле `d_ino` — номер индексного дескриптора.

Вот каким может быть результат выполнения программы opendir_demo:

```
# ./opendir_demo
              .:          i-number = 123875
             ..:          i-number = 62473
            run:          i-number = 123876
      stat_demo:          i-number = 123879
    stat_demo.c:          i-number = 123877
           test:          i-number = 123880
 opendir_demo.c:          i-number = 123887
       Stat_tut:          i-number = 123881
   opendir_demo:          i-number = 123882
```

3.4. Права доступа к файлам

В операционной системе UNIX все объекты файловой системы имеют определенные права доступа, которые позволяют или за-

прещают выполнять те или иные операции как отдельным пользователям, так и группам пользователей.

С *правами доступа* (access permission) мы сталкивались при анализе атрибутов файлов в предыдущем разделе, сейчас же рассмотрим их более подробно.

Каждый файл имеет определенную комбинацию прав доступа, состоящую из девяти битов. Эта группа битов определяет, например, пользователей, имеющих право на чтение содержимого файла, на запись данных или выполнение, если файл является исполняемым.

Биты этого набора вместе с другими тремя битами, определяющими способ запуска исполняемых файлов, образуют код режима доступа к файлу. Все двенадцать битов режима хранятся в 16-битовом поле атрибутов доступа индексного дескриптора вместе с четырьмя дополнительными битами, указывающими тип файла (см. рис. 3.3).

Четыре последних бита устанавливаются при создании файла и не должны изменяться. Биты доступа может изменить либо владелец файла, либо суперпользователь root при помощи команды UNIX `chmod`. Эти биты вместе с остальной информацией отображаются на экране командой `ls`.

В коде режима доступа есть биты специального назначения — им соответствуют восьмеричные значения 4000 и 2000. Один из этих битов называется битом замены идентификатора пользователя (SUID), а другой — битом замены идентификатора группы (SGID). Данные биты позволяют программам пользователя получать доступ к файлам и процессам, которые при иных обстоятельствах были бы пользователю недоступны. Бит, которому соответствует восьмеричное значение 1000, называется sticky-битом — в ранних системах он использовался, но современные UNIX-системы его игнорируют.

В UNIX-системах нельзя устанавливать биты прав доступа отдельно для каждого пользователя — вместо этого можно применять различные комбинации из трех битов (триады) для владельца файла, группы, которой принадлежит файл, и прочих

пользователей. Каждая триада включает бит чтения, бит записи и бит выполнения (для каталога последний называется битом поиска).

Очень часто код режима доступа представляют в виде восьмеричного числа, при этом каждая цифра в нем представляется тремя битами:

- три старших бита (восьмеричные значения 400, 200 и 100) определяют права доступа к файлу со стороны его владельца;
- три средних бита (восьмеричные значения 40, 20 и 10) задают доступ для пользователей группы;
- три младших бита (восьмеричные значения 4, 2 и 1) определяют права доступа к файлу для остальных пользователей.

Старший бит каждой триады определяет доступ по чтению, средний — по записи, и, наконец, младший бит определяет права на выполнение.

Каждый пользователь попадает только в одну из категорий, соответствующих одной из триад битов режима, при этом из возможных комбинаций выбираются те, которые устанавливают самые строгие права доступа. Например, права доступа для владельца файла всегда определяются триадой битов владельца и никогда — битами группы. Для того чтобы удалить или переименовать файл, необходимо, чтобы были установлены соответствующие биты прав доступа для каталога, где хранится имя файла. Установленный бит выполнения, например, разрешает выполнить файл как программу или командный сценарий, хотя для каталогов этот бит имеет иной смысл — если единственным установленным битом является только бит выполнения, то разрешается вход в каталог, но получить список его содержимого нельзя.

Содержимое каталога можно просмотреть только при установленных битах чтения и выполнения, а установленные биты записи и выполнения позволяют создавать, удалять и переименовывать файлы в данном каталоге.

В операционной системе UNIX есть несколько команд, позволяющих устанавливать права доступа к файлам и каталогам —

это команды `chmod` и `chown`, причем выполнять их может либо владелец файла, либо суперпользователь root.

Команда `chmod` устанавливает права доступа к указанным в командной строке файлам и имеет следующий синтаксис:

```
chmod [-fR] абсолютные_права файл ...
chmod [-fR] символьные_права файл ...
```

Первый параметр команды указывает права доступа, а второй и последующий задают имена файлов и права доступа, подлежащие изменению. Код прав доступа можно задавать в виде восьмеричного числа, хотя в современных версиях поддерживается более наглядная система мнемонических обозначений, основным преимуществом которой является возможность сброса/установки отдельных битов режима.

В восьмеричной нотации первая цифра относится к владельцу, вторая — к группе, а третья — к остальным пользователям. При необходимости задать биты SUID/SGID следует указывать не три, а четыре восьмеричные цифры, при этом первая цифра будет соответствовать трем специальным битам.

Опция `-f` команды блокирует вывод сообщения о невозможности установки прав доступа, а с помощью опции `-R` можно установить или изменить права доступа рекурсивно для всех подкаталогов, указанных в списке.

Абсолютные права доступа задаются восьмеричным числом, в то время как символьные права доступа указываются в виде списка выражений (через запятую), например:

```
[пользователи] оператор [права, ...]
```

Элемент *пользователи* списка определяет, для кого задаются или изменяются права. Он может принимать значения `u`, `g`, `o` и `a`, относящиеся к владельцу, группе, остальным пользователям, а также ко всем категориям пользователей соответственно. При этом символ `u` (user) обозначает владельца файла, символ `g` (group) — группу, символ `o` (others) — других пользователей, символ `a` (all) — всех пользователей сразу.

Если элемент *пользователи* не указан, изменения прав действуют для всех категорий пользователей, хотя не переопределяются установки, задаваемые маской создания файлов.

Элемент *оператор* списка может принимать значения `+`, `-` или `=`, означающие добавление, отмену права доступа и установку указанных прав соответственно. Если после оператора `=` ничего не указано, то все права доступа для соответствующих категорий пользователей отменяются.

Компонент *права* задается в виде любой совместимой комбинации следующих символов: `g`, `s`, `t`, `u`, `x`.

Не все сочетания символов для компонента *пользователи* и компонента *права* допустимы. Так, `s` можно задавать только для `u` или `g`, а `t` — только для `u`. Права `x` и `s`, например, несовместимы с `t` и т. д. Изменения прав доступа в списке выполняются последовательно, в порядке их перечисления.

Все возможные комбинации для каждого трехбитового набора перечислены в табл. 3.3 (символы `r`, `w` и `x` обозначают чтение, запись и выполнение соответственно).

Таблица 3.3. *Права доступа в команде* `chmod`

Восьмеричное число	Двоичное число	Маска режима доступа
0	000	—
1	001	`--x`
2	010	`-w-`
3	011	`-wx`
4	100	`r--`
5	101	`r-x`
6	110	`rw-`
7	111	`rwx`

Например, команда

```
chmod 744 text
```

предоставляет владельцу все права доступа к файлу text и право на чтение — остальным пользователям. Несколько примеров мнемонических обозначений приведено в табл. 3.4.

Таблица 3.4. *Примеры мнемонических спецификаций команды* `chmod`

Спецификация	Описание
`u+w`	Владельцу файла дополнительно разрешается выполнять файл
`ug=rwx,o=rx`	Владелец и группа получают право на чтение/запись и выполнение, остальные пользователи — право чтения и выполнения
`a-wx`	Все пользователи лишаются права записи и выполнения
`ug=srx,o=`	Владелец и группа получают право чтения/выполнения, и устанавливается бит SUID; доступ к файлу остальным пользователям запрещен
`g=u`	Группе назначаются такие же права, что и владельцу

Рассмотрим пример изменения прав доступа. Предположим, что в текущем каталоге имеется файл test, для которого мы будем изменять права доступа, как показано далее, одновременно наблюдая за результатами:

```
$ chmod +w test
$ ls -l test
-rw-r--r--   1 root 50            0 Sep 21 12:07 test
$ chmod a+w test
$ ls -l test
-rw-rw-rw-   1 root 50            0 Sep 21 12:10 test
$ chmod u+x,g=x,o= test
```

```
$ ls -l test
-rwx--x---   1 root 50             0 Sep 21 12:12 test
$ chmod ug-x,og+r,u=rwx test
$ ls -l test
-rwxr--r--   1 root 50             0 Sep 21 12:15 test
$ chmod 644 test
$ ls -l test
-rw-r--r--   1 root 50             0 Sep 21 12:18 test
```

При создании нового файла ему присваиваются права доступа, определяемые пользовательской маской режима создания файлов. Для этого используется встроенная команда `umask` интерпретатора shell, которая присваивает пользовательской маске режима создания файлов указанное восьмеричное значение. Три восьмеричные цифры соответствуют правам на чтение/запись/выполнение для владельца, членов группы и прочих пользователей соответственно.

Команда имеет следующий синтаксис:

```
umask [-S] [маска]
```

Выполненная без параметров, команда `umask` отображает текущее значение маски в восьмеричном виде. Маска используется при указании прав доступа следующим образом: ее значение необходимо "вычесть" из максимальных прав доступа (777 для исполняемых файлов и 666 для обычных файлов), например:

```
$ umask
022
```

При указанном значении маски обычные текстовые файлы будут создаваться с правами доступа 666 − 022 = 644, как это имеет место в следующем примере при создании файла test:

```
$ > test
$ ls -l test
-rw-r--r--   1 root 50             0 Sep 12 14:39 test
```

Операция "вычитания" для значения маски формально выполняется как побитовое логическое И дополнения маски и максимальных прав доступа. В нашем примере расчет прав доступа с использованием маски показан в табл. 3.5.

Таблица 3.5. *Расчет маски*

Последовательность выполнения	Значение	Результат
Шаг 1	Двоичное значение маски	000010010 (022)
Шаг 2	Дополнение маски	111101101 (755)
Шаг 3	Максимальное значение прав	110110110 (666)
Шаг 4	Логическое И предыдущих двух строк	110100100 (644)
Шаг 5	Результирующие биты прав	110100100 (644)

Опция `-S` позволяет выводить значение маски в символьном виде, при этом отображаются биты прав доступа у вновь создаваемого файла. Бит выполнения в этом случае также берется во внимание:

```
$ umask -S
u=rwx,g=rx,o=rx
```

Команду `umask` нередко включают в файлы начального запуска, устанавливающие рабочую среду для командного интерпретатора, используемого по умолчанию.

Далее приводится еще один пример создания файла при значении маски, равном 257:

```
$ umask 257
$ > test
$ ls -l test
-r---w----  root 1  50              0 Sep 14 17:14 test
```

Владелец файла, а также суперпользователь root могут изменить владельца и группу-владельца файла, используя команду `chown`, имеющую следующий синтаксис:

```
chown [-h][-R] владелец[:группа] файл ...
```

Для замены группы, владеющей файлом, можно использовать команду `chgrp`:

```
chgrp [-h][-R] группа файл
```

В качестве первого параметра обеих команд используется имя нового владельца файла или новой группы соответственно. Выполнить команду `chgrp` может только владелец файла, пользователь, входящий в назначаемую группу, или суперпользователь root.

Опция `-h` требует замены владельца файла, на который указывает символическая ссылка, а не самой ссылки, как это принято по умолчанию.

В большинстве версий команд `chown` и `chgrp` предусмотрен флаг `-R`, который задает смену владельца или группы не только самого каталога, но и всех его подкаталогов и файлов.

В операционных системах UNIX, совместимых с System V, пользователи могут беспрепятственно поменять владельца собственного файла при помощи команды `chown`, тогда как в BSD-совместимых системах эту команду может выполнять только суперпользователь root.

Необходимо учитывать, что после смены владельца файла бывший владелец будет иметь права доступа, установленные новым владельцем. Пусть требуется сменить владельца файла test, атрибуты которого показаны далее:

```
$ ls -l
total 2
-rw-r--r--    1 user1   others        6 Sep 10 16:19 test
```

Из результата выполнения команды `ls` видно, что владельцем файла test является пользователь user1, в сеансе которого и вы-

полняются команды. Если заменить владельца файла test с user1 на root, выполнив команды:

```
$ chown root test
$ ls -l
total 2
-rw-r--r--    1 root others           6 Sep 10 16:19 test
```

то увидим, что после выполнения команды `chown` пользователем файла test является root.

Теперь пользователь user1 никаким образом не сможет в своем сеансе заменить владельца опять на user1:

```
$ chown user1 test
UX:chown: ERROR: testf: Not privileged
```

Полученное сообщение об ошибке говорит о том, что у пользователя user1 нет прав привилегированного пользователя для смены владельца файла.

Помимо рассмотренных нами атрибутов доступа, объекты файловой системы операционной системы UNIX имеют и другие атрибуты, которые можно получить, используя команду `ls -l`, как в этом примере:

```
$ ls -l /bin/sh
-rwxr-x--x  1   root  bin  87924 Sep 21  2005 /bin/sh
```

Проанализируем полученный результат. Здесь в первом поле задается тип файла и маска режима доступа к нему: поскольку первым символом является дефис, то это обычный файл.

Обозначения различных типов файлов представлены кодами, состоящими из одного символа (табл. 3.6).

Таблица 3.6. *Обозначение типов файлов в выводе команды* `ls -l`

Тип файла	Символ	Создается командой	Удаляется командой
Обычный файл	—	Редакторы, `cp` и др.	—

Таблица 3.6 *(окончание)*

Тип файла	Символ	Создается командой	Удаляется командой
Каталог	`d`	`mkdir`	`rmdir`, `rm -r`
Файл байт-ориентированного устройства	`c`	`mknod`	`rm`

Вернемся к результату выполнения команды `ls`. Следующие за дефисом девять символов первого поля представляют триады битов режима, обозначенные литерами `r`, `w` и `x` (чтение, запись и выполнение соответственно). Для данного примера владелец обладает полным доступом к файлу, пользователи группы bin — правом на чтение и выполнение, а остальные пользователи могут только выполнить этот файл.

Если бы был установлен бит смены идентификатора пользователя (SUID), то вместо `x` стоял бы символ `s`. Аналогично, если бы был установлен бит смены идентификатора группы (SGID), то вместо `x` для группы стоял бы символ `s`. Последний бит режима (право выполнения для остальных пользователей) представляется буквой `t`, когда для файла задан sticky-бит. Если биты SUID/SGID или sticky-бит установлены, а надлежащий бит выполнения — нет, эти биты представляются, соответственно, символами, указывающими на наличие ошибки и игнорирование данных атрибутов.

В следующем поле вывода команды `ls` отображается количество жестких ссылок на файл — оно равно 1, а это значит, что /bin/sh — единственное имя, под которым известен данный файл. Всякий раз при создании жесткой ссылки на файл значение счетчика ссылок увеличивается на единицу, при этом символические ссылки в счетчике не учитываются. Что же касается каталогов, то любой из них имеет, как минимум, две жесткие ссылки: одну из родительского каталога и одну из специального файла внутри самого каталога.

Следующие два поля отображают владельца и группу-владельца файла — в данном примере владельцем файла является root, а

файл принадлежит группе bin. Ядро UNIX хранит эти данные не в виде строк, а как идентификаторы пользователя и группы. Если получить символьное представление имен по какой-либо причине невозможно, то в этих полях будут отображаться числа. Такое случается, если запись пользователя или группы была удалена из файла /etc/passwd или /etc/group соответственно.

Далее следует поле, отображающее размер файла в байтах: данный файл имеет размер 87 924 байта, т. е. почти 88 Кбайт. Следующее поле содержит дату последнего изменения: 21 сентября 2005 г., и, наконец, в последнем поле вывода содержится имя файла: /bin/sh.

Если мы имеем дело с файлом устройства, то вывод команды `ls` будет другим, например:

```
$ ls -l /dev/ttya

crw-rw-rw-  1   root  daemon     12,   0  Dec  20   1998
/dev/ttya
```

Результат работы этой команды иной, чем в предыдущем примере: вместо размера в байтах показаны старший и младший номера устройства, а в первом поле отображается тип устройства (в данном случае литера `c` в первой позиции означает, что устройство имеет символьный тип). Имя `/dev/ttya` относится к первому устройству, управляемому драйвером устройства 12 (в данной системе это драйвер терминала).

При поиске жестких ссылок часто бывает полезной команда `ls -i`, отображающая для каждого файла номер индексного дескриптора. Жесткие ссылки, указывающие на один и тот же файл, будут иметь один и тот же номер.

Операционная система автоматически отслеживает такие атрибуты, как время изменения, число ссылок и размер файла, автоматически устанавливая корректные значения. В то же время права доступа и идентификаторы принадлежности файла могут быть модифицированы явным образом с помощью команд `chmod`, `chown` и `chgrp`.

3.5. Операции с дисковыми файлами

В этом разделе мы рассмотрим практические аспекты работы с объектами файловой системы, такими как дисковые файлы и каталоги. Именно операции с файлами и каталогами, расположенными на жестком диске, в подавляющем большинстве случаев приходится выполнять пользователю. Как известно, к объектам файловой системы относятся также и устройства ввода/вывода, но анализ выполнения операций с такими объектами требует специальных знаний, поэтому рассматривать эти вопросы здесь мы не будем.

Операции над файлами можно выполнять с использованием команд UNIX или, при разработке программ на языках высокого уровня, посредством системных вызовов или библиотечных функций. Следует отметить, что все команды UNIX для манипуляций файлами реализованы с использованием системных вызовов, перечисленных в табл. 3.1.

Операционная система UNIX позволяет выполнять различные манипуляции над объектами файловой системы, включая:

- создание и удаление файлов;
- копирование, перемещение и создание ссылок на объекты файловой системы;
- чтение/запись данных;
- установку и изменение атрибутов файлов.

Некоторые из этих операций, например, установка и изменение атрибутов файлов, а также чтение и запись данных, мы рассматривали ранее, сейчас же сосредоточим внимание на операциях создания/удаления и копирования/перемещения дисковых файлов и каталогов и начнем с копирования файлов.

3.5.1. Копирование файлов

Для копирования файлов используется команда `cp`. Она позволяет копировать файлы или каталоги, допуская копирование

одного файла в другой, а также копирование группы файлов в заданный каталог.

Синтаксис команды `cp` можно представить следующим образом:

`cp [`*опции*`] `*файл путь*

`cp [`*опции*`] `*файл... каталог*

Если в качестве последнего параметра `cp` задан существующий каталог, то выполняется копирование исходных файлов в этот каталог с сохранением их имен. В том случае, если параметрами являются имена файлов, `cp` копирует первый файл во второй.

Если командная строка содержит более двух параметров, не являющихся опциями самой команды, а последний параметр не является именем какого-либо каталога, то команда генерирует ошибку. Попытка скопировать файл сам в себя ни к чему не приводит, кроме того, что выдается сообщение об ошибке.

Права доступа к скопированным файлам и каталогам вычисляются путем логического умножения (операция И) кода доступа исходных файлов на 0777, а также с учетом маски, установленной для пользователя.

Вот некоторые примеры использования команды `cp`. Для копирования одного каталога в другой можно выполнить команду:

```
# cp -r DIR DIR.OLD
```

Здесь каталог DIR вместе со своим содержимым копируется в каталог DIR.OLD.

В следующем примере команда

```
# cp -r DIR1 DIR2 DIR12
```

копирует содержимое каталогов DIR1 и DIR2 в каталог DIR12.

Операцию копирования несложно реализовать в приложении, написанном на одном из языков высокого уровня. Исходный текст простейшего аналога UNIX-команды `cp`, написанный на языке C, представлен в листинге 3.3.

Листинг 3.3. Копирование файлов с использованием функций API

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>

int main(int argc, char* argv[])
{
 if (argc != 3)
  {
    printf("Usage: %s [source] [dest]\n", argv[0]);
    exit(0);
  }

  char buf[1024];
  int bytes;
  int fsrc = open(argv[1], O_RDONLY);
  if (fsrc == -1)
   {
     printf("Can't open %s\n", argv[1]);
     exit(1);
   }
  int fdst = open(argv[2], O_RDWR | O_CREAT, 764);
  if (fdst == -1)
   {
     printf("Can't create %s\n", argv[2]);
     exit(2);
   }
  while (1)
  {
    bytes = read(fsrc, buf, sizeof(buf));
```

```
    if (bytes == 0)
      break;
    write (fdst, buf, bytes);
  }

  close(fsrc);
  close(fdst);
  return 0;
}
```

Как видно из исходного текста программы, в ней используются уже знакомые нам системные вызовы UNIX `open()`, `read()`, `write()` и `close()`. Программа принимает два параметра: первый параметр `argv[1]` указывает имя исходного файла, а второй `argv[2]` — имя файла назначения.

Исходный файл открывается при помощи системного вызова `open()` для чтения, после чего все операции чтения выполняются посредством дескриптора `fsrc`. Считанные из дескриптора `fsrc` данные (оператор `bytes = read(fsrc, buf, sizeof(buf));`) помещаются во временный буфер памяти `buf`, после чего записываются в дескриптор `fdst` вновь создаваемого файла функцией `write()`:

```
write (fdst, buf, bytes);
```

Операция копирования завершается, если при чтении в буфер количество прочитанных байтов `bytes` становится равным 0 — при этом происходит выход из цикла `while` и закрытие дескрипторов файлов функцией `close()`.

3.5.2. Перемещение файлов

Перемещение файлов в операционной системе UNIX выполняется с помощью команды `mv`, имеющей синтаксис:

`mv [`*опции*`...] `*исходный_файл файл_назначения*

`mv [`*опции*`...] `*исходный_файл*`... `*каталог*

Если последний параметр команды указывает на имя существующего каталога, то mv перемещает указанные файлы в этот каталог. В том случае, если в качестве параметров заданы имена двух файлов, то имя первого файла будет изменено на имя второго. Если же последний параметр не является каталогом, и заданы имена более чем двух файлов, то команда генерирует ошибку.

Когда *исходный_файл* и *файл_назначения* находятся в одной файловой системе, то изменяется имя файла, а владелец, права доступа, атрибуты времени остаются неизменными. Если же они находятся в разных файловых системах, то *исходный_файл* копируется и затем удаляется. Во время выполнения операции команда mv пытается скопировать время последней модификации, время доступа, идентификаторы пользователя и группы и права доступа к файлу.

Вот пример использования команды mv:

```
# mv test test.old
```

Здесь файл test переименовывается в файл test.old.

Программный аналог UNIX-команды mv несложно реализовать при помощи системных вызовов link() и unlink(), как это показано в листинге 3.4.

Листинг 3.4. Перемещение файлов с использованием функций API

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
 if (argc != 3)
  {
    printf("Usage: %s [source] [dest]\n", argv[0]);
```

```
    exit(0);
  }

 if (link(argv[1], argv[2]) == 0)
   if (unlink(argv[1]) == -1)
      printf("Can't delete file %s\n", argv[1]);
 return 0;
}
```

В этой программе с помощью системного вызова `link()` создается дополнительная жесткая ссылка (параметр `argv[2]`) на исходный файл, имя которого указано первым параметром `argv[1]`. Если эта операция выполнена успешно, то первая жесткая ссылка (она же является именем исходного файла) удаляется функцией `unlink()`.

3.5.3. Удаление файлов и каталогов

Для удаления файлов и каталогов в операционной системе UNIX используются команды `rm` и `rmdir`.

С помощью команды `rmdir` можно удалить одиночный каталог, причем он должен быть пустым. Если в каталоге имеются элементы, отличные от . и .., то команда `rmdir` такой каталог не удаляет.

Синтаксис этой команды таков:

`rmdir [-p][-s]` *`каталог`*

Команда `rmdir` имеет две опции:

- `-p` — позволяет удалить пустой каталог вместе с его родительскими каталогами, отображая сообщение об успешном или неуспешном выполнении операции;
- `-s` — подавляет выдачу сообщений при использовании опции `-p`.

Команда `rm` функционирует иначе — с ее помощью можно удалить указанные файлы, но каталоги по умолчанию не удаляются.

При указании опций `-r` или `-R` будет удаляться все дерево каталогов нижезаданного каталога, включая и сам каталог, причем на глубину дерева не накладывается никаких ограничений. Если последний компонент файла — символ `.` или `..`, то генерируется ошибка (это помогает избежать неприятных сюрпризов при выполнении команды `rm -r .*` или ей подобных).

Общие для UNIX-систем опции POSIX данной команды имеют следующий смысл:

- `-f` — не запрашивается подтверждение операции и не выдаются диагностические сообщения. При завершении команды с ошибками код ошибки не возвращается, если ошибки вызваны отсутствием файлов;
- `-i` — выводится запрос на подтверждение удаления (при указании опций `-f` и `-i` одновременно используется последняя);
- `-r` или `-R` — позволяет рекурсивно удалять дерево каталогов.

Команда `rm` довольно опасна — после ее выполнения восстановить удаленные файлы невозможно, поэтому нужно быть очень внимательным при ее использовании.

3.5.4. Создание каталогов

Создать новый каталог в UNIX можно с помощью команды `mkdir`. В простейшем варианте команда использует один параметр (имя каталога), создавая каталог с указанным именем.

С помощью одной команды `mkdir` можно создавать несколько каталогов одновременно, перечисляя их в одной командной строке. Синтаксис команды таков:

`mkdir [`*опции*`] [`*список_каталогов*`]`

В команде `mkdir` можно использовать две опции:

- `-m` — позволяет задать в восьмеричной или символьной форме права доступа (как и для команды `chmod`), которые будут присвоены создаваемым каталогам;

- `-p` — кроме указанного каталога создаются любые требуемые промежуточные каталоги. Если у пользователя нет прав на запись в родительский каталог, то новый каталог не создается, а если каталог уже существует (или файл с таким же именем), то команда генерирует ошибку.

3.6. Поиск файлов и каталогов

Кроме копирования/перемещения и создания/удаления к часто выполняемым операциям с файлами и каталогами относится и поиск объектов файловой системы. Файловая система UNIX содержит десятки тысяч файлов, поэтому для быстрого поиска используются очень эффективные средства, одним из которых является команда `find`.

Команда имеет синтаксис:

`find` *`каталог`* `...` *`выражение`*

Она просматривает иерархии каталогов в поисках файлов, удовлетворяющих критерию, задаваемому выражением *`выражение`*. Выражения строятся из элементов с помощью следующих конструкций:

- `-name` *`шаблон`* — условие истинно, если имя файла соответствует шаблону. При использовании метасимволов необходимо маскировать шаблоны от командного интерпретатора;
- `-type` *`тип`* — условие истинно, если файл — указанного типа. Типы файлов задаются символами `b`, `c`, `d`, `f`, `l`, `p` и `s`, обозначающими, соответственно, специальное блочное устройство, специальное символьное устройство, каталог, обычный файл, символическую ссылку, именованный канал и сокет;
- `-user` *`пользователь`* — условие истинно, если файл принадлежит пользователю, указанному по идентификатору или регистрационному имени;
- `-group` *`группа`* — условие истинно, если файл принадлежит группе, указанной по идентификатору или имени;

- `-perm [-] права` — если дефис не задан, то условие истинно, только если права доступа в точности соответствуют указанным (как в команде `chmod`). Если задан дефис, то условие истинно, если в правах доступа файла, как минимум, установлены те же биты, что и в указанных правах;
- `-size [+|-|=]n[c]` — условие истинно, если файл имеет длину `n` блоков (блок — 512 байтов) или символов (если указан суффикс `c`). Перед размером можно указывать префикс `+` (не меньше), `-` (не больше) или `=` (в точности равен);
- `-atime [+|-|=]n` — условие истинно, если к файлу последний раз обращались `n` дней назад. Перед `n` в элементах `-atime`, `-ctime` и `-mtime` можно указывать префикс `+` (не позже), `-` (не ранее) или `=` (ровно);
- `-ctime n` — условие истинно, если файл создан `n` дней назад;
- `-mtime n` — условие истинно, если файл был изменен `n` дней назад;
- `-newer файл` — условие истинно, если файл более новый, чем указанный;
- `-ls` — условие истинно всегда (выдает информацию о файле, аналогичную длинному листингу);
- `-print` — условие истинно всегда (выдает полное имя файла в стандартный выходной поток);
- `-exec команда {} \;` — условие истинно, если выполненная команда имеет код возврата 0. Команда заканчивается замаскированной точкой с запятой. В команде можно использовать конструкцию `{}`, заменяемую полным именем рассматриваемого файла;
- `-ok команда {} \;` — аналогично `exec`, но полученная после подстановки имени файла вместо `{}` команда выдается с вопросительным знаком и выполняется, если пользователь ввел символ `y`;
- `-depth` — условие истинно всегда — требует так обходить иерархию каталогов, чтобы файлы любого каталога всегда обрабатывались раньше, чем сам каталог (обход "в глубину");

- `-prune` — условие истинно всегда — требует не проверять файлы в каталоге, путевое имя которого присутствует в предыдущем выражении. Не действует, если ранее указан элемент `-depth`.

В различных версиях операционной системы UNIX могут поддерживаться и другие компоненты выражений в команде `find`. Если командная строка сформирована неправильно, команда немедленно завершает работу.

Вот несколько примеров использования команды `find`:

Пример 1.

Для отображения списка файлов текущего каталога программы достаточно выполнить команду

```
# find . -print
```

Пример 2.

Для получения содержимого произвольного каталога, например, /home/developer нужно выполнить команду

```
# find /home/developer -print
```

Пример 3.

Для поиска файлов в текущем каталоге с именами, которые заканчиваются на tmp, нужно выполнить команду

```
# find . -name '*tmp' -print
```

Пример 4.

Здесь с помощью команды `find` выполняется поиск файлов с расширением tmp или c, находящихся в текущем каталоге:

```
# find . \( -name '*.tmp' -o -name '*.c' \) -print
```

В команде `find` можно задавать временные критерии поиска файлов, причем в самых различных комбинациях. Следующий пример демонстрирует это: в нем используется опция `-atime [+|-|=]n`. Условие является истинным, если время последнего доступа к файлу больше/меньше, чем `n*24`. Например, команда

```
# find . \( -name '*.tmp' -o -name '*.pl' \) -atime +3 -print
```

выполняет поиск файлов с указанными шаблонами, к которым не было обращения больше трех суток.

Нередко требуется найти файлы, принадлежащие определенному пользователю. Например, следующая команда выполняет поиск файлов в каталоге /usr, владельцем которых является суперпользователь root:

```
# find /usr -user root -print
```

Если критерием поиска является размер файла, то можно использовать следующую опцию: `-size [+|-|=]n[c]`. Условие, задаваемое этой опцией, истинно, если размер файла больше/меньше *n*. При этом различают два случая: если присутствует опция `c`, то размер файла предполагается заданным в байтах, если опция `c` отсутствует — то в блоках по 512 байтов.

Следующая команда выполняет поиск файлов, размер которых превышает 2048 байтов, в каталоге /developer:

```
$ find /developer -size +2048c -print
```

Команда `find` может выполнять другие команды или группы команд, принимающих в качестве параметра результат поиска файлов. Для реализации такой возможности служит опция `-exec`. В этом случае команда должна заканчиваться пробелом и символами `\;`.

В следующем примере из каталога /developer удаляются все файлы, размер которых не превышает 1000 байтов:

```
# find /developer -size -1000c -print -exec rm {} \;
```

Для вывода на консоль атрибутов файлов (команда `ls -l`), удовлетворяющих шаблону t*, можно воспользоваться командой

```
# find /developer -name 't*' -exec ls -l {} \;
```

Расширить возможности команды `find` можно, перенаправив ее вывод не на стандартное устройство вывода, а в программный канал, как это показано в следующем примере:

```
# find TMP -name 't*' -print|grep tmp
```

Здесь команда `find` выполняет поиск файлов в каталоге TMP, удовлетворяющих шаблону t*, в имени которых присутствует tmp.

Конвейер программ чаще всего применяется в операциях копирования, перемещения и создания резервных копий файловых систем — при этом вывод команды `find` служит вводом для команды архивирования, как правило, `cpio`.

В следующем примере выполняется копирование файлов в другой каталог. Для этого применяется команда `cpio -p`, которая принимает из стандартного входного потока список файлов и копирует или создает на них ссылки (опция `-l`) в каталоге NEW (к моменту выполнения копирования он должен существовать). Опция `-d` требует создания каталогов при необходимости, а опция `-m` запрещает модификацию времени изменения файла.

Для генерации списка полных путевых имен файлов для `cpio` в команде `find` нужно задать опцию `-depth` — это позволяет создавать файлы в каталогах, доступных только для чтения. Вот так выглядит командная строка для выполнения операции копирования:

```
# find . -depth -print | cpio -pdlmv NEW
```

Заканчивая обзор возможностей операционной системы UNIX для работы с файлами, хочу добавить, что дополнительную информацию по данной теме можно почерпнуть из man-страниц или из многочисленных источников в Интернете.

Глава 4

Учетные записи пользователей

Доступ к операционным системам UNIX, как и ко многим другим, осуществляется только через учетные записи пользователей. *Учетная запись* (account) пользователя включает в себя информацию, необходимую для его регистрации в системе и начала сеанса работы:

- регистрационное имя;
- пароль для входа в систему;
- исходный (домашний) каталог пользователя;
- файлы инициализации рабочей среды пользователя.

Взаимодействие пользователя и операционной системы можно представить себе так, как показано на рис. 4.1.

На этом рисунке показан вход пользователя в систему, функционирующую на локальном компьютере. Проанализируем более подробно каждый из этапов, показанных на рис. 4.1.

Операционные системы UNIX для подключения пользователя к системе создают так называемые *терминальные соединения* или *терминальные линии*. Исторически сложилось так, что в ранних версиях UNIX для подключения нескольких пользователей использовались последовательные порты, доступ через которые осуществлялся под управлением программы `getty`. В современных системах UNIX несколько пользователей могут подклю-

чаться к одной системе (локальной или сетевой), применяя механизм так называемых "виртуальных терминалов" или, в контексте пользователя, "виртуальных консолей".

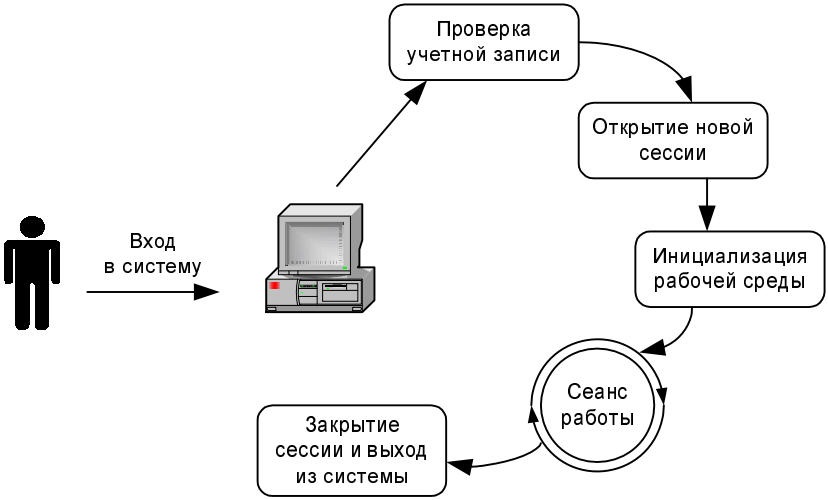


Рис. 4.1. Взаимодействие пользователя с операционной системой UNIX

Для пользователя виртуальная консоль представляется в виде отдельной терминальной линии, реализованной программно и не зависящей от физической природы соединения (локальный компьютер, сериальный порт или сетевое соединение). При этом пользователю выделяется отдельный логический дисплей, связанный с конкретным физическим устройством. Управление виртуальной консолью обычно осуществляется специальной программой, совместимой по функциям с `getty`. Например, во многих операционных системах Linux такой программой является `mingetty` (minimal getty), а для управления терминальными соединениями в Solaris используется намного более сложная и функциональная программа `ttymon`.

Несколько виртуальных консолей могут использовать одну и ту же физическую линию или одно и то же устройство. Сказанное

иллюстрирует рис. 4.2, где показаны виртуальные соединения для Linux-системы.

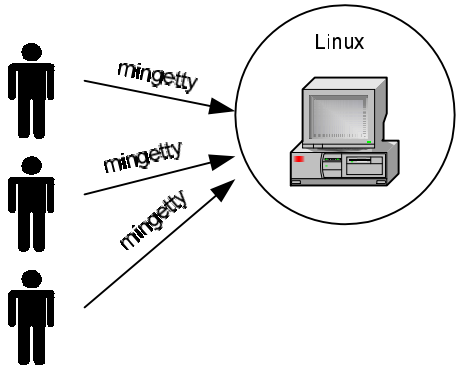


Рис. 4.2. Виртуальные терминалы в Linux

Здесь несколько пользователей могут подключиться к одной локальной системе, причем это можно делать даже с основной консоли, используя комбинации клавиш <Alt>+<Ctrl>+<F*n*>. При этом соответствующий данному виртуальному соединению процесс `mingetty` ожидает ввода имени и пароля, отображая на экране соответствующее приглашение.

Ожидающие процессы `mingetty` можно просмотреть с помощью команды `ps`:

```
# ps -ef|grep tty*
root      5288     1  0 10:03 tty1     00:00:00
/sbin/mingetty tty1
root      5290     1  0 10:03 tty3     00:00:00
/sbin/mingetty tty3
root      5291     1  0 10:03 tty4     00:00:00
/sbin/mingetty tty4
root      5292     1  0 10:03 tty5     00:00:00
/sbin/mingetty tty5
root      5293     1  0 10:03 tty6     00:00:00
/sbin/mingetty tty6
```

```
root      7675     1  0 11:18 tty2     00:00:00
/sbin/mingetty tty2
root      7679  5458  0 11:18 pts/0    00:00:00 grep tty*
```

Из результата выполнения команды `ps` видно, что в системе выполняется 6 процессов `mingetty`, управляющих терминальными линиями tty1—tty6 (такие обозначения используются в Linux).

Если теперь, например, переключиться на виртуальную консоль 2 (комбинация клавиш <Alt>+<Ctrl>+<F2>) и войти в систему как пользователь yury, а с консоли 3 (комбинация клавиш <Alt>+<Ctrl>+<F3>) зайти как пользователь user1, то команда `ps` даст такой результат:

```
# ps -ef|grep tty*
root      5288     1  0 10:03 tty1     00:00:00
/sbin/mingetty tty1
root      5291     1  0 10:03 tty4     00:00:00
/sbin/mingetty tty4
root      5292     1  0 10:03 tty5     00:00:00
/sbin/mingetty tty5
root      5293     1  0 10:03 tty6     00:00:00
/sbin/mingetty tty6
yury      7681  7675  0 11:23 tty2     00:00:00 -bash
user1     7777  7776  0 11:27 tty3     00:00:00 -bash
root      7819  5458  0 11:27 pts/0    00:00:00 grep tty*
```

Как видно из вывода команды `ps`, терминальные линии tty2 и tty3 используются в сеансах пользователей yury и user1 с запущенными командными интерпретаторами Bourne.

Хочу уточнить, что процессы `mingetty`, `ttymon` и им подобные в других системах не управляют входом пользователя в систему: их задача — обеспечить терминальное соединение с указанными параметрами (интерпретация служебных символов, эхо-контроль и т. д.). Сам процесс входа в систему и инициализация сеанса работы выполняются командой `login` и другими системными службами.

Процессы, подобные `mingetty`, во всех операционных системах запускаются процессом `init`, считывающим таблицу /etc/inittab в процессе запуска системы. Вот фрагмент системной таблицы /etc/inittab в Red Hat Linux, содержащий записи для запуска `mingetty` (выделены жирным шрифтом):

```
# cat /etc/inittab
#
# inittab       This file describes how the INIT process
                should set up
#               the system in a certain run-level.
#
# Author:       Miquel van Smoorenburg,
                <miquels@drinkel.nl.mugnet.org>
#               Modified for RHS Linux by Marc Ewing and
                Donnie Barnes
#

id:5:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6

. . .

# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
```

```
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
```

Собственно сеанс работы пользователя начинается с момента подтверждения его полномочий (правильно введены регистрационное имя пользователя и пароль). Естественно, что для входа пользователя в систему в ней уже должна быть зарегистрирована учетная запись, пароль учетной записи и настроена соответствующим образом рабочая среда (создан исходный (домашний) каталог пользователя и сделаны нужные установки в файлах инициализации).

Для регистрации пользователь вводит регистрационное имя, представляющее собой строку символов из цифр и букв алфавита. После ввода идентификатора процесс `getty` (`mingetty`, `ttymon` и т. д.) запускает программу регистрации `login`, которая использует в качестве параметра введенный идентификатор пользователя.

При вводе пароля вводимые символы обычно не отображаются на экране консоли. Если пользователь допустил ошибку во время процедуры регистрации, выдается сообщение:

```
Login incorrect
```

после чего появляется новое приглашение к вводу. В большинстве случаев количество попыток входа в систему ограничено определенным значением (обычно 5), хотя и может быть изменено пользователем root. Если количество попыток входа в систему исчерпано, запись об этом может быть помещена в файл /var/adm/loginlog (если такой файл существует в системе), а терминальная линия, используемая для входа в систему, будет разорвана.

Вот последовательность действий, выполняемых командой `login`:

1. Процесс `login` проверяет содержимое файла /etc/passwd, чтобы определить, требуется ли ввод пароля при входе в

систему. Если требуется, то система выдает приглашение к вводу пароля на экран (пароли пользователей в большинстве операционных систем UNIX хранятся, как правило, в зашифрованном виде в файле /etc/shadow). Процесс `login` анализирует также срок действия пароля и, если он истек, требует его замены.

2. Процесс `login` выполняет все необходимые действия по регистрации пользователя и установке параметров сеанса (процесса с уникальными идентификаторами пользователя uid и группы gid). Кроме этого, каждому пользователю назначаются права доступа, позволяющие выполнять команды интерпретатора shell. В этот момент операционная система добавляет запись в файл, где хранятся записи о зарегистрированных пользователях. При успешной регистрации пользователя операционная система устанавливает параметры для текущего сеанса работы. Каждый сеанс пользователя получает числовой идентификатор пользователя и группы — оба они определяют права доступа пользователя к объектам файловой системы, и без них невозможно выполнить ни одной команды.

После этих действий операционная система запускает командный интерпретатор shell (обычно это bash), позволяющий выполнять команды операционной системы.

Идентификация пользователей в операционной системе определяется несколькими файлами, имеющими ключевое значение: /etc/passwd, /etc/group и /etc/shadow. Информация о пользователе, сохраненная в этих файлах, позволяет системе устанавливать, изменять и выполнять другие функции управления учетной записью пользователя.

Важной частью процесса настройки учетной записи является установка исходного (домашнего) каталога пользователя и предоставление файлов инициализации данного пользователя командному процессору, используемому при регистрации пользователя в системе. Пользовательский файл инициализации представляет собой сценарий командного процессора, который

устанавливает рабочую среду для пользователя после того, как он зарегистрируется в системе.

С помощью файла инициализации пользователь способен реализовать любую задачу, которая может быть выполнена в рамках некоторого сценария командного интерпретатора, однако основной задачей этого файла является установка характеристик рабочего окружения, таких как маршрут поиска файлов и каталогов, переменные окружения и графический интерфейс пользователя.

Каждый командный интерпретатор shell, используемый при регистрации, имеет собственный файл (или файлы) инициализации пользователя, расположенный в его домашнем каталоге.

Эти файлы запускаются автоматически, когда данный пользователь регистрируется в системе.

Пользовательские файлы инициализации, принятые по умолчанию, такие как .cshrc, .profile и .login, создаются автоматически в домашнем каталоге при добавлении к системе новой учетной записи пользователя. Для каждого командного процессора системное программное обеспечение UNIX предоставляет по умолчанию пользовательские файлы инициализации, которые находятся в каталоге /etc/skel (для Linux и Solaris). Эти файлы перечислены в табл. 4.1.

Таблица 4.1. *Принятые по умолчанию файлы инициализации в Linux и Solaris*

Имя файла	Описание
.bashrc (Linux)	Файл .bashrc, принятый по умолчанию для командных интерпретаторов Bourne и Korn
local.cshrc (Solaris)	Файл .cshrc, принятый по умолчанию для командного интерпретатора C
local.login (Solaris)	Файл .login, принятый по умолчанию для командного интерпретатора C
local.profile (Solaris)	Файл .profile, принятый по умолчанию для командных интерпретаторов Bourne и Korn

В некоторых версиях операционной системы Linux при создании учетной записи пользователя файл по умолчанию .bashrc копируется системой в домашний каталог пользователя под именем .bash_profile.

Стандартные файлы инициализации можно использовать в качестве отправной точки и модифицировать их для создания некоторого обобщенного набора файлов, предоставляющего рабочее окружение, общее для всех пользователей.

В файлах инициализации UNIX используется целый ряд так называемых системных переменных или, по-другому, переменных окружения, используемых ядром и пользовательскими процессами для настройки и идентификации параметров системы.

Для каждой учетной записи пользователя устанавливаются несколько переменных окружения, которые могут присутствовать в файлах инициализации и настраиваться соответствующим образом. Вот наиболее важные из них:

- `HOME` — устанавливает маршрут к исходному (домашнему) каталогу пользователя;
- `LOGNAME` — указывает регистрационное имя пользователя;
- `PATH` — перечисляет в определенном порядке имена каталогов, которые командный интерпретатор просматривает в поисках программы, имя которой пользователь вводит с командной строки. При этом если конкретный каталог в маршруте поиска отсутствует, то пользователь должен набирать полное имя команды. Значение переменной `PATH` по умолчанию определено в файле .profile (для командных интерпретаторов Bourne или Korn) или в файле .cshrc (для командного интерпретатора C) как часть процесса регистрации в системе. Очень важно то, в каком порядке перечисляются каталоги — программа с одним и тем же именем может встречаться в разных каталогах, поэтому будет запущена та из них, которая первой встретится в списке;
- `SHELL` — определяет командный интерпретатор shell для текущего пользователя;

- TERM — определяет терминал. Эта переменная должна быть переустановлена в файле /etc/.profile /etc/.login. Когда пользователь вызывает какой-нибудь редактор, система ищет файл с именем, задаваемым этой переменной окружения;
- MAIL — устанавливает маршрут к почтовому ящику пользователя.

Значения переменных окружения хранятся в файлах инициализации для каждой учетной записи. Просмотреть значения переменных можно при помощи команды env.

Вот фрагмент вывода команды env:

```
# env
TERM=xterm
SHELL=/sbin/sh
. . .
USER=root
. . .
PATH=/usr/sbin:/usr/bin:/usr/openwin/bin:/bin:/usr/ucb
MAIL=/var/mail/root
. . .
LOGNAME=root
```

Нередко требуется изменить значения переменных окружения для конкретного пользователя. Так, например, во многих случаях необходимо в переменную PATH добавить маршруты к исполняемым файлам. Предположим, что в переменную PATH нужно добавить путь /home/user/programs. Если пользователь работает с командным интерпретатором Bourne или Korn, то следует выполнить команды

```
PATH=$PATH:/home/usr/programs:.;export PATH
```

При этом к существующим маршрутам, указанным в переменной PATH, будет добавлен новый. Команда export делает модифицированную переменную PATH доступной другим программам.

Для проверки модифицированного значения переменной можно выполнить команду

```
echo $PATH
```

Вообще, в командных интерпретаторах Bourne или Korn для модификации переменных окружения следует использовать такую цепочку команд:

```
ПЕРЕМЕННАЯ=значение;export ПЕРЕМЕННАЯ
```

Здесь *ПЕРЕМЕННАЯ* — одна из переменных окружения.

Для того чтобы продемонстрировать принципы настройки рабочей среды пользователя, приведу пример такой настройки в операционной системе Red Hat Linux.

Здесь для каждого пользователя создается файл инициализации .bash_profile. При настройке рабочей среды пользователя можно создать файл .profile с соответствующими установками для конкретного пользователя, а можно и откорректировать один из существующих файлов инициализации, который система создает из шаблона, находящегося в каталоге /etc/skel.

Воспользуемся созданным системой файлом .bash_profile и внесем в него некоторые изменения. Пусть, например, необходимо при входе в систему под учетной записью user1 выполнить такие действия:

- ❒ отображать имя пользователя;
- ❒ отображать дату входа в систему;
- ❒ приглашение к вводу команд отображать как полный путь к домашнему каталогу пользователя, завершающийся символом >.

Исходный файл .bash_profile в домашнем каталоге пользователя выглядит так:

```
# .bash_profile


# Get the aliases and functions
if [ -f ~/.bashrc ]; then
```

```
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH
unset USERNAME
```

Добавим в него следующие строки:

```
echo Hello, `who am I|cut -c1-8`
echo Now is `date`
PS1=\$PWD'>'
```

и сохраним. При следующем входе пользователя user1 в систему на экране консоли будет отображаться следующая информация:

```
Hello, user1
Now is Fri Oct 6 12:58:32 EEST 2006
/home/user1>
```

При настройке рабочей среды пользователя необходимо помнить, что прежде чем внести какие-либо изменения в стандартные файлы инициализации или использовать собственные, нужно тщательно проверить работу всех команд и значения измененных параметров, записав их в текстовый файл и сделав его исполняемым. Если в файл инициализации будут внесены неправильные изменения, пользователь может не войти в систему вообще, и придется использовать учетную запись root для уничтожения зависшей сессии и корректировки файлов инициализации данного пользователя!

До сих пор мы полагали, что пользователь входит в систему, работающую на локальной машине, хотя многие пользователи должны подключаться с локального компьютера к какой-либо удаленной машине, находящейся в сети. В таких случаях обычно

используют одну из программ удаленного доступа `rlogin`. Программа `rlogin` позволяет инициировать удаленный сеанс работы с сетевым хостом, обеспечивая передачу зашифрованных данных. Команда может принимать в качестве параметров имя пользователя и имя хоста.

На компьютере, к которому выполняется подключение, должен выполняться процесс-сервер `rlogind`. Сервер `rlogind` является сетевым аналогом команды `login`, принимая удаленные запросы на вход в систему, инициированные клиентом с помощью программы `rlogin`. В современных системах сетевые запросы отслеживаются демоном `inetd`, который, в свою очередь, передает запрос серверу `rlogind`. Сеанс удаленной работы пользователя показан на рис. 4.3.

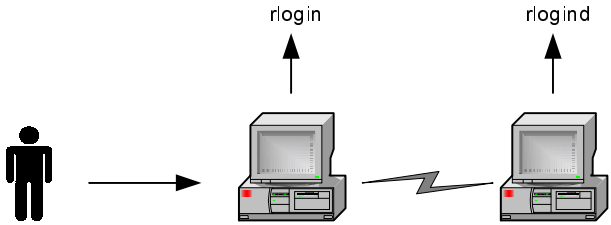


Рис. 4.3. Сеанс удаленной работы с помощью `rlogin`

Для сеанса удаленной связи могут использоваться и протоколы `telnet` и `ssh`. Последний обеспечивает высокий уровень защиты данных и безопасность сессии.

Любой пользователь операционной системы UNIX помимо домашнего каталога получает в свое распоряжение определенный набор ресурсов с установленными правами доступа к этим ресурсам. Например, пользователь может читать и записывать в файлы, выполнять печать документов на принтере, передавать и принимать сообщения электронной почты.

Для каждого пользователя в системе устанавливаются вполне определенные права (атрибуты) доступа к ресурсам, к тому же для разных пользователей они могут отличаться.

Все ресурсы операционной системы имеют определенные права доступа, причем часть таких прав устанавливается в процессе инсталляции операционной системы, а остальные атрибуты доступа устанавливаются или изменяются специальным пользователем root, наделенным всеми правами по управлению операционной системой. Пользователь root имеет все права на подключение, удаление и модификацию учетных записей, назначение и изменение паролей, назначение кода доступа к ресурсам операционной системы.

Рассмотрим подробнее, как осуществляются настройка и управление учетными записями пользователей операционной системы, и начнем с регистрации пользователей. После регистрации пользователя система устанавливает его параметры для текущего сеанса работы, одним из которых является идентификатор пользователя (сокращенно uid), представляющий собой 32-разрядное целое число в диапазоне от 0 до 2 147 483 647. Значение 0 идентификатора присваивается суперпользователю root, а идентификаторы 1 и 2 присваиваются пользователям с административными правами (bin, daemon).

Далее, пользователю назначаются определенные права доступа к файлам. Как известно, все объекты файловой системы имеют два идентификатора: пользователя и группы, которые наследуются от создавшего их процесса и определяют права доступа к файлу. При этом пользователь и группа, идентификаторы которых связаны с файлом, считаются его владельцами. Заменить владельца можно с помощью команд `chown` (change owner) и `chgrp` (change group).

В операционной системе UNIX существуют три категории доступа к объектам файловой системы:

- владелец файла (процесс, идентификатор пользователя которого равен идентификатору владельца файла);

- члены группы (процессы, идентификатор группы которых совпадает с идентификатором группы, установленной для файла);
- прочие — все остальные процессы.

Любой файл при создании получает код доступа, представляющий собой комбинацию битов в индексном дескрипторе файла. Комбинации единичных битов в коде доступа имеют такой смысл:

- 000400 (100h) — владелец файла имеет право на чтение его содержимого;
- 000200 (80h) — владелец имеет право на запись данных в файл;
- 000100 (40h) — владелец файла имеет права на выполнение файла;
- 000040 (20h) — члены группы имеют право на чтение;
- 000020 (10h) — члены группы имеют право на запись;
- 000010 (8h) — члены группы имеют право на выполнение;
- 000004 (4h) — прочие пользователи имеют право на чтение;
- 000002 (2h) — прочие пользователи имеют право на запись;
- 000001 (1h) — прочие пользователи имеют право на выполнение.

Все коды показаны в восьмеричном формате, а в скобках представлен их шестнадцатеричный эквивалент с завершающим символом "h".

Код доступа к файлу может быть изменен только привилегированным пользователем root. Мы сталкивались с этим термином ранее, поэтому дадим более точное его определение: *привилегированный пользователь* — это пользователь, имеющий процесс с идентификатором, равным нулю. Независимо от кода защиты, привилегированный пользователь имеет доступ ко всем файлам UNIX-системы. Учетный файл /etc/passwd содержит запись о привилегированном пользователе root, которого часто называют "суперпользователь".

Отдельные поля кода доступа предназначены для установки специальных атрибутов, позволяющих изменить идентификаторы пользователя и группы:

- 004000 (800h) — разрешение смены идентификатора пользователя при обращении к файлу (так называемый "set-uid"-флаг);
- 002000 (400h) — разрешение смены идентификатора группы при обращении к файлу (так называемый "set-gid"-флаг).

Смысл использования этих атрибутов рассмотрим более подробно.

Идентификаторы, полученные при запуске процесса (так называемые "реальные идентификаторы"), могут отличаться от идентификаторов, полученных после выполнения системного вызова `exec()` (так называемые "эффективные идентификаторы"). Изначально реальные и эффективные идентификаторы всегда совпадают, но если код доступа к файлу предусматривает смену идентификаторов, то после загрузки исполняемого файла с помощью системного вызова `exec()` реальные идентификаторы процесса изменяются на идентификаторы владельца (или группы) выполняемого файла, т. е. становятся эффективными.

Права доступа процесса проверяются по его эффективным идентификаторам. Если, например, владельцем файлов данных text1, text2 и text3 является пользователь user, то полный доступ к этим файлам разрешен только ему. Предположим, что пользователю user принадлежат исполняемые файлы program1, program2 и program3.

Пусть требуется, чтобы другие процессы, которые хотят выполнять операции над файлами text1, text2 и text3, могли это сделать с помощью программ program1, program2 и program3. Для этого в коде доступа файлов program1, program2 и program3 необходимо установить биты разрешения смены идентификатора пользователя для процесса, вызвавшего на выполнение один из этих файлов.

В этом случае другой процесс (не пользователя user!), которому требуется доступ к одному из файлов text1, text2 или text3, дол-

жен запустить через системный вызов `exec()` соответствующую программу обработки program1, program2 или program3, иначе он вообще не получит доступа к защищенным их владельцем файлам данных text1, text2 или text3.

Установку и изменение требуемых атрибутов доступа к файлу для пользователя, а также флагов set-uid и set-gid можно выполнить с помощью системного вызова `chmod()`. Можно воспользоваться также командой `chmod`, которая представляет собой "обертку" соответствующего системного вызова и доступна из командной оболочки shell.

Информацию о реальных и эффективных идентификаторах выполняющегося процесса можно получить с помощью системных вызовов `getuid()` и `getgid()`. Выполняющийся процесс может динамически изменять свои идентификаторы посредством системных вызовов `setuid()` и `setgid()`, однако такое допускается только для привилегированного процесса, выполняющегося с идентификатором root, или такого, у которого реальный идентификатор совпадает с устанавливаемым.

Например, специальная программа `passwd` позволяет изменить пароль пользователя в учетном файле пользователей /etc/passwd. Владельцем этих файлов является суперпользователь root. Код доступа файла /etc/passwd разрешает выполнять запись данных только владельцу, в то время как код доступа исполняемого файла `passwd` разрешает смену пароля пользователя. Следовательно, пользователь, отличный от root, может изменить свой пароль только с помощью программы `passwd`, что является корректным по отношению к пользователю, который всегда может изменить свой пароль, не сообщая об этом системному администратору.

Ключевую роль в регистрации и управлении учетными записями пользователей играют несколько файлов, упоминавшихся ранее в этой главе: /etc/passwd, /etc/group и /etc/shadow.

Рассмотрим их более подробно и начнем с файла паролей passwd. Этот файл обычно располагается в каталоге /etc и состоит из текстовых строк, представляющих собой отдельные учет-

ные записи, при этом отдельные поля каждой записи отделены друг от друга двоеточием.

Файл passwd имеет одинаковую структуру во всех операционных системах UNIX. Вот пример записи файла /etc/passwd для пользователя user1:

```
$ cat /etc/passwd|grep user1
user1:x:500:500:user1:/home/user1:/bin/bash
```

Все записи имеют семь полей, разделенных двоеточием. Приведем расшифровку (слева направо) отдельных полей, в качестве примера используя показанную выше запись:

- поле 1 — регистрационное имя пользователя (`user1`);
- поле 2 — зашифрованный пароль (`x`);
- поле 3 — идентификатор пользователя (`500`);
- поле 4 — идентификатор группы (`500`);
- поле 5 — информация о пользователе (`user1`);
- поле 6 — рабочий каталог пользователя (`/home/user1`);
- поле 7 — используемая командная оболочка (`/bin/bash`).

Поскольку операционные системы UNIX являются многопользовательскими, в них предусмотрен механизм группового доступа. При том, что учетная запись пользователя может входить в одну или несколько групп, принадлежать она может только одной группе, для чего в записях файла паролей /etc/passwd предусмотрено поле идентификатора группы (group ID, gid).

Информация о принадлежности пользователей к тем или иным группам находится в файле /etc/group. Для того чтобы включить пользователя в какую-либо группу, необходимо внести соответствующие изменения в файл /etc/group. В ранних версиях операционной системы UNIX пользователь мог быть членом только одной группы, но в современных системах такого ограничения более не существует, поэтому пользователь может входить в 16 групп одновременно. Поле идентификатора группы в настоящее время практически не используется операционной системой, тем не менее, ему присваивается определенное значение.

4.1. Команды UNIX для работы с учетными записями

Учетные записи пользователей можно создавать, изменять и удалять, для чего в системах UNIX предусмотрены специальные команды. Напомню, что создание, удаление и изменение учетных записей может выполнять только суперпользователь root. Для версий FreeBSD и System V эти команды несколько отличаются, поэтому рассмотрим их отдельно. Вначале остановимся на командах, используемых в системах System V:

- `useradd` — создает новую учетную запись пользователя или изменяет информацию о нем;
- `userdel` — удаляет регистрационное имя пользователя из системы;
- `passwd` — изменяет пароль пользователя;
- `su` — выполняет команду с заменой идентификатора пользователя и группы;
- `login` — инициализирует сессию пользователя в системе;
- `id` — отображает реальный и эффективный идентификатор пользователя и группы;
- `pwconv`, `pwunconv`, `grpconv`, `grpunconv` — выполняют преобразование обычных и теневых файлов паролей и групп;
- `pwck` — проверяет целостность файла паролей.

Команда `useradd`, вызванная без опции `-D`, создает новую учетную запись пользователя, используя при этом параметры командной строки и предполагая умолчания для остальных параметров. Если команда завершается успешно, то в системе будет зарегистрирована новая учетная запись пользователя, создан для него домашний каталог, в который копируются файлы инициализации.

Вот наиболее часто используемые опции команды `useradd`:

- `-u` *`идентификатор`* — указывает идентификатор пользователя (uid), представляющий собой неотрицательное целое число,

меньшее по значению, чем системный параметр MAXUID. По умолчанию обычно используется следующий доступный uid из указанного диапазона. Например, если в системе используются uid с номерами от 100 до 105, то следующий будет равен 106 (идентификаторы, имеющие значения 0—99, зарезервированы системой и не могут использоваться);

- -o — эта опция позволяет создать дубликат uid — применять ее следует крайне осторожно, поскольку обеспечение безопасности системы из-за такой неоднозначности усложняется;
- -g *группа* — представляет собой целочисленный идентификатор или символьное имя существующей группы, которая устанавливается как основная (primary) для нового пользователя;
- -G *группа* — представляет собой несколько элементов списка, разделенных запятыми, каждый из которых является целочисленным идентификатором или символьным именем существующей группы, при этом список может состоять из одного элемента. Содержимое списка устанавливает принадлежность пользователя к дополнительным группам, которые могут быть определены с помощью команды newgrp;
- -d *каталог* — начальный (домашний) каталог нового пользователя. По умолчанию в качестве начального используется каталог *HOME/registration_name*, где *HOME* — базовый каталог для начальных каталогов новых пользователей, а *registration_name* — регистрационное имя нового пользователя;
- -s shell — полный путь к командному интерпретатору, используемому пользователем сразу же после регистрации. По умолчанию этому полю значение не присваивается, поэтому система использует стандартный командный интерпретатор /usr/bin/sh. Для командной оболочки shell нужно указывать существующий исполняемый файл;
- -c *комментарий* — любая текстовая строка, кратко описывающая регистрационное имя (обычно указывает фамилию и имя реального пользователя). Эта информация хранится в

записи пользователя в файле /etc/passwd, а размер данного поля не должен превышать 128 символов;

- `-m` — создает домашний каталог для нового пользователя, если таковой отсутствует. Если каталог уже существует, вновь созданный пользователь должен обладать правами доступа к указанному каталогу;
- `-k skel_dir` — выполняет копирование содержимого каталога skel_dir в начальный каталог нового пользователя вместо использования стандартного "шаблонного" каталога /etc/skel, содержащего стандартные файлы, определяющие среду работы пользователя. Каталог skel_dir должен существовать до выполнения операции;
- `-f` *`активно_дней`* — максимально допустимый интервал времени в днях между использованиями регистрационного имени пользователя, когда это имя еще не объявляется недействительным. Обычно в качестве значений указываются положительные целые числа;
- `-e` *`дата`* — дата, начиная с которой регистрационное имя пользователя нельзя будет использовать: после этой даты ни один пользователь не сможет войти в систему, введя данное регистрационное имя;
- `login` — строка символов, задающая регистрационное имя для нового пользователя. В ней не должны присутствовать символы двоеточия и перевода строки, а первый символ не должен быть прописной буквой.

Рассмотрим пример создания учетной записи пользователя с регистрационным именем user1, который будет работать в операционных системах Solaris и Linux. Как обычно, для создания учетной записи пользователя необходимо зарегистрироваться в системе как суперпользователь root.

Вначале просмотрим опции по умолчанию для команды `useradd` — эта информация может оказаться полезной при создании и модификации учетных записей. Они могут быть такими:

```
# useradd -D
GROUP=100
```

```
HOME=/home
INACTIVE=-1
EXPIRE=
SHELL=/bin/bash
SKEL=/etc/skel
```

Результат выполнения этой команды позволяет сделать несколько важных выводов:

- в качестве корневого каталога для вновь создаваемых пользователей выбран каталог /home;
- пустое поле значения параметра `EXPIRE` означает, что учетная запись пользователя никогда не будет заблокирована;
- в качестве командного интерпретатора для всех вновь создаваемых пользователей по умолчанию установлен /bin/bash.

Для создания учетной записи пользователя user1 введем команду:

```
# useradd user1
```

Если команда выполнена успешно, то учетная запись пользователя user1 будет зарегистрирована в системе, а в файл /etc/passwd будет добавлена примерно такая запись:

```
user1:x:2307:2307::/home/user1:/bin/bash
```

Команда `useradd` автоматизирует процесс регистрации пользователя, но можно сделать это вручную, если нужно установить какие-либо индивидуальные параметры для пользователя, например, командную оболочку или домашний каталог. Предположим, необходимо создать учетную запись пользователя с регистрационным именем user2.

Вначале просмотрим файл /etc/passwd/ на предмет поиска наибольшего значения идентификатора пользователя uid. Наибольшее значение uid, равное 2307, имеет вновь созданный пользователь user1, поэтому следующим значением uid может быть 2308.

Добавим в файл /etc/passwd запись о пользователе user2, введя команду `echo`:

```
# echo user2:x:2308:2308::/home/user2:/bin/bash >>
/etc/passwd
```

Далее создадим начальный каталог пользователя user2:

```
#mkdir /home/user2
```

Пользователя user2 сделаем владельцем каталога /home/user2:

```
# chown user2 /home/user2
```

Приводим в соответствие записи файлов /etc/passwd и /etc/shadow с помощью команды `pwconv`:

```
#pwconv
```

Команда `pwconv` создает файл shadow из passwd, при этом может использоваться и существующий файл shadow (он будет перезаписан). Команда работает следующим образом:

1. Удаляются записи в теневом файле shadow, отсутствующие в основном файле паролей passwd.
2. Обновляются теневые записи, для которых в полях пароля в основном файле не стоит "x". Добавляются все недостающие теневые записи.
3. Пароли в основном файле заполняются символами "x".

Удалить учетную запись пользователя в операционных системах System V можно с помощью команды `userdel` — она удаляет информацию о пользователе из системы, выполняя соответствующие изменения в регистрационных файлах и файловой системе.

Дополнительно `userdel` запоминает идентификатор uid удаляемого пользователя в файле /etc/security/ia/ageduid, чтобы исключить повторное использование этого идентификатора в течение определенного периода времени — такой механизм называется "устареванием идентификатора" (uid aging).

Команда имеет синтаксис:

```
userdel [-r] [-n_месяцев] имя
```

Опции имеют такой смысл:

- `-r` — удаление начального каталога пользователя из системы (каталог должен существовать). При успешном выполнении

команды файлы и подкаталоги в домашнем каталоге будут недоступны;

- `-n` *месяцев* — задает интервал времени в месяцах, указывающий, как долго идентификатор пользователя должен устаревать перед повторным использованием. Если параметр равен −1, то идентификатор пользователя никогда не будет повторно использован, если он равен 0, то идентификатор пользователя можно использовать немедленно. Если опция `-n` не задана, принимается значение устаревания по умолчанию.

Изменить параметры учетной записи пользователя в системах System V можно при помощи команды `usermod`. Эта команда модифицирует файлы, содержащие информацию об учетных записях пользователей. Допустимы следующие опции:

- `-A` *метод*`|DEFAULT` — указывает новый метод идентификации пользователя и представляет собой имя программы, отвечающей за допустимую идентификацию пользователя. Строку `DEFAULT` можно использовать для установки стандартного метода идентификации;
- `-c` *комментарий* — указывает на другой комментарий для записи пользователя в файле паролей;
- `-d` *домашний_каталог* — новый домашний каталог пользователя. При указании опции `-m` содержимое текущего домашнего каталога будет перемещено в новый домашний каталог, который будет создан, если еще не существует;
- `-e` *дата* — дата, после которой учетная запись пользователя устареет. Дата указывается в формате `MM/DD/YY`;
- `-f` *активно_дней* — число дней между датой устаревания пароля и датой, когда учетная запись пользователя будет заблокирована. Значение, равное 0, блокирует учетную запись пользователя в момент устаревания пароля, а значение −1 запрещает блокировку (значение по умолчанию);
- `-g` *группа* — имя группы или номер группы, которые будут присвоены пользователю после входа в систему, причем группа с указанным именем должна существовать. Номер

группы также должен ссылаться на существующую группу (по умолчанию равен 1);

- `-G` *`дополнительная_группа`* — список дополнительных групп. Данный пользователь также является членом этих групп. Каждая группа отделяется от следующей группы запятой, без пробелов. Группы являются предметом для некоторых ограничений, например, группа, заданная с опцией `-g`. Если пользователь является членом группы, которая не находится в списке, то пользователь будет удален из группы;
- `-l` *`новое_имя`* — имя пользователя будет изменено с *`имя`* на *`новое_имя`*. Ничего другого сделано не будет. В частности, домашний каталог пользователя должен быть, вероятно, изменен;
- `-s shell` — имя командного интерпретатора, который будет использоваться новым пользователем при входе в систему. Установка этого поля в пустое значение будет выбирать системный shell по умолчанию;
- `-u` *`uid`* — числовое значение идентификатора пользователя uid. Это значение должно быть уникальным, исключение составляет использование опции `-o`. Значение должно быть положительным. Как было сказано ранее, значения между 0 и 99 обычно зарезервированы для системных бюджетов. Для любых файлов, владельцем которых является пользователь, и которые находятся в домашнем каталоге пользователя, идентификатор пользователя uid будет изменяться автоматически. Для файлов вне домашнего каталога пользователя идентификатор пользователя должен быть изменен вручную.

Важное замечание: если пользователь находится в системе, изменить его имя не удастся.

Вот пример использования команды `usermod`. Предположим, требуется изменить регистрационное имя пользователя user2, созданного ранее, на moduser2, а его домашний каталог — на /home/moduser2. Исходная запись для пользователя user2 в файле /etc/passwd выглядит так:

```
user2:x:2308:2308::/home/user2:/bin/bash
```

Следующая команда выполняет все необходимые изменения:

```
# usermod -l moduser2 -d /home/moduser2 -m user2
```

После выполнения этой команды запись для пользователя moduser2 в файле /etc/passwd должна выглядеть примерно так:

```
moduser2:x:2308:2308::/home/moduser2:/bin/bash
```

Легко проверить и наличие домашнего каталога пользователя moduser2, задав команду:

```
# ls -l /home
drwxr-xr-x   10 moduser2 root    4096 Aug  18 22:51 moduser2
```

Здесь нужно сделать одно важное замечание: при изменении регистрационного имени пользователя его uid остается неизменным. Это свидетельствует о том, что операционная система работает с одной и той же учетной записью пользователя, несмотря на то, что регистрационное имя пользователя изменилось. Таким образом, можно сделать очень важный вывод: для UNIX определяющим фактором при работе с пользователем является его идентификатор uid, а не регистрационное имя пользователя, которое может изменяться.

Проанализируем команду `passwd` — с ее помощью можно изменить пароли пользователей, при этом обычные пользователи могут изменить пароль только для своей учетной записи, в то время как суперпользователь root может это сделать для любого пользователя. Кроме этого, `passwd` позволяет изменить информацию об учетной записи: полное имя пользователя, его командный интерпретатор, дату истечения срока используемого пароля и интервал времени, в течение которого пароль действует.

Команда имеет синтаксис:

```
passwd [-f] имя
passwd [-g] [-r|-R] группа
passwd [-x max] [-n min] [-w warn] [-i inact] имя
passwd {-l|-u|-d|-S} имя
```

Для пользователей, имеющих пароль, перед установкой нового пароля команда `passwd` предлагает ввести текущий пароль (он хранится в зашифрованном виде). Обычному пользователю дается только одна попытка для ввода правильного пароля. Суперпользователь root может пропустить этот шаг, что оказывается полезным, если пароль забыт, поскольку его можно изменить даже в этом случае. После ввода пароля `passwd` проверяет наличие разрешения на изменение пароля в данное время — если это невозможно, команда завершает работу, не изменив пароль.

Вот смысл некоторых опций команды `passwd`:

- `-g` — замена пароля для заданной группы — может быть выполнена только суперпользователем root или администратором группы. Может быть использована вместе с опцией `-r` для удаления текущего пароля заданной группы, что делает группу доступной всем членам. Вместе с опцией `-R` используется для ограничения доступа к группе всем пользователям;
- `-x` — используется для установки максимального числа дней, в течение которых пароль остается допустимым, при этом после *`max`* дней требуется его изменение;
- `-n` — служит для установки минимального числа дней, после истечения которых пароль может быть изменен, при этом пользователю запрещается изменять пароль в течение *`min`* дней;
- `-w` — предназначена для установки числа дней, в течение которых пользователь будет получать предупреждающее сообщение об истечении времени действия его пароля, причем сообщения будут выводиться в течении *`warn`* дней, напоминая пользователю, сколько дней осталось до момента устаревания его пароля;
- `-i` — запрещает использование учетной записи пользователя по истечению промежутка времени после устаревания пароля, при этом, если устаревший пароль остается неизменным в течение *`inact`* дней, он не будет вновь принят системой;
- `-l` — блокирует учетную запись, изменяя пароль таким образом, что он становится непригодным для шифрования;

- `-u` — разблокирует учетную запись, изменяя пароль к его предыдущему значению;
- `-S` — вывод статусной информации учетной записи. Статусная информация состоит из шести полей, первое из которых кодируется следующим образом:
  - `L` — если бюджет пользователя заблокирован;
  - `NP` — если не существует пароля для данной учетной записи;
  - `P` — если пароль используется.

  Второе поле указывает дату последнего изменения пароля, а следующие четыре поля — минимальное время до истечения срока действия пароля, максимальное время до истечения срока действия пароля, период вывода предупреждающего сообщения об истечении срока действия пароля и период неактивности для этого пароля;
- `-f` — требует от пользователя изменить пароль при следующем входе в систему.

От выбора пароля во многом зависит безопасность операционной системы. Кроме того, существенную роль в этом играет алгоритм шифрования и размера ключа, используемый в данной UNIX-системе: в большинстве операционных систем метод криптографии основывается на алгоритме NBS DES, который имеет очень высокую степень безопасности, при этом размер ключа зависит от выбранного пароля.

Лучше не выбирать пароль, в котором используются литературные выражения, или основанный на личных данных (месяц, год рождения и т. д.) — такой пароль злоумышленнику расшифровать несложно.

Конечно, пароль должен быть хорошо запоминаем, поэтому одним из вариантов может быть знакомое слово, части которого разделены специальными символами. Пароль может представлять собой комбинацию из двух слов, объединенных вместе и разделенных специальными символами или цифрами. Примерами таких паролей являются P!e%ter$sbu)rg и n*ew!Pas#s%w$ord.

Организация учетных записей в системах, совместимых с BSD, принципиально не отличается от остальных. Отличия заключаются в следующем:

- используется файл /etc/master.passwd, являющийся в некотором смысле аналогом файла /etc/shadow, используемого в системах, совместимых с System V. Файл /etc/master.passwd хранит ту же информацию, что и /etc/passwd, хотя имеются и некоторые отличия: здесь хранятся хеш-коды (шифры) пользовательских паролей, а также зашифрованные пароли пользователей, поэтому он доступен для чтения только суперпользователю root;
- для управления учетными записями применяются команды с иной мнемоникой, чем в System V.

Для создания учетных записей пользователей применяется утилита `adduser`, которая выводит подсказки с предлагаемыми настройками, при этом синтаксис команды во многом напоминает тот, что используется для `useradd`.

Команда `chpass` применяется для изменения учетных записей пользователей в FreeBSD. Она позволяет изменять параметры учетной записи, включая пароль, срок действия учетной записи и стандартный интерпретатор команд, и имеет следующий синтаксис:

```
chpass [-a список] [-p зашифрованный_пароль] [-e
срок_действия]
       [-s интерпретатор] [login]
```

Опции команды означают следующее:

- `-a` *`список`* — позволяет суперпользователю определять полную запись в формате /etc/passwd;
- `-p` *`зашифрованный_пароль`* — разрешает изменить пароль, предварительно зашифрованный командой `crypt`. Эта опция используется в командных файлах, содержащих команду `crypt` и передающих полученный результат команде `chpass`;
- `-e` *`срок_действия`* — задает срок действия учетной записи;

- `-s` *`интерпретатор`* — обеспечивает смену стандартного интерпретатора команд на указанный;
- *`login`* — задает модифицируемую учетную запись.

Чаще всего команда `chpass` используется без параметров или с единственным параметром `login` — в этом случае запускается редактор, с помощью которого можно изменить параметры учетной записи.

Учетные записи пользователей FreeBSD можно отредактировать вручную непосредственно в файле /etc/master.passwd, после чего распространить изменения на другие файлы с помощью команды `pwd_mkdb`:

```
#pwd_mkdb -p /etc/master_passwd
```

Еще одна команда — `rmuser` — служит для удаления пользователя и информации, связанной с данной учетной записью, и имеет такой синтаксис:

```
rmuser [-y] login
```

Команда выполняет последовательность действий:

- уничтожает процессы, инициированные пользователем;
- удаляет задания демона `cron`, запланированные пользователем;
- удаляет задания команды `at`, запланированные пользователем;
- удаляет относящиеся к пользователю записи из файлов паролей (/etc/passwd, /etc/master.passwd);
- удаляет почтовую очередь пользователя из каталога /var/mail;
- удаляет файлы пользователя из каталогов /tmp, /var/tmp, /var/tmp/vi.recover;
- удаляет учетную запись пользователя из всех групп в файле /etc/group и саму группу, если пользователь является ее единственным членом;
- интерактивно позволяет удалить начальный каталог пользователя.

Группы пользователей в системе FreeBSD можно создавать либо с помощью программы `sysinstall`, либо вручную, редактируя файл /etc/group. Как видно из обзора, команды управления учетными записями пользователей System V и FreeBSD очень похожи и используют однотипные параметры.

Рассмотренные здесь команды являются основными для управления учетными записями пользователей, хотя кроме них имеется целый ряд других утилит, позволяющих выполнить более узкие задачи. Дополнительную информацию о таких командах можно получить из man-страниц операционной системы.

4.2. Программное управление учетными записями

Управление учетными записями пользователей в операционной системе UNIX возможно как с помощью инструментов командной строки, так и с использованием интерфейса API или библиотечных функций языка C — в этих случаях пользователи могут реализовать свои, довольно сложные и изощренные алгоритмы управления учетными записями пользователей, а также получать различного рода информацию, касающуюся учетных записей.

В языке C определен набор библиотечных функций, которые не имеют прямого соответствия в интерфейсах прикладного программирования (API), но, тем не менее, довольно широко используются при разработке программ для UNIX. Особое внимание хочу обратить на то, что в C имеется целый ряд библиотечных функций, не определенных в стандарте ANSI C, но, тем не менее, доступных во всех UNIX-системах.

В примерах программ, представленных в этой и последующих главах, мы будем использовать как системные вызовы UNIX, так и библиотечные функции C. Программы, представленные в этой и последующих главах, будут работать во всех наиболее популярных операционных системах (Linux, FreeBSD, Solaris и т. д.). Компиляцию исходных текстов программ можно выполнить

стандартными средствами, входящими в состав UNIX. Например, в операционной системе Linux, равно как и в других системах, можно использовать популярный пакет g++ компилятора C со стандартными опциями.

Рассмотрим несколько примеров программ на языке C, в которых показано, как можно получить ту или иную информацию об учетных записях пользователей.

Следующий фрагмент программного кода позволяет отобразить на экране дисплея домашний каталог пользователя, регистрационное имя которого указано в качестве аргумента программы. Исходный текст программы представлен в листинге 4.1.

Листинг 4.1. Вывод имени домашнего каталога пользователя

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>

int main(int argc, char* argv[])
{
 struct passwd *pwd;
 if (argc != 2)
 {
  printf("Usage: %s registration_name\n", argv[1]);
  exit(0);
 }
 pwd = getpwnam(argv[1]);
 if (!pwd)
 {
  printf("%s is not a valid user name!\n", argv[1]);
  exit(1);
 }
```

```
 printf("Home directory for registration name %s is %s\n",
argv[1], pwd->pw_dir);
 return 0;
}
```

В заголовочном файле `<pwd.h>` определяется набор функций, предназначенных для получения информации об учетной записи пользователя, содержащейся в файле /etc/passwd, и, кроме того, здесь же определена структура `passwd`, в поля которой помещается информация из файла /etc/passwd.

Структура имеет формат:

```
struct passwd
 {
  char*     pw_name
  char*     pw_passwd
  int       pw_uid
  int       pw_gid
  char*     pw_age
  char*     pw_comment
  char*     pw_dir
  char*     pw_shell
}
```

Поля структуры имеют такой смысл:

- `pw_name` — регистрационное имя пользователя;
- `pw_passwd` — зашифрованный пароль;
- `pw_uid` — идентификатор пользователя;
- `pw_gid` — идентификатор (gid) группы;
- `pw_age` — минимальный срок действия пароля;
- `pw_comment` — общая информация о пользователе;
- `pw_dir` — начальный каталог пользователя;
- `pw_shell` — регистрационный командный интерпретатор shell.

В данном примере используется функция `getpwnam()`, имеющая синтаксис:

```
const struct passwd* getpwnam(const char* имя_пользователя)
```

Здесь *имя_пользователя* — регистрационное имя пользователя. Функция заполняет поля структуры `passwd` информацией о данной учетной записи.

В другом примере приведен исходный текст программы, которая выводит на экран дисплея регистрационное имя пользователя и путь к регистрационному командному интерпретатору при заданном значении идентификатора `uid` пользователя, который является единственным параметром программы (листинг 4.2).

Листинг 4.2. Вывод имени пользователя и пути к интерпретатору shell

```
int main(int argc, char* argv[])
{
 struct passwd *pwd;
 int uid;

 if (argc != 2)
 {
  printf("Usage: %s uid\n", argv[0]);
  exit(0);
 }
 uid = atoi(argv[1]);
 pwd = getpwuid(uid);

 if (!pwd)
 {
  printf("%s is not a valid user UID!\n",argv[1]);
  exit(1);
 }
```

```
 printf("Registration shell for user with uid = %d is: %s\n",
pwd->pw_uid, pwd->pw_shell);
 printf("USER_NAME for user with UID = %d is: %s\n",
 pwd->pw_uid, pwd->pw_name);
 return 0;
}
```

В этой программе используется функция `getpwuid()`, синтаксис которой таков:

```
const struct passwd* getpwuid(const int uid)
```

В качестве параметра функция принимает значение идентификатора пользователя *uid*.

Программа, исходный текст которой показан в листинге 4.3, выводит на экран дисплея имена и идентификаторы пользователей, записи о которых находятся в файле /etc/passwd.

Листинг 4.3. Вывод регистрационных имен и идентификаторов всех пользователей

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>

int main(void)
{
 struct passwd *pwd;
 setpwent();
 while(pwd = getpwent())
  {
   printf("Registration name:%s, uid:%d\n", pwd->pw_name,
   pwd->pw_uid);
  }
```

```
 endpwent();
 return 0;
}
```

В этой программе используются функции `setpwent()`, `getpwent()` и `endpwent()`. Функция `setpwent()` устанавливает указатель чтения на начало файла /etc/passwd, функция `getpwent()` смещает указатель на следующую запись файла /etc/passwd, а функция `endpwent()` закрывает файл /etc/passwd.

Для отображения идентификатора группы, зная ее имя, можно воспользоваться программой, исходный текст которой представлен в листинге 4.4. Программа принимает единственный параметр, которым является имя группы.

Листинг 4.4. Отображение идентификатора группы

```
#include <stdio.h>
#include <stdlib.h>
#include <grp.h>

int main(int argc, char* argv[])
{
 struct group *grp;
 if (argc != 2)
 {
  printf("Usage: %s [group_name]\n", argv[0]);
  exit(0);
 }
 grp = getgrnam(argv[1]);
 printf("gid = %d for group %s\n", grp->gr_gid,
 grp->gr_name);
 return 0;
}
```

В файле заголовка <grp.h> определяется набор функций, предназначенных для получения информации о группах, содержащейся в файле /etc/group. Кроме того, здесь же определена структура group, в поля которой помещается информация из файла /etc/group. Структура имеет формат:

```
struct group
 {
  char*     gr_name
  char*     gr_passwd
  int       gr_gid
  char*     pw_comment
}
```

Здесь:

- gr_name — имя группы;
- gr_passwd — зашифрованный пароль группы;
- gr_gid — идентификатор группы gid;
- pw_comment — имена членов группы.

Функция getgrnam() принимает в качестве аргумента имя группы и возвращает указатель на запись типа struct group, которая содержит информацию о группе, если группа определена в системе.

В последнем примере (листинг 4.5) показан исходный текст программы, которая отображает на экране дисплея имя группы и ее идентификатор, содержащиеся в файле /etc/group.

Листинг 4.5. Вывод имени группы и ее идентификатора на экран

```
#include <stdio.h>
#include <stdlib.h>
#include <grp.h>

int main(void)
{
```

```
 struct group *grp;
 setgrent();
 while (grp = getgrent())
 {
  printf("Group name : %s, gid: %d\n", grp->gr_name,
  grp->gr_gid);
 }
 endgrent();
 return 0;
}
```

В этой программе функция `setgrent()` устанавливает указатель чтения файла на начало файла /etc/group, функция `getgrent()` смещает на следующую запись файла /etc/group, а функция `endgrent()` закрывает файл /etc/group.

Рассмотренные примеры демонстрируют только небольшую часть тех возможностей, которые предоставляет операционная система для создания собственных алгоритмов управления учетными записями пользователей и получения различного рода информации о пользователях.

В состав библиотечных функций C включена группа функций, предназначенных для шифрования и дешифрования данных. Функции этой группы очень важны, поскольку позволяют обеспечить безопасность системы.

Так, например, файлы пользовательских паролей и системных данных, которым необходима высокая степень защиты, обычно должны храниться в зашифрованном виде. Файл заголовка crypt.h определяет несколько функций шифрования/дешифрования — `crypt()`, `setkey()` и `encrypt()`. Функция `crypt()` используется в UNIX-системах для шифрования пользовательских паролей и проверки действительности пароля пользователя.

Функции `setkey()` и `encrypt()` выполняют действия, аналогичные тем, которые выполняет функция `crypt()`, с той лишь разницей, что в них используется алгоритм шифрования данных по стандарту DES, который более надежен, чем тот, что используется функцией `crypt()`.

Глава 5

Установка, запуск и функционирование UNIX

Материал этой главы посвящен анализу общих принципов инсталляции, запуска и бесперебойной работы операционных систем UNIX. Инсталляция или, по-другому, установка операционной системы — это процесс, выполняемый один раз, по крайней мере, до того момента, пока не потребуется повторная установка системы.

Анализ процесса установки системы мы не будем привязывать к какой-либо конкретной аппаратной платформе, например, для SPARC или x86 — здесь будут рассмотрены важнейшие этапы инсталляции UNIX, общие для всех систем.

От того, насколько успешно выполнена установка, зависит дальнейшее функционирование операционной системы UNIX. Многие проблемы, возникающие при работе операционных систем, часто связаны с некорректными установками тех или иных параметров системы во время инсталляции системы, причем может случиться так, что для устранения таких проблем может потребоваться даже повторная установка UNIX.

К счастью, большинство операционных систем предусматривают так называемый стандартный режим инсталляции, когда программа-установщик самостоятельно выбирает нужные параметры системы, требуя минимального вмешательства пользователя. Преимущество такого подхода состоит в том, что от пользовате-

ля не требуются глубокие знания архитектуры устанавливаемой системы, что позволяет выполнить установку UNIX даже новичкам. В этом смысле самой дружелюбной является операционная система Linux, которая без особых проблем устанавливается с дистрибутивных дисков, требуя минимального вмешательства пользователя. Операционные системы FreeBSD и Solaris более прихотливы при установке, и даже в стандартном режиме могут потребовать квалифицированного вмешательства в процесс инсталляции со стороны пользователя.

Даже при выборе стандартного режима инсталляции следует очень внимательно анализировать предлагаемые инсталлятором опции. Особенно это касается этапа разбивки дискового пространства для установки системы и выбора способа загрузки. Если на машине будет установлена только одна операционная система UNIX, то особо беспокоиться нечего, но если на компьютере уже есть другие установленные операционные системы, например, Windows, что чаще всего и бывает, то лучше всего выполнить разбивку пространства жесткого диска вручную, чтобы избежать случайного уничтожения другой операционной системы.

Инсталляторы современных систем UNIX отслеживают местоположение других операционных систем и для установки пытаются использовать свободное дисковое пространство, однако полагаться целиком на это не стоит.

Второй важный момент касается установки начального загрузчика системы. Опять-таки, если на машине устанавливается только UNIX-система, то можно разрешить инсталлятору установить загрузчик системы. При наличии ранее установленных операционных систем может случиться так, что после установки загрузчика UNIX остальные системы перестанут загружаться.

Многие операционные системы UNIX устанавливают в качестве начального загрузчика программу `grub`, которая позволяет выполнить мультизагрузку, настроив соответствующим образом файлы конфигурации. В качестве альтернативы можно установить на машину одну из программ-менеджеров загрузки, например, Osloader или System Commander, и настроить их для загрузки нескольких операционных систем.

В этой главе мы рассмотрим типичные вопросы, возникающие в процессе инсталляции различных операционных систем. Каждая из версий операционной системы (Linux, FreeBSD, Solaris и т. д.) имеет свои, отличные от других систем, нюансы инсталляции, поэтому не существует единого подхода к установке различных версий UNIX из-за различия в программной архитектуре. Процедуру инсталляции операционных систем рассмотрим на примерах FreeBSD, Solaris и Linux.

Перед инсталляцией операционной системы желательно выполнить несколько подготовительных шагов — это позволит в общих чертах определить параметры системы и оценить ее возможную конфигурацию:

1. Изучить документацию по конкретной операционной системе, чтобы понимать, хотя бы в общих чертах, особенности функционирования данной версии UNIX — это сделает понятными шаги, которые выполняются в процессе инсталляции. Следует обратить внимание на требования к аппаратной части — это позволяет оценить, пусть даже приблизительно, возможную производительность операционной системы на имеющемся оборудовании. Кроме того, хорошие знания аппаратной части очень часто помогают при выборе правильных настроек в процессе инсталляции, что, в конечном счете, оптимизирует производительность системы.
2. Определить функциональное назначения устанавливаемой системы — будет ли это файл-сервер, прокси, почтовый сервер и т. д. — это нужно для правильной оценки имеющихся аппаратных ресурсов и выбора необходимых опций инсталляции. Большинство версий UNIX позволяют устанавливать серверный или клиентский вариант операционной системы. Серверный вариант предполагает, что операционная система должна функционировать как провайдер (сервер) услуг, которые будут предоставляться клиентским системам. Так, например, сервер имен DNS обеспечивает функционирование службы имен в сети TCP/IP, сервер DHCP — динамическое выделение IP-адресов клиентским станциям и т. д. Поскольку серверы могут обслуживать запросы многих клиентов, то аппаратные ресурсы такой системы должны быть намного

мощнее, чем у обычной клиентской станции. Клиентский вариант операционной системы рассчитан больше на работу конечного пользователя (разработка приложений, работа с офисными пакетами программ и т. д.), поэтому для его нормального функционирования не требуется значительных аппаратных ресурсов. Это вовсе не означает, что серверный вариант операционной системы не подходит для работы клиентов — просто серверные конфигурации UNIX оптимизированы больше для работы в сети, чем для выполнения, например, приложений пользователя.

3. Хотя бы приблизительно определить, какое дополнительное программное обеспечение может понадобиться в дальнейшем и достаточно ли будет для этого ресурсов.

Во многих случаях инсталлятор предлагает выбрать тип конфигурации системы — стандартный или пользовательский. Если вы недостаточно четко представляете себе назначение отдельных настроек, то лучше выбрать стандартный вариант установки. Изучив систему, можно будет в любой момент установить дополнительное программное обеспечение, если оно не было установлено при выборе стандартного варианта установки.

Очень важно документировать каждый шаг инсталляции — это окажет неоценимую помощь при анализе ошибок в системе. Неправильно установленные параметры системы на этапе инсталляции могут стать причиной постоянно возникающих без видимых причин сбоев в работе системы. Значение документирования процесса инсталляции очень часто недооценивают и начинают понимать его пользу только после нескольких часов бесплодных попыток найти ту или иную аномалию в работе системы.

5.1. Этапы установки системы

Процессы инсталляции различных версий операционных систем UNIX во многом схожи и включают несколько основных этапов (рис. 5.1).

Рис. 5.1. Этапы инсталляции операционных систем UNIX

В большинстве типов инсталляционных программ используется графический интерфейс пользователя, поэтому в начале процедуры установки запрашиваются данные о типе графической карты, монитора и мыши. Инсталлятор пытается вначале определить тип оборудования и, если это удается, то отображает информацию о выбранном типе устройства. Иногда определить тип оборудования не удается (чаще всего такое случается с графическими картами), поэтому пользователь должен самостоятельно вводить подходящий тип. Здесь может пригодиться подробная информация о дистрибутиве устанавливаемой системы UNIX, в которой, как правило, указывается перечень поддерживаемых типов оборудования.

Вообще, все дистрибутивы операционной системы позволяют установить тип оборудования по умолчанию, который можно использовать в случаях, если не удается установить оригинальный тип. Например, при невозможности определить тип гра-

фической карты можно установить стандартный тип видеоадаптера, позволяющий в любом случае запустить графическую подсистему.

Для работы графического интерфейса инсталлятора требуется определение типов клавиатуры, мыши, графической карты и дисплея. В некоторых инсталляторах при невозможности установить графический режим процедура установки может быть продолжена в режиме текстовой консоли.

При выполнении инсталляции все программы установки требуют указывать пароль суперпользователя root, а в некоторых случаях и добавления нового пользователя в систему. Характеристиками, идентифицирующими систему в сети, являются имя хоста и его IP-адрес. Здесь все инсталляторы предлагают выбор: либо автоматическое конфигурирование при помощи сервера DHCP, либо ручная установка IP-адреса. Если указана настройка DHCP, то при запуске системы она запросит адрес у сервера DHCP, находящегося где-нибудь в сети. Кроме этой, имеются, как правило, и другие сетевые настройки, требующие, например, указания адреса маршрутизатора, сервера имен (DNS) или альтернативного имени хоста. В некоторых операционных системах на этапе установки можно конфигурировать и протоколы высокого уровня, например, telnet и FTP.

На этапе конфигурирования сети не обязательно указывать все настройки — можно сделать это позже во время эксплуатации системы.

Следующим, пожалуй, самым важным и ответственным этапом, является конфигурирование файловой системы UNIX. В свою очередь, конфигурирование файловой системы можно представить в виде последовательности шагов (см. рис. 5.1).

1. Выбор жесткого диска или раздела на жестком диске для установки системы. Инсталляторы большинства систем обычно предлагают установить систему на свободное дисковое пространство. Здесь очень важно знать, какой объем пространства требуется для инсталляции конкретной конфигурации UNIX. Обычно для полной инсталляции систем пространства 3—5 Гбайт на жестком диске вполне достаточно. В практиче-

ском плане желательно иметь достаточный запас свободного дискового пространства, которое может понадобиться в случае установки дополнительного программного обеспечения.

2. Выбор структуры файловой системы. Здесь можно согласиться с предложением инсталлятора и выбрать конфигурацию файловой системы автоматически, хотя не всегда иерархия файлов в такой системе будет совпадать с той, что принята в "классической" UNIX. Например, в некоторых версиях UNIX при инсталляции дерево каталогов может оказаться неполным из-за отсутствия каталогов /var или /usr/local, хотя принципиального значения это не имеет.
   В процессе автоматической разбивки инсталлятор будет выделять для каждой файловой системы определенный объем дискового пространства.
3. Создание файловой системы. На этом этапе создается файловая система того типа, которая поддерживается данной версией UNIX. Обычно программа инсталляции кроме создания самой файловой системы резервирует место для области свопинга, являющегося частью механизма виртуальной памяти.
4. После создания файловой системы инсталляторы обычно предлагают сделать раздел операционной системы UNIX активным, т. е. таким, который будет загружаться при включении системы. Если на вашей машине установлено еще несколько операционных систем, то лучше этого не делать, а для загрузки нескольких систем воспользоваться одной из многочисленных программ-менеджеров загрузки, имеющихся в Интернете.

Все инсталляторы предлагают, как правило, несколько вариантов конфигурации программного обеспечения, исходя из предназначения системы. Для обычных пользователей программная конфигурация может быть более простой, в то время как для разработчиков программного обеспечения может потребоваться, например, установка дополнительных пакетов разработки на языках высокого уровня или дополнительных библиотек функций. Независимо от выбираемой конфигурации в набор устанавливаемых пакетов программ входят стандартные компиляторы

C++ (обычно это GNU-версия g++) и интерпретатор языка Perl. В большинство дистрибутивов систем (если это не загружаемые с оптического диска системы, так называемые Live-CD) включается и очень популярный компилятор Java.

Далее мы рассмотрим наиболее важные аспекты установки операционных систем. В следующем разделе мы проанализируем общие основы создания файловых систем UNIX.

5.2. Основы создания файловых систем

Ключевым моментом в инсталляции UNIX является создание файловой системы. Многие пользователи, особенно начинающие, с трудом представляют себе, как происходит этот процесс, особенно учитывая то, что инсталляторы современных систем скрывают детали процесса от пользователя. С одной стороны, это не так уж и плохо, поскольку в этом случае исключается потенциальная угроза ошибок пользователя, могущих привести к разрушению содержимого разделов диска, но, с другой стороны, это не позволяет глубже понять, что же происходит во время создания файловой системы. Такие знания не помешают, особенно при необходимости восстановления поврежденных файловых систем.

В основе создания файловых систем и управления ими лежат определенные принципы, одинаковые для всех операционных систем и используются специальные утилиты, имеющие одинаковые названия и одни и те же функциональные особенности, например, `fdisk`, `mkfs`, `newfs`, `mount`. Инсталляторы операционных систем используют эти утилиты неявным образом, скрывая детали их работы от пользователя.

Перед тем как создать файловую систему на некотором диске, необходимо понимать основы геометрии дискового накопителя. Жесткие диски поставляются различных форм и размеров, а количество головок, дорожек, секторов, т. е. емкость диска изменяются от одной модели к другой.

Жесткий диск состоит из нескольких отдельных дисковых пластин, смонтированных на общем шпинделе. Данные, которые хранятся на каждой поверхности пластины, записываются и считываются с помощью головок диска. Круговая траектория, которую головка диска отслеживает над поверхностью вращающейся пластины, называется *дорожкой*.

Каждая дорожка образована из нескольких секторов, которые покрывают ее от начала до конца. *Сектор* состоит из заголовка, трейлера (записи с контрольной суммой в конце сектора) и 512 байтов данных. Заголовок и трейлер содержат информацию для контроля ошибок, которая помогает обеспечить точность данных. Собранный вместе набор дорожек, которые проходят сквозь каждую поверхность отдельных дисковых пластин для одного и того же положения головок, называется *цилиндром*.

С каждым диском связан *контроллер* — интеллектуальное устройство, отвечающее за организацию данных на этом диске. Контроллеры некоторых дисков размещаются на отдельной плате электронной схемы (контроллеры дисков с интерфейсом SCSI), а во многих случаях контроллеры интегрированы с дисковым накопителем (контроллеры дисков с интерфейсом IDE/EIDE).

Диски могут содержать области, на которые данные не могут быть надежно записаны или считаны из них. Такие области носят название *дефектов*. Контроллер использует информацию, которая содержится в трейлере каждого дискового блока, для определения того, имеются ли в данном блоке дефекты или нет. Когда некоторый блок диагностируется как дефектный, контроллер может получить команду на добавление этого блока к списку дефектов и исключение его из использования в дальнейшем. Последние два цилиндра диска выделяются для решения диагностических задач и хранения списка дефектов данного диска.

На каждом диске выделяется специальная область для хранения информации о контроллере данного диска, его геометрических параметрах и разбивке. Эта информация носит название *метки* данного *диска* или VTOC (Volume Table Of Contents — оглавление тома).

Разметить диск — это значит записать на него информацию о разбивке. Обычно метка присваивается после определения разделов диска. Если не присвоить метку диску после создания разделов, то эти разделы будут недоступны, поскольку операционная система не имеет возможности узнать об их существовании.

Диски делятся на области, которые называются *частями* или *разделами диска*. Раздел диска образуется единой совокупностью смежных боков. Первоначально поддерживались только четыре раздела на диск. Они называются *главными*. Чтобы обойти это ограничение и дать возможность создавать более чем четыре раздела, был создан новый тип раздела — *расширенный* (extended) *раздел*. Диск может содержать только один расширенный раздел. Специальные разделы, называемые *логическими разделами*, могут быть созданы внутри расширенного раздела.

Каждый раздел имеет идентификатор (ID) раздела — номер, который используется для определения типа данных раздела. FreeBSD, например, устанавливает ID раздела равным 165. Каждая операционная система, с которой вы работаете, определяет разделы своим способом. Например, операционные системы линейки Windows присваивают каждому главному и логическому разделу букву диска, начиная с C.

Операционные системы UNIX нужно устанавливать в главный раздел. Если все разделы на диске уже заняты, нужно освободить один из них, используя программы, поставляемые с имеющейся операционной системой (например, fdisk для MS-DOS или Windows). Возможно, сначала придется уменьшить размер одного или нескольких существующих разделов с помощью одной из коммерческих программ, например, PartitionMagic.

Утилита PartitionMagic — это одна из немногих программ, способных изменять размер файловой системы NTFS. FIPS. Программу PartitionMagic и другие утилиты разбивки дисков на разделы следует применять очень осторожно — сбои по питанию и другие проблемы могут привести к разрушению структур файловой системы и потере данных. Все это может сделать диск недоступным для работы. По этой причине желательно выполнять резервное копирование данных перед запуском этих программ.

Инсталляционные программы операционных систем UNIX позволяют создавать файловые системы либо в автоматическом, либо в ручном (custom) режиме. Второй режим лучше использовать только опытным пользователям, тогда как первый подходит больше для новичков. Но даже в ручном режиме многие операции по созданию файловой системы скрыты от пользователя и выполняются в режиме диалога, когда требуется подтверждать тот или иной выбор.

Остановимся более подробно на практических аспектах создания файловых систем в различных версиях UNIX. Рассмотрим создание файловых систем в двух популярных версиях этой операционной системы — Linux Mandrake и FreeBSD. Показанные примеры могут помочь и при работе с другими операционными системами, поскольку методика создания файловых систем во всех версиях UNIX приблизительно одинакова, как одинаковы и программы, участвующие в этом процессе.

Предварительно хочу заметить, что создание файловых систем — дело довольно тонкое, поэтому приступайте к нему, только тщательно изучив сопутствующую документацию по организации файловой системы. Кроме того, вы должны неплохо разбираться в аппаратной части системы, особенно в вопросах, касающихся работы дисковых накопителей.

5.2.1. Файловая система UFS

Файловая система UFS (ufs) основана на традиционной файловой системе BSD FAT Fast и используется в качестве базовой практически во всех версиях UNIX. Кроме нее используется и целый ряд других типов файловых систем, но их мы рассматривать не будем, сосредоточив внимание именно на UFS.

Необходимость в создании (или модификации) файловой системы UFS может возникнуть в одном из следующих случаев:

- добавления или замены накопителей на жестких дисках;
- изменения части существующего раздела;
- выполнения полного восстановления файловой системы;

- изменения параметров файловой системы, таких, например, как размер блока или размер свободного пространства.

При создании файловой системы UFS вначале создается раздел, в котором выделяются группы цилиндров. Затем создаются блоки для контроля и организации структуры файлов внутри данной группы цилиндров.

В файловой системе каждый блок выполняет специфическую функцию. Файловая система UFS имеет следующие четыре типа блоков:

- загрузочный блок (boot block) — хранит информацию, которая используется при выполнении загрузки системы;
- суперблок (superblock) — хранит важную информацию о файловой системе;
- индексный дескриптор (inode) — хранит всю информацию о файле за исключением его имени;
- блок данных — хранит данные каждого файла.

Рассмотрим более подробно назначение каждого из этих типов блоков.

Загрузочный блок хранит процедуры, используемые при выполнении загрузки системы. Без загрузочного блока система не может быть загружена. Если файловая система не используется при загрузке системы, загрузочный блок остается пустым. Загрузочный блок находится только в первой группе цилиндров и занимает первые 8 Кбайт дискового пространства.

Суперблок хранит основную информацию о файловой системе. Наиболее важными структурами данных суперблока являются:

- размер и статус файловой системы;
- метка (имена файловой системы и тома);
- размер логического блока файловой системы;
- дата и время последнего обновления;
- размер группы цилиндров;
- количество информационных блоков в группе цилиндров;

- блок итоговой информации;
- состояние файловой системы (чистая, стабильная или активная);
- имя маршрута последней точки монтирования.

Без суперблока файловая система становится недоступной для чтения. Суперблок размещается в начале каждого раздела диска и дублируется в каждой группе цилиндров. Поскольку суперблок содержит критически важные данные, при создании файловой системы формируется множество суперблоков.

Копия суперблока каждой файловой системы постоянно находится в оперативной памяти и регулярно обновляется. Если система остановилась раньше, чем была обновлена копия суперблока на диске, самые последние изменения теряются и данная файловая система становится несогласованной (inconsistency).

Для немедленной записи данных суперблока, находящегося в оперативной памяти, на диск служит программа `sync`. Проверить целостность файловой системы можно, примерив программу `fsck`, которая позволяет обнаружить и исправить проблемы, возникающие в случаях, когда команда `sync` не была применена перед закрытием системы.

Вместе с суперблоком хранится блок итоговой информации. Этот блок не дублируется, но группируется с первым суперблоком, находящимся в первой группе цилиндров. В этом блоке регистрируются изменения, которые произошли в файловой системе в процессе ее использования. Кроме того, в этом блоке хранится информация о количестве индексных дескрипторов и каталогов.

Индексный дескриптор хранит всю информацию о файле за исключением имени (оно хранится в каталоге). Размер индексного дескриптора равен 128 байтов. Информация индексного дескриптора хранится в информационном блоке цилиндра и состоит из элементов, указывающих:

- тип файла (обычный, каталог, блочный специальный, символьный специальный, ссылка);

- режим файла (набор полномочий на чтение/запись/исполнение);
- количество жестких ссылок на данный файл;
- идентификатор владельца файла;
- идентификатор группы, к которой принадлежит файл;
- количество байтов в файле (размер файла);
- массив из 15 адресов блоков диска;
- дата и время последнего доступа к файлу;
- дата и время последнего изменения файла;
- дата и время создания файла.

Максимальное количество файлов в файловой системе UFS определяется количеством индексных дескрипторов, выделенных ей. Количество дескрипторов зависит от размера дискового пространства, выделяемого под каждый индексный дескриптор, и от общего размера файловой системы. По умолчанию один индексный дескриптор выделяется на каждые 2 Кбайт пространства данных.

Блоки хранения данных (блоки данных) занимают все остальное дисковое пространство, выделенное для файловой системы. Размер таких блоков определяется в момент создания файловой системы. По умолчанию блоки хранения выделяются с учетом двух размеров: логического блока размером 8 Кбайт и размера фрагментации, равного 1 Кбайт.

Блоки данных обычного файла хранят его содержимое, а блоки данных каталога — элементы каталога, указывающие на индексный дескриптор, а также имена файлов в каталоге.

Блоки, которые в настоящее время не используются, такие, например, как индексные дескрипторы, блоки косвенной адресации и блоки данных, помечаются в таблице группы цилиндров как свободные. Эта же таблица используется для предотвращения фрагментации, ведущей к снижению производительности диска.

5.2.2. Примеры создания файловых систем

Вначале рассмотрим создание файловой системы в Linux. Предположим, что у нас имеется дисковый накопитель SCSI с идентификатором, равным 2, на котором нужно создать первый раздел, т. е. имеющий номер 1. В операционных системах Linux первому SCSI-диску соответствует файл устройства /dev/sda, первому разделу — /dev/sda1, второму SCSI-диску — файл /dev/sdb и т. д.

Для создания разделов на жестком диске в Linux используется утилита `fdisk`. В качестве параметра программа `fdisk` принимает имя файла устройства. В нашем случае следует ввести команду:

```
# fdisk /dev/sdb

Command (m for help): m
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition
   l   list known partition types
   m   print this menu
   n   add a new partition
   o   create a new empty DOS partition table
   p   print the partition table
   q   quit without saving changes
   s   create a new empty Sun disklabel
   t   change a partition's system id
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit
   x   extra functionality (experts only)
```

Программа `fdisk` работает в интерактивном режиме, выполняя односимвольные команды, введенные пользователем (перечень и описание команд выводится при вводе `m`). Вот смысл некоторых команд:

- `l` — отображение списка типов разделов;
- `n` — создание нового раздела на диске;
- `d` — удаление раздела;
- `t` — замена идентификатора типа раздела;
- `a` — установка флага загрузки. Установка этого флага означает, что данный раздел является загрузочным;
- `v` — проверка таблицы раздела;
- `p` — отображение таблицы разделов;
- `w` — запись таблицы разделов на диск и выход из программы;
- `q` — выход из программы без сохранения изменений.

Вот как выглядит отображаемая информация о диске 2 (после ввода команды `p`):

```
Desirably to see the partition table before you
begin to do something.

Disk /dev/sdb: 4359 MB, 4359768576 bytes
255 heads, 63 sectors/track, 530 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id
   System
```

Перед разбивкой диска полезно просмотреть список идентификаторов разделов файловых систем, чтобы правильно указать идентификатор создаваемого раздела (команда `l`):

```
Command (m for help): l

 0  Empty           1c  Hidden W95 FAT3 70  DiskSecure Mult bb
    Boot Wizard hid
 1  FAT12           1e  Hidden W95 FAT1 75  PC/IX           be
    Solaris boot
```

```
 2  XENIX root      24  NEC DOS         80  Old Minix       c1
    DRDOS/sec (FAT-
 3  XENIX usr       39  Plan 9          81  Minix / old Lin c4
    DRDOS/sec (FAT-
 4  FAT16 <32M      3c  PartitionMagic  82  Linux swap      c6
    DRDOS/sec (FAT-
 5  Extended        40  Venix 80286     83  Linux           c7
    Syrinx
 6  FAT16           41  PPC PReP Boot   84  OS/2 hidden C:  da
    Non-FS data
 7  HPFS/NTFS       42  SFS             85  Linux extended  db
    CP/M / CTOS / .
 8  AIX             4d  QNX4.x          86  NTFS volume set de
    Dell Utility
 9  AIX bootable    4e  QNX4.x 2nd part 87  NTFS volume set df
    BootIt
 a  OS/2 Boot Manag 4f  QNX4.x 3rd part 8e  Linux LVM       e1
    DOS access
 b  W95 FAT32       50  OnTrack DM      93  Amoeba          e3
    DOS R/O
 c  W95 FAT32 (LBA) 51  OnTrack DM6 Aux 94  Amoeba BBT      e4
    SpeedStor
 e  W95 FAT16 (LBA) 52  CP/M            9f  BSD/OS          eb
    BeOS fs
 f  W95 Ext'd (LBA) 53  OnTrack DM6 Aux a0  IBM Thinkpad hi ee
    EFI GPT
10  OPUS            54  OnTrackDM6      a5  FreeBSD         ef
    EFI (FAT-12/16/
11  Hidden FAT12    55  EZ-Drive        a6  OpenBSD         f0
    Linux/PA-RISC b
12  Compaq diagnost 56  Golden Bow      a7  NeXTSTEP        f1
    SpeedStor
14  Hidden FAT16 <3 5c  Priam Edisk     a8  Darwin UFS      f4
    SpeedStor
16  Hidden FAT16    61  SpeedStor       a9  NetBSD          f2
    DOS secondary
17  Hidden HPFS/NTF 63  GNU HURD or Sys ab  Darwin boot     fd
    Linux raid auto
```

```
18  AST SmartSleep  64  Novell Netware  b7  BSDI fs         fe
    LANstep
1b  Hidden W95 FAT3 65  Novell Netware  b8  BSDI swap       ff
    BBT
```

При создании раздела в Linux (если это не область обмена, swap) обычно выбирают ID = 0x83 (в шестнадцатеричной системе).

Создадим основной (primary) раздел с номером 1 размером 250 Мбайт, в котором будет размещаться файловая система Linux:

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-530, default 1):
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-530, default
530): +250M
```

Проверим параметры созданного раздела:

```
Command (m for help): p


Disk /dev/sdb: 4359 MB, 4359768576 bytes
255 heads, 63 sectors/track, 530 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes


   Device Boot      Start       End      Blocks   Id  System
/dev/sdb1               1        31      248976   83  Linux
```

Если параметры раздела удовлетворяют требованиям, нужно выполнить последний шаг создания раздела — записать таблицу разделов на диск (команда `w`):

```
Command (m for help): w
The partition table has been altered!
```

```
Calling ioctl() to re-read partition table.
Syncing disks.
```

На этом первый этап создания файловой системы закончен — мы создали раздел на жестком диске, в котором будет размещаться файловая система Linux. Далее создаем саму файловую систему.

В Linux, как и в большинстве UNIX-систем, для создания файловых систем используется команда `mkfs` (хотя некоторые системы включают и более дружелюбную утилиту `newfs`). Для создания файловой системы в командной строке указывается ее тип (`ФС`), а также целый ряд специальных опций и операндов. Вот синтаксис команды `mkfs`:

```
mkfs [-F ФС][-V][-m] [-o опции] устройство размер [операнды]
```

Задаваемые опции и операнды зависят от типа создаваемой файловой системы. Основные из них представлены далее:

- `-F` — определяет тип создаваемой файловой системы. Он задается либо в командной строке, либо берется из файла, содержащего таблицу стандартных файловых систем (/etc/vfstab в System V, /etc/fstab в других версиях UNIX);
- `-V` — отображает введенную командную строку без выполнения самой команды. Командная строка создается как при помощи опций, указанных пользователем, так и путем добавления информации, взятой из таблицы стандартных файловых систем. Эта опция позволяет проверить корректность командной строки;
- `-m` — отображает командную строку, использованную для создания файловой системы, при этом файловая система уже должна существовать. Эта опция неприменима с другими параметрами;
- `-o` — задает опции, специфические для указанного типа физической файловой системы, например, специальное символьное устройство, на котором будет создана файловая система, или размер файловой системы в 512-байтовых блоках.

Создадим файловую систему в разделе /dev/sdb1:

```
# mkfs /dev/sdb1
mke2fs 1.34 (25-Jul-2003)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
62248 inodes, 248976 blocks
12448 blocks (5.00%) reserved for the super user
First data block=1
31 block groups
8192 blocks per group, 8192 fragments per group
2008 inodes per group
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345, 73729, 204801, 221185

Writing inode tables: done
Writing superblocks and filesystem accounting information:
done

This filesystem will be automatically checked every 29 mounts
or
180 days, whichever comes first.  Use tune2fs -c or -i to
override.
```

На этом создание файловой системы Linux в первом разделе жесткого диска можно считать законченным.

Перед использованием созданной файловой системы UNIX должна выполнить процедуру подключения или, в терминах UNIX, *монтирования*, для чего используется программа `mount` (включается во все версии UNIX). Команда `mount` связывает файловую систему из указанного раздела на диске с существующей иерархией файловых систем, а команда `umount` (демонтировать) выключает файловую систему из иерархии.

Таким образом, команда `mount` дает пользователям возможность обращаться к данным в дисковом разделе как к файловой системе, а не как к последовательности дисковых блоков. Все физические файловые системы, кроме корневой (/), считаются съемными в том смысле, что они могут быть как доступны для пользователей, так и не доступны.

Команда `mount` сообщает системе, что блочное устройство или удаленный ресурс доступны для пользователей в точке монтирования, которая к моменту монтирования уже должна существовать. В этом случае точка монтирования становится корневым каталогом вновь смонтированного устройства или ресурса.

Таким образом, команда `mount` монтирует физическую файловую систему или ресурс к общей логической файловой системе.

Данная команда может иметь следующий синтаксис:

```
mount [-v | -p]
mount [-F ФС] [-V] [-o опции]{устройство|точка_монтирования}
mount [-F ФС] [-V] [-o опции]{устройство точка_монтирования}
```

Команда `mount` при указанных во время вызова аргументах проверяет их, за исключением устройства *`устройство`*, и вызывает соответствующий модуль монтирования для указанного типа файловой системы. Вызванная без аргументов команда `mount` отображает список всех смонтированных файловых систем, считывая информацию из соответствующей таблицы.

Если указан неполный список аргументов (например, только устройства или точки монтирования, или указаны оба эти аргумента, но не задан тип файловой системы), команда `mount` просматривает таблицу стандартных файловых систем в поисках недостающих аргументов, после чего вызывает модуль монтирования для соответствующего типа файловой системы.

Обратная процедура по отношению к монтированию называется *демонтированием* и выполняется командой `umount` со следующим синтаксисом:

```
umount [-V] [-o опции]{устройство|точка_монтирования}
```

В большинстве систем занятую файловую систему демонтировать невозможно. В ней не должно быть открытых файлов и выполняющихся процессов. Если демонтируемая файловая система содержит исполняемые программы, они не должны быть запущены. Для большинства типов файловых систем нет специфического модуля демонтирования. Если такой модуль существует, он выполняется, иначе файловая система демонтируется стандартным модулем.

Команды `mount` и `umount` могут работать со следующими основными опциями:

- `-v` — дополнительно отображаются тип файловой системы и флаги. Поля *`точка_монтирования`* и *`устройство`* переставлены;
- `-p` — отображает список смонтированных файловых систем в формате таблицы смонтированных файловых систем;
- `-F` — задает тип файловой системы для монтирования. Тип файловой системы либо должен быть задан, либо определяется по таблице стандартных файловых систем в ходе монтирования;
- `-V` — отображает результирующую командную строку, но не выполняет команду. Командная строка генерируется с помощью опций и аргументов, указанных пользователем, путем добавления к ним, при необходимости, информации, взятой из таблицы стандартных файловых систем;
- `-o` — задает специфические опции для указанного типа физической файловой системы.

Любой пользователь может вызывать команду `mount` для получения списка смонтированных файловых систем и ресурсов. Например:

```
# mount
/dev/hda8 on /              type ext3     (rw)
none      on /proc          type proc     (rw)
usbdevfs  on /proc/bus/usb type usbdevfs (rw)
```

```
/dev/hda7 on /boot         type ext3     (rw)
none      on /dev/pts      type devpts   (rw,gid=5,mode=620)
none      on /dev/shm      type tmpfs    (rw)
```

Монтирование и демонтирование файловых систем разрешено только суперпользователю root.

Команда `mount` добавляет запись в таблицу смонтированных файловых систем, а `umount` удаляет запись из этой таблицы. Поля в таблице смонтированных устройств разделены пробелами и предоставляют информацию о:

- типе блочного специального устройства;
- точке монтирования;
- типе смонтированной файловой системы;
- опциях монтирования;
- времени монтирования файловой системы.

Для корректного выполнения операций монтирования/демонтирования необходима дополнительная информация, которая находится в таблице стандартных файловых систем (файле /etc/vfstab или /etc/fstab, в зависимости от реализации UNIX). В соответствующих полях этой таблицы, разделенных пробелами или символами табуляции, описаны стандартные параметры для физических файловых систем:

- специальное блочное устройство или имя монтируемого ресурса;
- неформатированное (специальное символьное) устройство для проверки утилитой `fsck`;
- стандартный каталог монтирования;
- тип файловой системы;
- числовой параметр, используемый командой `fsck` для принятия решения об автоматической проверке файловой системы и о порядке этой проверки по отношению к другим файловым системам;

- признак автоматического монтирования файловой системы;
- опции монтирования.

Далее показано содержимое таблицы fstab в операционной системе Linux:

```
# cat fstab
LABEL=/     /           ext3        defaults              1 1
LABEL=/boot /boot       ext3        defaults              1 2
none        /dev/pts    devpts      gid=5,mode=620        0 0
none        /proc       proc        defaults              0 0
none        /dev/shm    tmpfs       defaults              0 0
/dev/hda9   swap        swap        defaults              0 0
/dev/cdrom  /mnt/cdrom  udf,iso9660 noauto,owner,kudzu,ro 0 0
/dev/fd0    /mnt/floppy auto        noauto,owner,kudzu    0 0
```

В следующем примере создается точка монтирования /mnt/data, к которой монтируется созданная файловая система:

```
# mkdir /mnt/data
# mount /dev/sdb1 /mnt/data
```

Запись о смонтированной файловой системе добавляется в файл /etc/mtab, часть которой показана далее:

```
# cat /etc/mtab
/dev/scsi/host0/bus0/target0/lun0/part5 / ext3 rw 0 0

rw,dev=/dev/scsi/host0/bus0/target2/lun0/part1,fs=ext2:vfat,--,umask=0,iocharset=iso8859-1,codepage=850 0 0
                                . . .
/dev/scsi/host0/bus0/target2/lun0/part1 /mnt/data ext2 rw 0 0
```

Создание файловых систем в операционных системах FreeBSD во многом напоминает аналогичный процесс для Linux, хотя есть определенные отличия.

Для создания файловых систем в FreeBSD используются утилиты `fdisk` и `newfs`. Параметры утилиты `fdisk` и механизм ее использования отличаются от принятых в Linux. Лучше всего ме-

ханизм создания файловой системы показать на примере. Как и для операционной системы Linux, я буду использовать конкретные физические устройства накопителей на SCSI-дисках. Для дисковых накопителей IDE-типа имена файлов устройств, естественно, будут отличаться.

Итак, у нас имеется SCSI-диск с номером (ID), равным 2. На разделе 1 диска размещена файловая система операционной системы FreeBSD версии 5.2.1, которую легко просмотреть с помощью команды `mount`:

```
# mount
/dev/da2s1a on / (ufs, local)
devfs on /dev (devfs, local)
/dev/da2s1e on /tmp (ufs, local, soft-updates)
/dev/da2s1f on /usr (ufs, local, soft-updates)
/dev/da2s1d on /var (ufs, local, soft-updates)
```

В операционной системе FreeBSD файлы устройств дисковых накопителей SCSI именуются как `/dev/da`*n*, где *n* — идентификатор накопителя на SCSI-шине.

Нужно создать еще один основной раздел диска, на котором бы размещалась небольшая файловая система размером 50—60 Мбайт, имеющая тип ufs. Для этого создадим раздел 2 на диске (раздел 1 уже имеется).

Раздел создается утилитой `fdisk` со следующими опциями:

```
fdisk -f config_file
```

Здесь config_file содержит информацию о создаваемой файловой системе. Для создания записей такого файла необходимо знать текущую разбивку диска и размещение уже существующих разделов, чтобы не затереть важные данные. Информацию о разделах диска в FreeBSD можно получить, задав утилиту `fdisk` с параметром, указывающим на файл соответствующего устройства или без параметров (тогда отобразятся данные на рабочей системе):

В нашем случае для устройства /dev/da2 получим такую информацию:

```
# fdisk
******* Working on device /dev/da2 *******
parameters extracted from in-core disklabel are:
cylinders=530 heads=255 sectors/track=63 (16065 blks/cyl)

parameters to be used for BIOS calculations are:
cylinders=530 heads=255 sectors/track=63 (16065 blks/cyl)

Media sector size is 512
Warning: BIOS sector numbering starts with sector 1
Information from DOS bootblock is:
The data for partition 1 is:
sysid 165 (0xa5),(FreeBSD/NetBSD/386BSD)
    start 63, size 7984242 (3898 Meg), flag 80 (active)
        beg: cyl 0/ head 1/ sector 1;
        end: cyl 496/ head 254/ sector 63
The data for partition 2 is:
 <UNUSED>
The data for partition 3 is:
<UNUSED>
The data for partition 4 is:
<UNUSED>
```

Исходя из этой информации, можно сделать следующие выводы:

- на данном диске уже есть раздел 1, поэтому следующий раздел, который можно создать, — это раздел 2 (ему соответствует специальный файл /dev/da2s2);
- раздел 2 можно создавать, например, начиная с сектора 8 000 000;

- для обеспечения необходимого размера дискового пространства (50—60 Мбайт) в разделе 2 можно выбрать количество секторов, равное 120 000 (размер сектора равен 512 байтов).

Таким образом, можно сформировать следующие записи для файла config_file:

```
g c530 h255 s63
p 2 165 8000000 120000
```

Первая строка содержит геометрию диска, а вторая, начинающаяся с литеры p, собственно параметры создаваемого раздела:

- первый параметр (равный 2) указывает на номер создаваемого раздела;
- второй параметр (165) указывает на тип файловой системы;
- третий параметр (8000000) задает начальный сектор раздела;
- четвертый параметр (120000) указывает на размер раздела.

Сохраним записи в файле config_slice и выполним команду:

```
# fdisk -f config_slice
******* Working on device /dev/da2 *******
fdisk: WARNING: adjusting start offset of partition 2
    from 8000000 to 8000055, to fall on a head boundary
fdisk: WARNING: adjusting size of partition 2 from 120000 to
112770
    to end on a cylinder boundary
```

Из результата выполнения видно, что раздел 2 создан успешно. Следует учитывать, что программа `fdisk` может незначительно изменять номер начального сектора (смещение раздела) для выравнивания головок чтения/записи, а также выравнивать конечный сектор по границе, что видно из результата. При создании разделов диска нужно учитывать эти нюансы, чтобы получить требуемый окончательный размер дискового пространства.

После создания раздела полезно проверить его параметры:

```
# fdisk
******* Working on device /dev/da2 *******
parameters extracted from in-core disklabel are:
cylinders=530 heads=255 sectors/track=63 (16065 blks/cyl)

parameters to be used for BIOS calculations are:
cylinders=530 heads=255 sectors/track=63 (16065 blks/cyl)

Media sector size is 512
Warning: BIOS sector numbering starts with sector 1
Information from DOS bootblock is:
The data for partition 1 is:
sysid 165 (0xa5),(FreeBSD/NetBSD/386BSD)
    start 63, size 7984242 (3898 Meg), flag 80 (active)
        beg: cyl 0/ head 1/ sector 1;
        end: cyl 496/ head 254/ sector 63
The data for partition 2 is:
sysid 165 (0xa5),(FreeBSD/NetBSD/386BSD)
    start 8000055, size 112770 (55 Meg), flag 0
        beg: cyl 497/ head 250/ sector 1;
        end: cyl 504/ head 254/ sector 63
The data for partition 3 is:
<UNUSED>
The data for partition 4 is:
<UNUSED>
```

Второй этап — создание файловой системы в разделе /dev/da2s2. Это выполняется при помощи программы `newfs`:

```
# newfs /dev/da2s2
/dev/da2s2: 55.1MB (112768 sectors) block size 16384,
fragment size 2048
```

```
        using 4 cylinder groups of 13.78MB, 882 blks, 1792
inodes.
super-block backups (for fsck -b #) at:
 160, 28384, 56608, 84832
```

Вот, собственно, и все. Далее можно смонтировать вновь созданную файловую систему и работать с ней:

```
# mkdir /data
# mount /dev/da2s2 /data
```

Здесь при помощи команды `mkdir` создается каталог /data, который будет точкой монтирования файловой системы на /dev/da2s2, а затем файловая система монтируется к этому каталогу с помощью команды `mount`.

Можно проверить смонтированные файловые системы командами `mount` и `df`:

```
# mount
/dev/da2s1a on / (ufs, local)
devfs on /dev (devfs, local)
/dev/da2s1e on /tmp (ufs, local, soft-updates)
/dev/da2s1f on /usr (ufs, local, soft-updates)
/dev/da2s1d on /var (ufs, local, soft-updates)
/dev/da2s2 on /data (ufs, local)
# df -k
Filesystem  1K-blocks   Used   Avail Capacity  Mounted on
/dev/da2s1a    253678  37542  195842    16%    /
devfs               1      1       0   100%    /dev
/dev/da2s1e    253678    628  232756     0%    /tmp
/dev/da2s1f   2111806 899556 1043306    46%    /usr
/dev/da2s1d    253678   6082  227302     3%    /var
/dev/da2s2      54382      4   50028     0%    /data
```

Хочу добавить, что программу `fdisk` можно задавать и без файла конфигурации, при этом следует указать, что данные будут поступать со стандартного устройства ввода:

```
fdisk -f -
```

Здесь дефис после опции `-f` указывает на прием данных со стандартного ввода (клавиатуры, например).

Дополнительная информация по командам `fdisk`, `mkfs`, `newfs` и `mount` находится в man-страницах операционной системы Linux, FreeBSD и других операционных систем. Кроме того, для конкретных версий операционных систем параметры данных команд могут изменяться, поэтому желательно ознакомиться с работой этих программ именно в тех версиях, с которыми вы работаете.

5.2.3. Диагностика файловых систем

Файловые системы следует проверять время от времени на целостность, для чего в операционной системе UNIX имеются встроенные программные средства, наиболее эффективным из которых является утилита `fsck`.

Программа `fsck` включена во все версии операционных систем, и при подозрении на наличие ошибок всегда следует ее использовать. Утилита позволяет найти и исправить повреждения (inconsistency) в файловых системах, причем допускает два режима работы — автоматический или интерактивный.

При значительных повреждениях файловой системы программу `fsck` лучше запускать в интерактивном режиме — при этом пользователь должен подтверждать каждый свой шаг, что вынуждает задумываться, прежде чем что-то сделать. Имейте в виду, что некоторые исправления могут привести к определенным потерям данных, что отображается в диагностических сообщениях.

В интерактивном режиме перед каждым исправлением программа `fsck` ожидает ответа от пользователя — "Да" (`YES`) или "Нет" (`NO`). Если при запуске утилиты указать параметр `-y`, `fsck` допускает ответ "Да" и не делает паузы для ответа. Замечу, что программа `fsck` — многопроходная команда контроля файловых систем, поэтому на каждом проходе файловой системы выполняются различные этапы: проверка блоков и размеров файлов, полных имен файлов, связности, карты свободных блоков (возможно, с ее перестройкой), подсчет ссылок и т. д.

Перед запуском утилиты `fsck` файловая система должна быть демонтирована. Кроме того, если исправления вносятся в критическую файловую систему, например, в корневую, то после `fsck` систему необходимо перезагрузить.

5.3. Особенности установки различных операционных систем

Рассмотрим некоторые практические аспекты установки различных версий операционной системы UNIX. Предполагаем, что установка UNIX выполняется для платформы x86.

Начнем с FreeBSD. Эта операционная система поставляется с простой в использовании текстовой программой установки `sysinstall`, которая является основной для данной версии.

Перед установкой FreeBSD желательно собрать информацию об аппаратной части компьютера. Во время установки FreeBSD сообщит сведения об устройствах (жестких дисках, сетевых картах, CD-ROM и т. д.) с номером модели и производителем. FreeBSD также попытается определить правильную конфигурацию для этих устройств, включая информацию об используемых прерываниях (IRQ) и портах ввода/вывода. Из-за возможных проблем с оборудованием этот процесс не всегда завершается успешно, и, возможно, придется корректировать определенную FreeBSD-конфигурацию.

Если на компьютере уже есть установленная операционная система, например, Windows или Linux, то можно просмотреть настройки оборудования.

Если компьютер, на который вы устанавливаете FreeBSD, содержит важные данные, убедитесь в наличии резервных копий и проверьте их сохранность перед установкой FreeBSD. Во время установки ОС запросит подтверждение перед тем, как записать данные на диск, но если процесс запущен, изменения нельзя отменить.

Минимальное дисковое пространство, требуемое для установки FreeBSD, должно быть не менее 100 Мбайт. Хочу заметить, что при этом не останется места для личных файлов пользователя. Более реальный объем — 250 Мбайт без графической оболочки и более 350 Мбайт с графической оболочкой. Если вы собираетесь устанавливать большое количество дополнительного ПО, вам понадобится больший объем дискового пространства.

При наличии установочного компакт-диска FreeBSD можно выполнить инсталляцию операционной системы без какой-либо специальной подготовки — это возможно, если компьютер поддерживает загрузку с CD-ROM или DVD (обычно этот пункт в BIOS называется "Boot Order").

В процессе установки программа-инсталлятор не будет производить никаких изменений на дисках, пока не выдаст следующее сообщение:

```
Last Chance: Are you SURE you want continue the installation?


If you're running this on a disk with data you wish to save
then WE
STRONGLY ENCOURAGE YOU TO MAKE PROPER BACKUPS before
proceeding!


We can take no responsibility for lost disk contents!
```

До этого предупреждения установка может быть прервана в любой момент, поскольку никаких изменений на жестком диске еще не сделано. Если вы считаете, будто что-то настроили неправильно, можете просто выключить компьютер без риска что-либо повредить.

В процессе инсталляции FreeBSD на каком-то этапе потребуется разбивка дискового пространства для создания файловой системы и создание самой файловой системы. Эти процедура выполняются программой `sysinstall`, которая в свою очередь вызывает программы `fdisk` и `newfs`, рассмотренные нами ранее.

Следует учитывать, что FreeBSD не использует BIOS и не получает никакой информации о логическом отображении дисков в BIOS, что может вызвать определенные проблемы в ситуациях, когда диски имеют одинаковую геометрию и содержат точную копию данных друг друга.

При использовании FreeBSD всегда нужно восстанавливать настройки BIOS к первоначальной нумерации дисков перед установкой системы и оставлять их в таком виде. Если нужно переключить диски, то лучший вариант — физическое переконфигурирование путем переключения перемычек и кабелей в интерфейсе.

После разбиения дискового пространства программа-инсталлятор предлагает установить менеджер загрузки (Boot Manager).

Как правило, установка менеджера загрузки требуется в следующих случаях:

- в системе имеется больше одного диска, и установка FreeBSD выполняется не на первый диск;
- существует уже другая установленная операционная система, и требуется установить FreeBSD вместе с ней на один и тот же диск так, чтобы можно было выбирать вариант загрузки.

Программа инсталляции предлагает несколько вариантов выбора менеджера загрузки. Если FreeBSD — единственная операционная система, установленная на вашем компьютере, и находится на первом жестком диске, можно выбрать менеджер загрузки **Standard**. Если будет использоваться менеджер загрузки сторонних разработчиков, следует выбрать опцию **None**.

Если установка операционной системы выполняется впервые, то не нужно акцентировать внимание на принципах разбиения диска. Более важно установить FreeBSD и начать с ней работать, тем более что в любой момент можно переустановить систему для изменения схемы разделов.

В табл. 5.1 показаны четыре раздела: один для подкачки и три для файловых систем.

Таблица 5.1. *Планирование разделов для первого диска*

Раздел	Файловая система	Размер	Описание
A	/	100 Мбайт	Корневая файловая система. Любая другая файловая система будет смонтирована к корневой. 100 Мбайт — это оптимальный размер для этой файловой системы. Здесь не будет храниться слишком много данных. Обычная установка FreeBSD разместит около 40 Мбайт данных. Оставшееся пространство используется для временных файлов, а также оставляет возможность расширения для будущих версий FreeBSD, которым может понадобиться больше места в данной файловой системе
B	N/A	2×RAM или 3×RAM	Если у вас больше одного диска, можно расположить область подкачки (swapping) на каждом диске. FreeBSD будет использовать каждый диск, что серьезно увеличит скорость подкачки. В этом случае определите общий размер области подкачки (например, 128 Мбайт) и разделите его на число имеющихся дисков (например, два) для определения размера разделов подкачки, которые нужно разместить на каждом диске (в этом примере 64 Мбайт на диск)
E	/var	50 Мбайт	Каталог /var содержит преимущественно временные файлы. Эти файлы создаются ОС и ее утилитами. Например, здесь хранятся почтовые сообщения до тех пор, пока не будут прочитаны пользователями. Многие из этих файлов интенсивно читаются и записываются в процессе ежедневной работы FreeBSD. Размещение их в отдельной файловой системе позволяет FreeBSD оптимизировать доступ к этим файлам без привлечения других каталогов, не имеющих такой же модели доступа.

Таблица 5.1 *(окончание)*

Раздел	Файловая система	Размер	Описание
			Обычно размер раздела не превышает 1 Гбайт, хотя некоторые системы, например, мощные почтовые серверы, могут потребовать значительно большего размера
F	/usr	Остальная часть диска	Этот каталог обычно содержит программные файлы. В стандартной конфигурации это самый большой раздел. Желательно выделить под этот раздел не менее 1 Гбайт дискового пространства. Минимальный размер обычно принимается равным 100 Мбайт

После разбивки дисков и создания файловых систем программа установки начинает инсталляцию программного обеспечения FreeBSD.

Вначале нужно определиться с дистрибутивным набором для установки — его выбор зависит в основном от целевого назначения и доступного дискового пространства, при этом опции варьируются от наименьшей (minimal) конфигурации до полной (All) установки. Для начинающих пользователей FreeBSD лучшим выбором будет одна из этих предустановленных опций, а выборочную (custom) настройку могут задать более опытные пользователи.

Если нужен графический интерфейс пользователя, то следует выбрать дистрибутив, содержащий один из графических пакетов (обычно это XFree86 версий 4.6 и выше).

После выбора подходящего дистрибутива можно будет указать установку коллекции портов FreeBSD — набор файлов, который автоматизирует загрузку, компилирование и установку пакетов программного обеспечения сторонних разработчиков. Програм-

ма установки не проверяет, достаточно ли места на диске, так что следует это иметь в виду, поскольку в FreeBSD 5.2.1, например, коллекция портов занимает около 300 Мбайт.

Перед установкой коллекции портов появляется приглашение:

```
                    User Confirmation Requested

 Would you like to install the FreeBSD ports collection?


 This will give you ready access to over 10,000 ported
software packages,

 at a cost of around 300 MB of disk space when "clean" and
possibly much

 more than that if a lot of the distribution tarballs are
loaded

 (unless you have the extra CDs from a FreeBSD CD/DVD
distribution

 available and can mount it on /cdrom, in which case this is
far less

 of a problem).


 The ports collection is a very valuable resource and well
worth having

 on your /usr partition, so it is advisable to say Yes to
this option.


 For more information on the ports collection & the latest
ports,

 visit:

     http://www.FreeBSD.org/ports


                        [ Yes ]     No
```

Нужно выбрать вариант **Yes** для установки коллекции портов, или вариант **No**, чтобы пропустить установку.

Время установки зависит от выбранного дистрибутивного набора, источника установки и производительности компьютера.

В процессе установки могут появляться сообщения о статусе процесса. Установка считается завершенной, когда будет выведено следующее сообщение:

```
                        Message

Congratulations! You now have FreeBSD installed on your
system.

We will now move on to the final configuration questions.
For any option you do not wish to configure, simply select
No.

If you wish to re-enter this utility after the system is up,
you may
do so by typing: /stand/sysinstall .

                        [ OK ]

                [  Press enter to continue  ]
```

После успешной установки необходимо настроить множество параметров. Некоторые из них могут быть заданы из меню параметров после инсталляции перед загрузкой установленной FreeBSD или позднее с использованием программы /stand/sysinstall, в которой нужно выбрать пункт **Configure**.

Для настройки сетевых устройств (Network Device Configuration) нужно выбрать опцию **Yes** в ответ на приглашение:

```
                        User Confirmation Requested
   Would you like to configure any Ethernet or SLIP/PPP
   network devices?

                        [ Yes ]   No
```

Надо указать интерфейс для настройки с помощью клавиш навигации и нажать клавишу <Enter>. Появится подтверждение:

```
                User Confirmation Requested

    Do you want to try IPv6 configuration of the
    interface?


                          Yes   [ No ]
```

Для частной локальной сети обычного протокола IPv4 вполне достаточно, поэтому можно выбрать опцию **No**. Далее программа конфигурации запросит подтверждение на конфигурирование DHCP:

```
    Do you want to try DHCP configuration of the
    interface?


                          Yes   [ No ]
```

Если DHCP (Dynamic Host Configuration Protocol, протокол динамического конфигурирования компьютера) не нужен, выберите вариант **No**. Выбор опции **Yes** запустит утилиту `dhclient`, и если все пройдет нормально, заполнит информацию о конфигурации сети автоматически.

Следующий экран конфигурации сети показывает настройку устройства Ethernet для системы, которая будет работать шлюзом для локальной сети. Назначение полей:

- **Host** — полное имя хоста;
- **Domain** — имя домена, в котором находится ваш компьютер;
- **IPv4 Gateway** — IP-адрес хоста, пересылающего пакеты из локальной сети. Необходимо заполнить его, если компьютер подключен к сети. Оставьте это поле пустым, если компьютер является шлюзом в Интернет для сети. Шлюз IPv4 известен так же как шлюз по умолчанию или маршрут по умолчанию;
- **Name server** — IP-адрес местного сервера DNS;

- **IPv4 address** — IP-адрес, использованный для этого интерфейса;
- **Netmask** — сетевая маска локальной сети;
- дополнительные параметры для программы `ifconfig` — любые специфичные для интерфейса опции `ifconfig`, которые вы хотите добавить.

После завершения настроек нужно выбрать вариант **Yes** и активизировать сетевой интерфейс:

```
                    User Confirmation Requested

      Would you like to Bring Up the ed0 interface
      right now?


                              [ Yes ]   No
```

Теперь сетевые настройки станут активными, однако компьютер нужно перезагрузить.

Следующий этап конфигурирования сети — настройка шлюза. Для этого нужно подтвердить выбор:

```
                     User Confirmation Requested

     Do you want this machine to function as a network
     gateway?


                               [ Yes ]    No
```

Если настройка шлюза не требуется, нужно отметить вариант **No**.

Настройка сервисов Интернета (Configure Internet Services) выполняется, если получено следующее приглашение к настройке:

```
                     User Confirmation Requested

Do you want to configure inetd and the network services that
it provides?


                               Yes   [ No ]
```

Если выбран ответ **No**, различные сервисы, такие как `telnetd`, не будут работать, а это означает, что удаленные пользователи не смогут зайти через telnet на этот компьютер.

Эти сервисы могут быть включены после установки путем редактирования файла /etc/inetd.conf. Выберите вариант **Yes**, если хотите настроить данные сервисы во время установки. Появится дополнительный запрос подтверждения:

```
                    User Confirmation Requested

The Internet Super Server (inetd) allows a number of simple
Internet services to be enabled, including finger, ftp and
telnetd.  Enabling these services may increase risk of
security problems by increasing

the exposure of your system.


With this in mind, do you wish to enable inetd?


                                [ Yes ]   No
```

Выберите опцию **Yes** для продолжения настройки:

```
                    User Confirmation Requested

inetd(8) relies on its configuration file, /etc/inetd.conf,
to determine which of its Internet services will be
available.  The default FreeBSD inetd.conf(5) leaves all ser-
vices disabled by default, so they must be specifically en-
abled in the configuration file before they will function,
even once inetd(8) is enabled.  Note that services for IPv6
must be separately enabled from IPv4 services.


Select [Yes] now to invoke an editor on /etc/inetd.conf,
or [No] to use the current settings.


                                [ Yes ]   No
```

Выбор ответа **Yes** позволит добавить сервисы путем удаления символа # перед началом строки.

После добавления нужных сервисов нажатие клавиши <Esc> отобразит меню, позволяющее выйти с сохранением изменений.

Следующий пункт настройки — настройка доступа анонимного пользователя по FTP-протоколу к этому компьютеру. Приглашение выглядит так:

```
                User Confirmation Requested

 Do you want to have anonymous FTP access to this machine?


                              Yes    [ No ]
```

Выбор ответа по умолчанию **No** все же позволит пользователям, имеющим учетные записи с паролями, использовать протокол FTP для доступа к компьютеру.

Доступ к компьютеру сможет получить кто угодно, если разрешить анонимные соединения по FTP, поэтому следует оценить возможные проблемы с безопасностью. Чтобы разрешить анонимный доступ по FTP, следует выбрать опцию **Yes**. После подтверждения появится следующее сообщение:

```
This screen allows you to configure the anonymous FTP user.

The following configuration values are editable:

UID:     The user ID you wish to assign to the anonymous FTP
user.
         All files uploaded will be owned by this ID.

Group:   Which group you wish the anonymous FTP user
to be in.
```

```
Comment: String describing this user in /etc/passwd

FTP Root Directory:

        Where files available for anonymous FTP will be kept.

Upload subdirectory:

        Where files uploaded by anonymous FTP users will go.
```

Корневой каталог ftp по умолчанию будет размещен в /var. Если в нем не хватает места для ftp, можно использовать каталог /usr, выбрав в качестве корневого каталог /usr/ftp.

Когда будут выбраны подходящие значения, нажмите клавишу <Enter>, чтобы продолжить.

```
                        User Confirmation Requested

        Create a welcome message file for anonymous FTP users?

                                [ Yes ]    No
```

При выборе опции **Yes** можно создать (при желании) приглашение для анонимного пользователя, после чего программа конфигурирования предложит настроить NFS.

Сетевая файловая система (Network File System, NFS) позволяет совместно использовать файлы в сети. Компьютер может быть настроен как сервер, клиент или как и то, и другое. Стандартное приглашение к настройке NFS выглядит так:

```
                        User Confirmation Requested

        Do you want to configure this machine as an NFS
        server?

                                Yes    [ No ]
```

Если сервер NFS не нужен, выберите опцию **No**.

Если отмечена опция **Yes**, появится сообщение, информирующее о том, что должен быть создан файл exports:

```
                    Message

  Operating as an NFS server means that you must first
configure an /etc/exports file to indicate which hosts are
allowed certain kinds of access to your local filesystems.

  Press [Enter] now to invoke an editor on /etc/exports

                    [ OK ]
```

Для продолжения следует нажать клавишу <Enter>, тогда запустится текстовый редактор, позволяющий создать и отредактировать файл exports.

Используйте инструкции для добавления экспортируемых файловых систем сейчас или позднее с помощью выбранного вами текстового редактора. Обратите внимание, что имя/расположение файла показаны внизу окна редактора.

NFS-клиент позволяет организовать доступ к серверам NFS. Вот приглашение:

```
                    User Confirmation Requested

        Do you want to configure this machine as an NFS
        client?


                         Yes    [ No ]
```

Выберите требуемую опцию и нажмите клавишу <Enter>.

Еще одним важным элементом настройки является *профиль безопасности* (security profile) — набор параметров настройки, с помощью которого достигается требуемый уровень безопасности и удобства работы за счет разрешения или запрещения запуска определенных программ и другими настройками. Чем бóльшие ограничения указаны в профиле безопасности, тем меньше программ будет запущено по умолчанию — это является одним из основных принципов безопасности: не запускать ничего, кроме того, что необходимо.

Профиль безопасности — это лишь установка по умолчанию, поэтому любая программа может быть запущена или остановлена после инсталляции FreeBSD посредством редактирования или добавления соответствующих строк в файл /etc/rc.conf.

Таблица 5.2 описывает действие каждого профиля безопасности. В столбцах показан выбранный профиль безопасности, а в строках — программы или функции, которые профиль включает или выключает.

Таблица 5.2. *Возможные профили безопасности*

Программа или функция	Extreme (максимальный)	Moderate (умеренный)
sendmail	Нет	Да
sshd	Нет	Да
NFS server	Нет	Да

Инсталляционная программа выдает такое приглашение для настройки профиля безопасности:

```
                    User Confirmation Requested

 Do you want to select a default security profile for this
host (select No for "medium" security)?


                                [ Yes ]    No
```

Выбор опции **No** установит профиль безопасности к среднему значению, выбор варианта **Yes** позволит назначить другой профиль безопасности. После этого будет отображен запрос на подтверждение, соответствующий отмеченным настройкам безопасности:

```
                          Message


Moderate security settings have been selected.
```

```
Sendmail and SSHd have been enabled, securelevels are
disabled, and NFS server setting have been left intact.
PLEASE NOTE that this still does not save you from having
to properly secure your system in other ways or exercise
due diligence in your administration, this simply picks
a standard set of out-of-box defaults to start with.

To change any of these settings later, edit /etc/rc.conf

                        [OK]
                      Message

Extreme security settings have been selected.

Sendmail, SSHd, and NFS services have been disabled, and
securelevels have been enabled.
PLEASE NOTE that this still does not save you from having
to properly secure your system in other ways or exercise
due diligence in your administration, this simply picks
a more secure set of out-of-box defaults to start with.

To change any of these settings later, edit /etc/rc.conf

                        [OK]
```

Профиль безопасности еще не гарантирует высокой безопасности системы — даже если выбраны экстремальные установки, понадобится поддерживать безопасность, анализируя соответствующие списки рассылки, используя эффективные пароли и т. д. Профиль всего лишь повышает желаемый уровень безопасности вновь установленной системы до приемлемого значения.

Для использования графического интерфейса пользователя, например, KDE, GNOME или другого, нужно настроить X-сервер.

Используя программу /stand/sysinstall, выберите опцию **Configure**, а затем — **XFree86**. Следует иметь в виду, что неправильная настройка X-сервера может привести к "зависанию" компьютера, поэтому настройку X-сервера рекомендуют производить после завершения установки.

Здесь очень важно правильно настроить монитор и видеокарту — перед конфигурированием этих устройств нужно знать их параметры, поскольку указание неправильных параметров может привести к повреждению оборудования.

Если вы не располагаете этой информацией, выберите вариант **No** и выполните конфигурирование после установки, когда необходимая информация будет собрана.

Есть несколько вариантов настройки X-сервера. Используйте клавиши навигации для выбора одного из них и подтвердите выбор клавишей <Enter>.

Имейте в виду, что выполнение команд `xf86cfg` и `xf86cfg -textmode` может сделать экран темным в течение нескольких секунд.

Выбранные настройки нужно сохранить. Убедитесь, что выбран путь /etc/XF86Config для сохранения настроек.

```
I am going to write the XF86Config file now. Make sure you
don't accidently overwrite a previously configured one.


Shall I write it to /etc/X11/XF86Config? y
```

Если настройка была прервана на каком-либо этапе, вы можете запустить ее вновь, выбрав вариант **Yes** при появлении следующего сообщения:

```
          User Confirmation Requested

The XFree86 configuration process seems to have

failed.  Would you like to try again?


               [ Yes ]    No
```

Если в процессе настройки XFree86 возникли проблемы, выберите опцию **No**, нажмите клавишу <Enter> и продолжайте процесс установки. После завершения всех настроек можно использовать команду `xf86cfg -textmode` или `xf86config` для получения доступа к утилитам настройки в режиме командной строки под учетной записью пользователя root.

Для перезагрузки сервера по умолчанию используется комбинация клавиш <Ctrl>+<Alt>+<Backspace>. Такую возможность разрешено использовать, если какие-то настройки сервера выполнены неправильно. Это помогает предотвратить повреждение оборудования.

Настройкой по умолчанию, разрешающей переключать видеорежимы во время работы X-сервера, является комбинация клавиш <Ctrl>+<Alt>+<+> или <Ctrl>+<Alt>+<->.

После установки высота, ширина или центровка экрана может быть выбрана с использованием программы `xvidtune`, если вы запустите XFree86 с `xvidtune`.

Если настройка XFree86 проведена успешно, будет предложен выбор оконного менеджера. Для FreeBSD доступны самые разные оконные менеджеры. Их функциональность варьируется в широких пределах.

Некоторые оконные менеджеры довольствуются минимумом места на диске и небольшим объемом памяти, другие, с большим набором функций, требуют гораздо больше ресурсов. Лучший путь определить наиболее подходящий менеджер — попробовать несколько из предложенных программой настройки XFree86. Они доступны из коллекции портов и в виде пакетов и могут быть добавлены после установки.

В процессе установки нужно добавить хотя бы одного пользователя, чтобы работать в системе без входа под учетной записью пользователя root. Корневой каталог обычно имеет небольшой размер, и запуск приложений с использованием учетной записи root быстро заполнит его.

Далее показано сообщение в процессе настройки учетной записи пользователя:

```
                    User Confirmation Requested

 Would you like to add any initial user accounts to the
system? Adding at least one account for yourself at this
stage is suggested since working as the "root" user is
dangerous (it is easy to do things which

 adversely affect the entire system).


                              [ Yes ]   No
```

Отметьте вариант **Yes** и подтвердите выбор, чтобы продолжить добавление пользователя.

При выборе полей будут появляться опции меню, позволяющие ввести необходимую информацию:

- **login ID** — имя нового пользователя (обязательно);
- **UID** — числовой идентификатор для этого пользователя (оставьте пустым для автоматического выбора);
- **group** — имя группы этого пользователя (оставьте пустым для автоматического выбора);
- **password** — пароль этого пользователя (заполняйте данное поле внимательно);
- **full name** — полное имя пользователя (комментарий);
- **member groups** — группы, к которым принадлежит пользователь (т. е. имеет права доступа);
- **home directory** — домашний каталог пользователя (оставьте пустым для выбора по умолчанию);
- **login shell** — оболочка пользователя, запускаемая при входе в систему (оставьте пустым для оболочки по умолчанию, например, /bin/sh).

Пользователь добавляется в группу wheel, чтобы иметь возможность стать суперпользователем с привилегиями root.

Далее следует установить пароль для пользователя root. Появляется приглашение:

```
                        Message
 Now you must set the system manager's password.
 This is the password you'll use to log in as "root".

                        [ OK ]

                [ Press enter to continue ]
```

Нажмите клавишу <Enter> для установки пароля пользователя root. После подтверждения необходимо два раза правильно ввести пароль:

```
Changing local password for root.
New password :
Retype new password :
```

Напомню, что должна быть предусмотрена возможность восстановления пароля, если вы его забудете. Установка продолжится после успешного ввода пароля.

Мы рассмотрели практически все базовые настройки операционной системы FreeBSD. Перед выходом из программы конфигурирования система предложит выполнить дополнительные настройки, если вы их не сделали. Это можно сделать сейчас или после установки с помощью программы /stand/sysinstall:

```
            User Confirmation Requested
 Visit the general configuration menu for a chance to set any
last options?

                Yes   [ No ]
```

Выберите опцию **Exit Install** с помощью клавиш навигации и подтвердите выбор, нажав клавишу <Enter>. Появится вопрос о подтверждении выхода из программы установки:

```
                    User Confirmation Requested

 Are you sure you wish to exit? The system will reboot (be
sure to

 remove any floppies from the drives).


                              [ Yes ]   No
```

Выберите вариант **Yes** и извлеките дискету из дисковода, если загрузка была произведена с нее. CD-ROM будет заблокирован до тех пор, пока компьютер не начнет перегружаться. После начала перезагрузки компакт-диск нужно будет извлечь из привода. В процессе загрузки системы внимательно наблюдайте за сообщениями об ошибках.

На этом анализ процесса инсталляции операционной системы FreeBSD можно закончить.

5.3.1. Установка операционной системы Solaris

Для запуска инсталляции с графическим интерфейсом пользователя в системе должно быть установлено, по крайней мере, 32 Мбайт оперативной памяти — в современных системах оперативной памяти намного больше, поэтому требования по памяти выполняются практически всегда.

Еще один важный момент касается объема жесткого диска — он должен быть достаточно большим, чтобы можно было установить базовое программное обеспечение операционной системы, области свопинга и дополнительное программное обеспечение.

Перед установкой операционной системы Solaris необходимо знать и другие параметры, например:

- аппаратный адрес Ethernet-адаптера;
- IP-адрес хоста;

- характеристики видеоадаптера и монитора;
- максимальную частоту кадровой развертки видеокарты;
- тип мыши.

Желательно приблизительно оценить разбивку раздела диска. Следует учитывать, что 100 Мбайт дискового пространства нужно выделить для swap-области. Один из возможных вариантов разбивки показан далее:

```
/            =  64 MB (лучше делать небольшим)
/var         = 200 MB (log-файлы, spool, e-mail, updates)
/opt         = 700 MB
/usr         = 700 MB (системные файлы)
swap         = 100 MB
/export/home = остающееся дисковое пространство
```

Инсталляцию системы можно выполнить или в интерактивном режиме, или в режиме Web Start. Здесь мы рассмотрим только интерактивный метод. Для начала инсталляции нужно вставить компакт-диск и перезагрузить компьютер. После приглашения загрузчика OpenBoot нужно выполнить команду:

```
boot cdrom
```

Система осуществит загрузку с компакт-диска и начнется конфигурирование устройств. После этого программа инсталляции в режиме диалога будет задавать различные вопросы о конфигурации системы.

После вывода раздела с системной информацией на экране появится следующее сообщение:

```
Solaris Interactive Installation:
This system is upgradable, so there are two ways to install
the Solaris software.
The Upgrade option updates the Solaris software to the new
release, saving as many modifications to the previous version
of Solaris software as possible. Back up the system before
using the Upgrade option.
```

```
The Initial option overwrites the system disks with the new
version of Solaris software. This option allows you to
preserve any existing file systems. Back up any modifications
made to the previous version of Solaris software before
starting the Initial option.
After you select an option and complete the tasks that
follow, a summary of your actions will be displayed.
F2_Upgrade F4_Initial F5_Exit F6_Help
```

При инсталляции вся информация в разделах (если они уже существуют) /, /usr, /opt, /var будет уничтожена, поэтому при необходимости нужно выполнить резервное копирование данных перед началом установки.

Для полной переинсталляции системы нажмите клавишу <F4>. Затем на экране появится диалог:

```
You'll be using the initial option for installing Solaris
software on the system. The initial option overwrites the
system disks when the new Solaris software is installed.
On the following screens, you can accept the defaults or you
can customize how Solaris software will be installed by:
- Selecting the type of Solaris software to install
- Selecting disks to hold software you've selected
- Specifying how file systems are laid out on the
disks
After completing these tasks, a summary of your selections
(called a profile) will be displayed.
```

Для продолжения нужно нажать клавишу <F2>, после чего будет предложено выбрать географический регион и язык. Далее программа инсталляции задаст вопрос о необходимости поддержки 64-разрядных пакетов программного обеспечения.

После нажатия клавиши <F2> программа инсталляции предложит выбрать конфигурацию программного обеспечения для установки. Для инсталляции сервера по возможности лучше

выбрать полный дистрибутив, чтобы не возвращаться потом к установке отдельных пакетов. При установке Solaris можно потребовать от программы установки автоматического планирования файловых систем, хотя можно выполнить планирование и вручную.

5.3.2. Установка Linux

Рассмотрим некоторые вопросы инсталляции еще одной, очень популярной операционной системы — Linux. По операционной системе Linux, ее установке и настройкам к настоящему времени накопился огромный архив документации, поэтому нет особого смысла подробно рассматривать процесс инсталляции. Напомню лишь основные моменты.

Устанавливать Linux можно одним из следующих способов:

- с локального компакт-диска;
- с жесткого диска, на который скопирован дистрибутив Linux;
- с файл-сервера локальной сети по NFS;
- с удаленного компьютера по FTP;
- с другого компьютера в локальной сети.

Рассмотрим наиболее распространенный вариант установки — с компакт-диска.

Как и для любой другой операционной системы, для Linux нужно провести ревизию оборудования, на которое будет установлена операционная система. Поэтому сбор сведений об аппаратуре является важным этапом в подготовке к инсталляции.

Кроме того, если предполагается, что хост Linux будет работать в сети, то желательно знать имя домена, серверы имен, IP-адреса хоста и маршрутизаторов. Операционная система может быть установлена как сервер (опция **Server**), рабочая станция (опция **Workstation**) или с опциями пользовательских настроек (опция **Custom**).

Рассмотрим вариант инсталляции в режиме рабочей станции — наиболее простой из предлагаемых. Кроме того, на примере настройки **Workstation** можно лучше понять основные принципы конфигурирования Linux.

Установка в этом режиме использует все свободное пространство на выбранном жестком диске для создания:

- ❒ swap-раздела;
- ❒ раздела, в котором будет располагаться ядро и некоторые дополнительные файлы (монтируется как /boot);
- ❒ раздела, занимающего оставшееся место на диске, который монтируется как корневая файловая система.

Для установки Red Hat Linux надо создать на диске один или несколько разделов, которые будут после инсталляции иметь тип "Linux native", и один раздел типа "Linux swap".

При принятии решения о том, где расположить Linux, необходимо учесть то, как будет происходить загрузка операционной системы. Если нужно сохранить возможность запуска на компьютере привычной операционной системы Windows 95/98 или Windows 2000/XP (за счет перезагрузки ОС), то придется применить какой-то из многовариантных загрузчиков (multiloaders).

Один из таких загрузчиков — LILO (LInux LOader), входит в состав дистрибутива. Из-за ограничений, накладываемых BIOS большинства Intel-совместимых компьютеров, программный код LILO должен располагаться на одном из двух доступных жестких дисков, причем в пределах первых 1024 цилиндров этого диска.

Все данные, которые необходимы LILO в процессе загрузки (включая ядро Linux), располагаются в каталоге /boot, который обычно находится в корневом каталоге загрузочного раздела. Приведем несколько рекомендаций, которым вы должны следовать при выборе места для размещения LILO и необходимых ему данных.

Нужно иметь в виду, что некоторые источники рекомендуют размещать swap-раздел на диске, отличном от диска, где нахо-

дится основной раздел "Linux native". В этом случае повышается быстродействие системы.

Для нормальной установки самой системы Red Hat Linux достаточно 500 Мбайт, а если при установке сразу исключить часть предлагаемых по умолчанию пакетов, то инсталляция потребует приблизительно 300 Мбайт. Полная версия операционной системы в настоящий момент составляет примерно 2,5—3 Гбайт.

Если имеется свободный диск или можно освободить имеющийся раздел достаточного объема, то разумно сразу переходить к установке. Однако наиболее вероятно, что вы должны будете устанавливать Linux на диск, который уже содержит данные и программное обеспечение других операционных систем. В этом случае возникает необходимость нового разбиения жесткого диска.

Обычно разбиение диска производится с помощью программы `fdisk` для Linux, которая запускается на одном из этапов установки. Кроме этого, в современных дистрибутивах имеются специальные программные оболочки, являющиеся надстройками утилиты `fdisk` и позволяющие графически представлять разбиение диска на разделы.

Программа `fdisk` имеет интерфейс командной строки и может запускаться так:

```
fdisk <drive>
```

Здесь *<drive>* — имя устройства в Linux, для которого нужно выделить раздел. Например, если нужно выполнить разбивку для первого диска, можно использовать команду:

```
#fdisk /dev/hda
```

Если нужно создать разделы для Linux более чем на одном диске, выполните `fdisk` отдельно для каждого диска.

После разбиения дискового пространства можно получить информацию на консоли:

```
Command (m for help):  p
Disk /dev/hda:  16 heads, 38 sectors, 683 cylinders
```

```
Units = cylinders of 608 * 512 bytes
   Device Boot Begin Start End  Blocks Id System
/dev/hda1   *   1      1   203  61693   6 DOS  16-bit >=32M
/dev/hda2      204   204   473  82080  81 Linux/MINIX
/dev/hda3      474   474   507  10336  81 Linux/MINIX
```

Нужно задокументировать эту информацию, поскольку она может пригодиться в дальнейшем.

Некоторые дистрибутивы Linux требуют перезагрузки системы после окончания работы `fdisk`. Это позволяет изменениям в таблице разделов стать действительными для последующей инсталляции. Новые версии `fdisk` автоматически изменяют соответствующую информацию в ядре, так что перезагрузка не требуется. Чтобы обезопасить себя, после выполнения `fdisk` вам следует снова загрузить средства инсталляции, как и раньше — перед продолжением инсталляции.

Для того чтобы начать установку Red Hat Linux, нужно перезагрузить компьютер и вставить первый инсталляционный компакт-диск. После небольшой задержки на мониторе должна появиться строка приглашения

```
boot:
```

На экране выдаются несколько строк подсказки по опциям (вариантам) загрузки. С каждым вариантом загрузки могут быть выданы на экран одна или несколько порций соответствующей подсказки. Чтобы их просмотреть, нажмите одну из клавиш, указанных в нижней части экрана.

Если в течение минуты после появления приглашения не нажата ни одна клавиша, автоматически стартует процедура установки. Чтобы отменить автоматический старт, нажмите одну из клавиш, вызывающих подсказку.

В процессе инсталляции вы можете пользоваться системой виртуальных консолей, переключаясь с одной на другую нажатием комбинации клавиш <Alt>+<F*x*>.

Все вновь созданные разделы должны быть отформатированы. Выбор раздела для форматирования и отмена выбора осуществляются клавишей <Пробел>. Дополнительная опция позволяет задать проверку на наличие плохих блоков в процессе форматирования.

После того как сконфигурированы разделы и отмечены те из них, которые подлежат форматированию, следует выбрать программные пакеты для установки. Выбирать можно компоненты системы — целые группы пакетов.

Вначале предлагается список компонентов системы, т. е. групп пакетов. Выбор и отмена выбора производятся клавишей <Пробел>. В конце списка компонентов есть строка **Everything**, которая позволяет отметить для установки все пакеты, включенные в дистрибутив Red Hat Linux.

Если нужно провести выбор отдельных пакетов, отметьте строку **Select individual packages**.

Если флажок выбора отдельных пакетов установлен, то программа выдаст список допустимых групп. Теперь можно указать группу, из которой вы хотите установить только часть пакетов, и нажать клавишу <Enter>. Вам будет показан список пакетов данной группы, в котором можно клавишей <Пробел> выбрать для установки отдельные пакеты.

Некоторые пакеты (как, например, ядро и отдельные библиотеки) необходимы в любой конфигурации Red Hat Linux, поэтому их нельзя ни включить в состав системы, ни отменить. Получить детальную информацию о выбранном пакете можно, нажав клавишу <F1>.

Обратите внимание на то, что при добавлении или отмене пакета для установки программа инсталляции сразу же пересчитывает объем дискового пространства, необходимого для инсталляции выбранных пакетов. Это позволяет выбрать такую конфигурацию системы, которая сможет разместиться на жестком диске.

Если вы впервые устанавливаете Linux и не можете оценить необходимость того или иного пакета, то лучше снимать отметки с

максимально возможного числа пакетов. Программные пакеты можно установить отдельно в процессе работы с системой.

После того как все необходимые пакеты отмечены для установки, программа инсталляции сообщает, что файл протокола install.log, содержащий список всех устанавливаемых пакетов, будет записан в каталог /tmp.

С этого момента начинается собственно инсталляция программ. Вначале выполняется форматирование тех разделов, которые были выбраны. После завершения форматирования производится установка пакетов. В это время на экране отображается информация о текущем состоянии устанавливаемого пакета.

В процессе установки программа инсталляции предлагает пользователю выбрать службы, которые будут запускаться при загрузке операционной системы. Такое предложение появляется в диалоговом окне **Services**. Просмотрите список и выберите те из них, которые должны автоматически запускаться при перезагрузке. Нажатие клавиши <F1> приводит к появлению подсказки по той службе, которая в этот момент выделена.

Как и при выборе пакетов для установки, нужно минимизировать число запускаемых служб, чтобы уменьшить загрузку оперативной памяти. После завершения установки и запуска Linux всегда можно изменить перечень загружаемых служб, используя команды `/usr/sbin/ntsysv` или `/sbin/chkconfig`.

Если в процессе инсталляции были установлены пакеты системы X Window, то после конфигурирования мыши нужно настроить эту систему. Настройка сервера XFree86 осуществляется путем запуска программы `Xconfigurator`.

Хочу отметить, что в версии 6.0 программа `Xconfigurator` запускается в самом конце процедуры инсталляции, после установки всех компонентов файловой системы. Это сделано для того, чтобы в случае сбоя программы `Xconfigurator` можно было запустить систему в текстовом режиме и сконфигурировать X Window. Кроме того, `Xconfigurator` тестирует X Window в процессе инсталляции.

Программа `Xconfigurator` предлагает после инсталляции загружать систему сразу в графическом режиме. Лучше отказаться от этого варианта и запускать систему, хотя бы первое время, в консольном режиме. После анализа конфигурации можно настроить систему так, чтобы она загружалась в графический режим. Графическую систему можно запускать вручную с помощью команды `startx`.

При настройке X Window программа `Xconfigurator` пытается определить тип видеокарты вашего компьютера. Если это не удается, программа выдает список известных ей видеокарт. Если тип видеокарты, установленной на компьютере, совпадает с одной из списка, нужно ее выбрать и нажать клавишу <Enter>. Если видеокарты нет в списке, значит, XFree86 не поддерживает ее. Однако при наличии достаточной информации о видеокарте можно попытаться установить ее вручную, выбрав вариант **Unlisted Card**.

После выбора видеокарты инсталлируется соответствующий XFree86-сервер, и `Xconfigurator` выдает на экран список поддерживаемых мониторов. Если ваш монитор есть в списке, выбирайте его. Если монитора в списке нет, тогда нужно выбрать опцию **Custom**. В этом случае `Xconfigurator` попросит ввести параметры монитора вручную.

Не рекомендуется выбирать монитор, похожий на имеющийся, если только нет уверенности в том, что возможности вашего монитора перекрывают возможности того типа монитора, который будет выбран. В противном случае можно легко повредить монитор.

В качестве рекомендации можно посоветовать выбрать минимально возможные значения параметров (горизонтальной и вертикальной развертки), поскольку после инсталляции Linux можно повторно сконфигурировать X Window.

Кроме этого, `Xconfigurator` попросит выбрать видеорежим, который вы хотите использовать; клавишей <Пробел> отметьте один или несколько видеорежимов. После настройки X Window выбранные значения сохраняются в файле /etc/X11/XF86Config.

Для загрузки операционной системы Red Hat Linux без помощи дискеты требуется установить загрузчик LILO. Возможны два варианта установки LILO:

- в главный загрузочный сектор диска (Master Boot Record, MBR). Это предпочтительный вариант для установки LILO, если только MBR уже не занят загрузчиком другой операционной системы. Если LILO устанавливается в MBR, то в процессе загрузки на экран будет выдано приглашение в виде `boot:`, что позволит выбрать загружаемую операционную систему, не только Red Hat Linux;
- в первый сектор корневого раздела. Такой вариант можно применить в том случае, если уже используется другой загрузчик, который первым перехватит управление процессом загрузки. Этот загрузчик нужно сконфигурировать так, чтобы он вызвал LILO, который, в свою очередь, загрузит Linux.

При установке загрузчика LILO можно передать ему какие-либо параметры по умолчанию, а сама установка системы завершается перезагрузкой компьютера. При необходимости можно повторять установку несколько раз, исправляя обнаруженные ошибки или делая исправления в ходе последующей настройки без переустановки системы.

5.4. Запуск и остановка UNIX

Этапы запуска и останова операционной системы являются начальной и конечной точками отсчета работы UNIX. Во время запуска выполняются важнейшие процедуры инициализации, которые в дальнейшем могут оказывать решающее влияние на работоспособность системы. Знание внутренних механизмов процесса загрузки позволяет правильно настроить системные сервисы и устройства, точно так же понимание последовательности выключения системы помогает предотвратить потерю данных. Рассмотрим процессы запуска и останова для операционных систем FreeBSD, Solaris и Linux.

5.4.1. Загрузка FreeBSD

Процесс загрузки FreeBSD предоставляет различные варианты гибкой настройки системы, позволяя вам выбирать одну из нескольких операционных систем, установленных на одном и том же хосте, или даже одну из нескольких версий той же самой операционной системы. Хочу заметить, что здесь описан процесс загрузки и останова FreeBSD только для архитектуры x86.

На оборудовании архитектуры x86 за загрузку операционной системы отвечает BIOS (Basic Input/Output System, базовая система ввода/вывода). Для этого BIOS ищет на жестком диске MBR (Master Boot Record, главная загрузочная запись), которая должна располагаться в определенном месте на диске. BIOS может загрузить и запустить MBR, и предполагается, что MBR может взять на себя остальную работу, связанную с загрузкой операционной системы.

Если на вашем диске установлена только одна операционная система, то стандартной MBR будет достаточно. Программа, находящаяся в MBR, выполняет поиск на диске первого загрузочного раздела (части диска), после чего запускает из этого раздела код загрузки операционной системы.

Если на ваших дисках установлено несколько операционных систем, то вы можете установить другую MBR, ту, что может выдать список различных операционных систем и позволит вам выбрать одну из них для загрузки. Операционная система FreeBSD поставляется с одной из таких MBR, которую можно установить на диск. Другие производители операционных систем также предоставляют свои MBR.

Далее процесс загрузки проходит по такому сценарию: вызывается вторичный загрузчик, известный как загрузчик этапов 1 и 2 операционной системы FreeBSD. Вторичный загрузчик загружает в оперативную память ядро, которое находится в файле /boot/kernel/kernel. После этого работа вторичного загрузчика заканчивается.

Затем стартует ядро, которое начинает опознавать устройства и выполняет их инициализацию. После завершения процесса сво-

ей загрузки ядро передает управление пользовательскому процессу с именем `init`, который выполняет проверку дисков на возможность использования. Затем `init` запускает пользовательский процесс настройки ресурсов, который монтирует файловые системы, производит настройку сетевых адаптеров для работы в сети и вообще осуществляет запуск всех процессов, обычно выполняемых в системе FreeBSD при загрузке.

MBR для FreeBSD находится в /boot/boot0. Это копия MBR, т. к. настоящая MBR должна располагаться в специальном месте диска, вне области FreeBSD.

Загрузчик boot0 очень прост, т. к. программа в MBR может иметь размер, не превышающий 512 байтов. Если вы установили MBR FreeBSD и несколько операционных систем на ваш жесткий диск, то во время загрузки увидите приблизительно такие строки:

```
F1 DOS
F2 FreeBSD
F3 Linux
F4 ??
F5 Drive 1

Default: F2
```

Известно, что другие операционные системы, в частности, Windows 2000/XP, записывают поверх существующей MBR свою собственную. Если это произошло в вашем случае, или же вы хотите заменить существующую MBR на MBR от FreeBSD, то воспользуйтесь следующей командой:

```
# fdisk -B -b /boot/boot0 device
```

Здесь `device` является устройством, с которого вы загружаетесь, таким, как ad0 в случае первого диска IDE, ad2 в случае первого диска IDE на втором контроллере IDE, da0 для первого диска SCSI и т. д.

Если кроме FreeBSD на машине установлена ОС Linux, то можно сделать так, чтобы процесс загрузки управлялся через LILO. Для этого можно отредактировать файл /etc/lilo.conf для FreeBSD или выбрать опцию **Leave The Master Boot Record Untouched** в процессе установки FreeBSD. Если вы установили менеджер загрузки FreeBSD, то можете снова загрузить Linux и изменить конфигурационный файл /etc/lilo.conf для LILO, добавив следующий параметр:

```
other=/dev/hdXY
table=/dev/hdb
loader=/boot/chain.b
label=FreeBSD
```

После этого можно будет загружать FreeBSD и Linux посредством LILO. В нашем примере мы указываем `XY` для обозначения номера диска и раздела. Если вы используете диск SCSI, то вам может потребоваться замена устройства /dev/hd*XY* на другое устройство, например, /dev/sd*XY*, где снова используется обозначение *XY*. Строка

```
loader=/boot/chain.b
```

может быть опущена, если обе операционные системы располагаются на одном и том же диске. Для того чтобы изменения были восприняты системой, нужно выполнить программу

```
/sbin/lilo -v
```

Первый и второй этапы загрузки инициируются с помощью двух небольших программ, которые расположены в одной и той же области диска. Обе программы располагаются в загрузочном секторе загрузочного раздела, т. е. там, где `boot0` или любая другая программа из MBR ожидает найти программу, которую следует запустить для продолжения процесса загрузки. Файлы в каталоге /boot являются копиями реальных файлов, которые хранятся вне файловой системы FreeBSD.

Загрузчик boot1 имеет размер, не превышающий 512 байтов. Его основные задачи — найти метку диска, хранящую информацию о разделе FreeBSD, и передать управление загрузчику boot2.

Загрузчик boot2 устроен несколько сложнее и предназначен для работы с файловой системой FreeBSD в объеме, достаточном для нахождения в ней файлов. Кроме того, он может предоставлять простой интерфейс для выбора и передачи управления основному загрузчику, входящему в ядро операционной системы.

Этот загрузчик более сложен, чем boot2, и предоставляет удобный и простой интерфейс для настройки процесса загрузки. Далее показан образец экрана boot2:

```
>> FreeBSD/i386 BOOT

Default: 0:ad(0,a)/kernel

boot:
```

Если вам когда-либо понадобится заменить установленные boot1 и boot2, то используйте утилиту `disklabel`:

```
# disklabel -B diskslice
```

Здесь *`diskslice`* обозначает диск и раздел, с которого выполняется загрузка, например, ad0s1 в случае первого раздела на первом диске IDE.

Передача управления основному загрузчику является последним этапом в процессе начальной загрузки, а сам загрузчик находится в файловой системе, обычно это /boot/loader.

Загрузчик представляет собой удобный в использовании инструмент для настройки при помощи простого набора команд, управляемого более мощным интерпретатором с более сложным набором команд. Во время инициализации загрузчик пытается произвести поиск консоли, дисков и определить, с какого диска он был запущен. Соответствующим образом он задает значения переменных и запускает интерпретатор, которому могут передаваться пользовательские команды как из скрипта, так и в интерактивном режиме.

Затем загрузчик читает файл /boot/loader.rc, который по умолчанию использует файл /boot/defaults/loader.conf, устанавливающий значения по умолчанию для системных переменных. Кроме того, загрузчик читает файл /boot/loader.conf для изменения в

этих переменных. Далее с этими переменными работает loader.rc, загружающий выбранные модули и ядро.

И, наконец, по умолчанию загрузчик выдерживает 10-секундную паузу, ожидая нажатия клавиши <Enter>, и загружает ядро, если этого не произошло. Если ожидание было прервано, пользователю выдается приглашение ввести одну из набора команд, с помощью которых пользователь может изменить значения переменных, выгрузить все модули, загрузить модули и продолжить процесс загрузки или перезагрузить машину.

Рассмотрим наиболее часто используемые команды загрузчика. Полное описание всех имеющихся команд можно найти на странице справки о команде `loader`. Перечень основных команд приводится далее:

- `autoboot` *`секунды`* — продолжает загрузку ядра, если не будет прервана в течение указанного в секундах промежутка времени. Команда выводит счетчик, и по умолчанию выдерживается интервал в 10 секунд;
- `boot [-`*`параметры`*`] [`*`имя_ядра`*`]` — продолжает процесс загрузки ядра, если оно было указано с заданными параметрами;
- `boot-conf` — повторно выполняет процесс автоматической настройки модулей на основе переменных, что был произведен при загрузке. Это имеет смысл, если ранее вы выполнили команду `unload`, изменили некоторые переменные, например, наиболее часто меняемую `kernel`;
- `help [`*`тема`*`]` — выводит сообщения подсказки из файла /boot/loader.help. Если в качестве темы указано слово index, то появляется список имеющихся тем;
- `include` *`имя_файла`* `...` — запускает файл с указанным именем. Файл считывается, и его содержимое интерпретируется строка за строкой. Ошибка приводит к немедленному прекращению выполнения команды `include`;
- `load [-t` *`тип`*`]` *`имя_файла`* — загружает ядро, модуль ядра или файл указанного типа с заданным именем. Все аргументы после имени файла передаются в файл;

- `ls [-l] [маршрут]` — выводит список файлов по указанному маршруту или в корневом каталоге, если маршрут не был задан. Если присутствует параметр `-l`, будут выводиться и размеры файлов;
- `lsdev [-v]` — выводит список всех устройств, с которых могут быть загружены модули. Если указан параметр `-v`, выводится дополнительная информация;
- `lsmod [-v]` — выводит список загруженных модулей. Если указан параметр `-v`, то отображается дополнительная информация;
- `more имя_файла` — выводит указанный файл с паузой при отображении каждой строки;
- `Reboot` — выполняется немедленная перезагрузка машины;
- `set переменная`, `set переменная=значение` — задает значения переменных окружения загрузчика;
- `unload` — удаляет из памяти все загруженные модули.

Вот пара примеров практического использования загрузчика:

- загрузка ядра обычным образом, но в однопользовательском режиме:

```
boot -s
```

- выгрузка обычных ядра и модулей, а потом загрузка старого (или другого) ядра:

```
unload
load kernel.old
```

Вы можете использовать kernel.GENERIC для обозначения стандартного ядра, поставляемого на установочном диске, или kernel.old для обращения к ранее установленному ядру (после того, как, например, вы обновили или сконфигурировали новое ядро).

Замечание

Для загрузки ваших обычных модулей с другим ядром используйте такие команды:

```
unload
set kernel="kernel.old"
boot-conf
```

Для загрузки скрипта конфигурации ядра (автоматизированный скрипт, который выполняет то, что вы обычно делаете в конфигураторе ядра во время загрузки) нужно выполнить команду:

```
load -t userconfig_script /boot/kernel.conf
```

Как только ядро окажется загруженным при помощи загрузчика (обычный способ) или boot2 (минуя загрузчик), оно проверяет флаги загрузки, если они есть, и действует соответствующим образом. Вот наиболее часто используемые флаги загрузки:

- `-a` — во время инициализации ядра запрашивать устройство для его монтирования в качестве корневой файловой системы;
- `-C` — загрузить с компакт-диска;
- `-c` — перейти в режим UserConfig для конфигурации ядра во время загрузки;
- `-s` — после загрузки перейти в однопользовательский режим;
- `-v` — во время запуска ядра выводить более подробную информацию.

> ***ЗАМЕЧАНИЕ***
>
> Есть и другие флаги загрузки, обратитесь к странице справочника по команде `boot` для получения более подробной информации.

Во время начального запуска системы загрузчик `loader` производит чтение файла device.hints. В этом файле хранится необходимая для загрузки ядра информация, задаваемая в виде переменных, которую иногда называют *"подсказками устройств"* (device hints). Эти подсказки используются драйверами устройств для их конфигурации.

Подсказки для устройств могут быть заданы и в приглашении начального загрузчика на третьем этапе загрузки. Переменные могут быть добавлены при помощи команды `set`, удалены по-

средством команды `unset` и просмотрены командой `show`. В этот момент могут быть также переопределены переменные, заданные в файле /boot/device.hints. Подсказки для устройств, введенные в начальном загрузчике, не сохраняются и при следующей перезагрузке будут утеряны.

После загрузки системы для выдачи значений всех переменных можно воспользоваться командой `kenv`.

Синтаксис записей в файле /boot/device.hints таков: в каждой строке определяется по одной переменной, в качестве метки начала комментария используется стандартный символ `#`. Строки строятся следующим образом:

`hint.driver.unit.keyword="`*значение*`"`

Синтаксис для начального загрузчика на третьем этапе таков:

`set hint.driver.unit.keyword=`*значение*

Опция `driver` определяет имя драйвера устройства, `unit` соответствует порядковому номеру модуля устройства, а `keyword` является ключевым словом хинта. В качестве ключевых слов могут применяться следующие опции:

- `at` — шина, к которой подключено устройство;
- `port` — начальный адрес используемого диапазона ввода/вывода (I/O);
- `irq` — используемый номер запроса на прерывание;
- `drq` — номер канала DMA;
- `maddr` — физический адрес памяти, занимаемый устройством;
- `flags` — различные битовые флаги для устройства;
- `disabled` — если установлено в значение 1, то устройство не используется.

Драйверы устройств могут поддерживать (и даже требовать) другие хинты, здесь не перечисленные, поэтому рекомендуется просматривать документацию к этим драйверам. Для получения дополнительной информации обратитесь к страницам справки по `device.hints`, `kenv`, `loader.conf` и `loader`.

После того как ядро завершит загрузку, оно передает управление пользовательскому процессу `init`, который расположен в файле /sbin/init или в файле, маршрут к которому указан в переменной `init_path` загрузчика.

Процесс автоматической перезагрузки проверяет целостность имеющихся файловых систем. Если целостность нарушена и утилита `fsck` не может исправить положение, то процесс `init` переводит систему в однопользовательский режим для того, чтобы системный администратор сам разобрался с возникшими проблемами.

В этот режим можно перейти во время процесса автоматической перезагрузки, при ручной загрузке с параметром `-s` или установив переменную `boot_single` для программы `loader`.

Переход в данный режим может быть также вызван запуском команды `shutdown` из многопользовательского режима.

Если режим доступа к системной консоли console установлен в файле /etc/ttys в значение `insecure`, то система выведет запрос на ввод пароля суперпользователя root перед переходом в однопользовательский режим.

Рассмотрим пример незащищенной консоли в /etc/ttys:

```
# name getty                type    status      comments
#
# Если консоль помечена как "insecure", то init будет
# запрашивать пароль пользователя root при переходе
# в однопользовательский режим.
console none                                unknown off insecure
```

Здесь нужно сделать одно замечание. Обозначение консоли как `insecure` говорит о том, что вы считаете физический доступ к консоли незащищенным и хотите, чтобы только тот, кто знает пароль, пользователя root, мог воспользоваться однопользовательским режимом. Но это вовсе не означает, что вы хотите работать с консолью небезопасным способом. Таким образом, если вы желаете добиться защищенности, указывайте `insecure`, а не `secure`.

Система переходит в многопользовательский режим в одном из случаев: если процесс `init` определит, что ваши файловые системы находятся в полном порядке, или после того, как пользователь выйдет из однопользовательского режима. Работа в многопользовательском режиме начинается с настройки ресурсов системы.

Система настройки ресурсов считывает установки, используемые по умолчанию, из файла /etc/defaults/rc.conf, а настройки, специфичные для конкретной системы, — из файла /etc/rc.conf. После этого выполняется монтирование файловых систем, перечисленных в файле /etc/fstab, запуск сетевых служб, различных системных демонов и, наконец, выполнение скриптов запуска дополнительно установленных пакетов.

Страница справочника для файла rc является хорошим источником информации о системе настройки ресурсов, так же как и самостоятельное изучение скриптов.

Остановка системы может происходить посредством утилиты `shutdown`, при этом процесс `init` попытается запустить скрипт /etc/rc.shutdown, после чего всем процессам будет послан сигнал `SIGTERM`, а затем и сигнал `SIGKILL` тем процессам, которые еще не завершили свою работу.

Для выключения машины с FreeBSD на аппаратных платформах и системах, которые поддерживают управление электропитанием, просто воспользуйтесь командой

```
shutdown -p now
```

для немедленного отключения электропитания. Чтобы перезагрузить систему FreeBSD, выполните команду

```
shutdown -r now
```

Для запуска команды `shutdown` вам необходимо быть пользователем root или членом группы `operator`. Кроме того, можно также воспользоваться командами `halt` и `reboot`. Для получения дополнительной информации обратитесь к соответствующим разделам справки и к справочной странице по команде `shutdown`.

Для управления электропитанием требуется наличие поддержки ACPI (Advanced System Configuration and Power Interface, усовершенствованный интерфейс конфигурирования и управления системой энергопитания) в ядре или в виде загруженного модуля при использовании FreeBSD 5.*x*.

5.4.2. Запуск Solaris 9

Операционная система Solaris может быть загружена для работы в одном из нескольких режимов. Вначале рассмотрим общие вопросы запуска операционной системы, после чего проанализируем особенности загрузки Solaris для работы в различных режимах.

Напомню, что мы рассматриваем процесс загрузки для операционной системы на платформе x86. Работа операционной системы, например, на платформе SPARC, не отличается от x86, но процессы запуска различны.

Если имеется много разделов, то загрузчик, размещенный в разделе Solaris, спросит, с какого раздела нужно загружать систему. Этот раздел должен быть отмечен, как активный. Если в течение нескольких секунд ответ не будет получен, то по умолчанию загрузится операционная система Solaris.

Сам загрузчик работает довольно просто. Его нельзя настроить, нельзя изменить загрузочный раздел по умолчанию на другой, с операционной системой, отличной от Solaris. Кроме того, нельзя изменить время задержки и описание раздела.

Операционная система предполагает несколько вариантов загрузки. Рассмотрим их по порядку, но вначале остановимся на некоторых общих вопросах процедуры запуска Solaris.

После инициализации ядра командой `boot` начинается загрузка системы. В самой первой фазе загружается ядро операционной системы /kernel/unix. Оно расположено в корневом каталоге диска.

Само ядро состоит из небольшого базового (core) модуля и многочисленных загружаемых модулей (loadable kernel modules). Большинство из этих модулей автоматически загружается во

время начальной загрузки системы. Остальные модули, такие, например, как драйверы устройств, загружаются ядром по мере необходимости.

Когда загружается ядро, система считывает в память файл /etc/system. В нем содержится список модулей, подлежащих загрузке, и параметры загрузки этих модулей. Данный файл может редактироваться в процессе настройки системы.

После этого управление процессом загрузки передается ядру, которое выполняет инициализацию системы. Система переходит в одно из восьми состояний запуска, называемых также состояниями инициализации (init). Состояние запуска 4 в настоящее время не используется. Все состояния представлены в табл. 5.3.

Таблица 5.3. *Состояния запуска системы*

Состояние запуска	Описание
0	В этом состоянии останавливаются системные службы и демоны, а файловые системы демонтируются
S, s	Однопользовательский режим работы. Используется системными администраторами для настройки и восстановления системы. Регистрация на системной консоли разрешается только суперпользователю root, остальные пользователи при переходе в этот режим отсоединяются от системы. Базовые системные службы продолжают функционировать в этом режиме, а файловые системы остаются смонтированными. Остальные службы останавливаются в определенной последовательности
1	Однопользовательский режим работы. Используется системными администраторами для настройки и восстановления системы. Регистрация на системной консоли разрешается только суперпользователю root, остальные пользователи при переходе в этот режим отсоединяются от системы. Базовые системные службы продолжают функционировать в этом режиме, а файловые системы остаются смонтированными. Остальные службы останавливаются в определенной последовательности. Все зарегистрированные пользователи продолжают оставаться в системе

Таблица 5.3 *(окончание)*

Состояние запуска	Описание
2	Работа в многопользовательском режиме. В этом режиме не используются сетевые файловые системы (NFS). Кроме того, монтируется файловая система /usr и очищаются каталоги /tmp и /var/tmp. Выполняется загрузка сетевых интерфейсов. Запускается служба печати `lp` и демон `cron`. Наконец, запускается система электронной почты `sendmail`
3	Обычная работа в многопользовательском режиме. Отличается от состояния 2 тем, что запущена файловая система NFS
4	Многопользовательский режим (в настоящее время не используется)
5	Состояние выключения питания. Выполняется останов системы таким образом, чтобы можно было безопасно отключить питание
6	Перезагрузка системы

Более подробно перечень задач, которые могут выполняться в каждом состоянии, приведен в табл. 5.4.

Таблица 5.4. *Перечень задач, которые могут выполняться в каждом состоянии*

Состояние запуска	Задачи
0	Используется, когда необходимо безопасно отключить питание системы. Может выполняться только системным администратором
S, s	Используется для выполнения системных задач, требующих отключения пользователей от системы. К числу таких задач можно отнести восстановление файловых систем, выполнение резервного копирования системы

Таблица 5.4 *(окончание)*

Состояние запуска	Задачи
1	Служит для выполнения системных задач, но дополнительные пользователи в систему не допускаются. К числу таких задач можно отнести восстановление файловых систем или создание резервных копий данных, с которыми пользователи, находящиеся в системе, не работают
2	Применяется для нормальной работы, когда не требуется доступ пользователей к разделяемым ресурсам локальной файловой системы. Работают все системные службы и демоны, кроме `syslog` и NFS
3	Используется для нормальной работы, когда доступна файловая система NFS
5	Предназначено для безопасного выключения питания. Система останавливается таким образом, чтобы можно было безопасно отключить питание. Все пользователи выводятся из системы, а системные службы останавливаются в определенной последовательности
6	Используется для закрытия системы с целью последующей перезагрузки системы

Ядро операционной системы в первую очередь запускает процесс, называемый *подкачкой* (swapper). Процесс подкачки имеет идентификатор 0 и называется `sched` (планировщиком). Самое первое, что делает `sched`, — запускает процесс `init`.

Команда /sbin/init порождает процессы для запуска системы на основе записей, находящихся в файле /etc/inittab. Процесс `init` является родительским процессом по отношению к остальным процессам. Он анализирует записи в файле /etc/inittab и определяет последовательность запуска, остановки и перезапуска остальных процессов. Каждая запись в файле /etc/inittab имеет следующий формат:

```
id:runlevel:action:process
```

Вот назначение полей записи:

- `id` — уникальный идентификатор, состоящий из одного или двух символов;
- `runlevel` — уровень запуска. Задает уровень запуска, на котором должна обрабатываться эта запись. Уровни запуска соответствуют конфигурации процессов в системе и представлены числами от 0 до 6. Например, если система работает на уровне запуска 1, обрабатываются только записи, в поле `runlevel` которых указано значение 1;
- `action` — способ запуска процесса. Допустимы следующие способы:
  - `respawn` — если процесс не существует, то создать его, а когда процесс завершит существование, перезапустить его. Если процесс выполняется, продолжить просмотр файла /etc/inittab;
  - `wait` — на этом уровне запускается процесс и ожидается его завершение. Для всех последующих просмотров файла /etc/inittab процессом `init` на том же уровне данная запись игнорируется;
  - `once` — процесс запускается и `init` не ожидает его завершения. При завершении выполнения процесс не перезапускается;
  - `boot` — процесс выполняется только при первом просмотре файла /etc/inittab в момент загрузки процесса `init`. Процесс `init` запустит процесс и не будет ожидать его завершения; а когда процесс завершится, `init` его не перезапустит;
  - `bootwait` — процесс выполняется только при первом переходе процесса `init` с однопользовательского в многопользовательский режим после загрузки системы. Демон `init` запускает процесс, ждет его завершения и после этого его уже не перезапускает;
  - `powerfail` — процесс выполняется только при получении демоном `init` сигнала сбоя по питанию;

- `off` — работающему процессу посылается сигнал `SIGTERM`, затем процесс завершается принудительно посылкой сигнала `SIGKILL`. Если процесс не существует, запись игнорируется;
- `ondemand` — процесс выполняется, так же как и для опции `respawn`;
- `initdefault` — выполняется только при первоначальном вызове `init`. Процесс `init` использует эту запись, чтобы определить, на какой уровень выполнения первоначально переходить;
- `sysinit` — процесс выполняется до того, как `init` выполнит обращение к консоли. Демон `init` ждет завершения процесса, прежде чем продолжить работу;

❒ `process` — имя команды, подлежащей выполнению. Задает команду, которую надо выполнить. Перед этим полем добавляется команда `exec`. В поле `process` можно указывать любую допустимую команду.

Записи разделяются символами новой строки; но если перед символом новой строки присутствует обратный слэш (\), то запись продолжается на следующей строке. Максимальная длина записи составляет 512 символов. В поле `process` можно использовать комментарии. На количество записей в файле inittab никаких ограничений не налагается (только на размер одной записи).

Когда процесс `init` получает запрос на изменение уровня выполнения, всем процессам, для которых в поле `runlevel` не указан целевой уровень, посылается сигнал `SIGTERM`, а через 5 секунд их работа принудительно прерывается сигналом `SIGKILL`. Как уже отмечалось, поле `runlevel` может задавать несколько уровней выполнения для процесса в виде любой комбинации значений от 0 до 6. Если уровень выполнения не задан, предполагается, что процесс может работать на всех уровнях выполнения.

Новые записи можно добавлять в файл /etc/inittab в любой момент. Однако процесс `init` будет ждать наступления одного из

перечисленных выше условий, прежде чем просматривать файл /etc/inittab. Для немедленной инициализации только что добавленного процесса нужно выполнить одну из команд:

```
init Q
init q
```

Эти команды вынуждают процесс `init` немедленно пересмотреть файл /etc/inittab.

Если процесс `init` получает сигнал о сбое питания, он ищет в файле /etc/inittab специальные записи типа `powerfail` и `powerwait`. Соответствующие этим записям процессы запускаются (если позволяет `runlevel`) перед выполнением любой дальнейшей обработки. Процесс `init` может выполнить необходимые действия по завершению работы операционной системы и записи информации при отключении питания.

Пример содержимого файла /etc/inittab представлен далее:

```
ap::sysinit:/sbin/autopush -f /etc/iu.ap
ap::sysinit:/sbin/soconfig -f /etc/sock2path
fs::sysinit:/sbin/rcS sysinit >/dev/msglog 2<>/dev/msglog
</dev/console
is:3:initdefault:
p3:s1234:powerfail:/usr/sbin/shutdown -y -i5 -g0 >/dev/msglog
2<>/dev/msglog
sS:s:wait:/sbin/rcS >/dev/msglog 2<>/dev/msglog </dev/console
s0:0:wait:/sbin/rc0 >/dev/msglog 2<>/dev/msglog </dev/console
s1:1:respawn:/sbin/rc1 >/dev/msglog 2<>/dev/msglog
</dev/console
s2:23:wait:/sbin/rc2 >/dev/msglog 2<>/dev/msglog
</dev/console
s3:3:wait:/sbin/rc3 >/dev/msglog 2<>/dev/msglog </dev/console
s5:5:wait:/sbin/rc5 >/dev/msglog 2<>/dev/msglog </dev/console
s6:6:wait:/sbin/rc6 >/dev/msglog 2<>/dev/msglog </dev/console
fw:0:wait:/sbin/uadmin 2 0 >/dev/msglog 2<>/dev/msglog
</dev/console
```

```
of:5:wait:/sbin/uadmin 2 6 >/dev/msglog 2<>/dev/msglog
</dev/console

rb:6:wait:/sbin/uadmin 2 1 >/dev/msglog 2<>/dev/msglog
</dev/console

sc:234:respawn:/usr/lib/saf/sac -t 300

co:234:respawn:/usr/lib/saf/ttymon -g -h -p "`uname -n` con-
sole login: " -T

sun -d /dev/console -l console -m ldterm,ttcompat
```

Рассмотрим примеры загрузки операционной системы в различных режимах и начнем с многопользовательского режима, который иногда называют *режимом 3* (run level 3). Это основной режим работы операционной системы — в этом режиме выполняются все системные функции в полном объеме, и допускается работа пользователей.

После перезагрузки или при включении компьютера через несколько секунд появляется меню **Current Boot Parameters**. Нажатие клавиши <B> вызывает немедленную загрузку системы в многопользовательский режим (режим 3). Если выбор не произведен в течение нескольких секунд, система автоматически начинает загрузку в данный режим.

Признаком успешной загрузки в многопользовательский режим является приглашение:

```
hostname console login:
```

Однопользовательский режим работы служит для настройки системы и имеет альтернативное название — *режим S*. После включения машины и появления меню **Current Boot Parameters** нужно ввести команду

```
b -s
```

Если в течение нескольких секунд не будет сделан выбор, то система автоматически загрузится в режим 3.

Если система потребует пароль суперпользователя root, следует ввести его. Для проверки того, что система работает в однопользовательском режиме, нужно ввести команду

```
# who -r
   .       run-level S  Jul 19 14:37     S      0  3
```

После выполнения всех административных задач можно перейти в многопользовательский режим (режим 3), нажав комбинацию клавиш <Ctrl>+<D>.

Работающую операционную систему можно остановить, если в режиме суперпользователя root выполнить команду

```
# init 0
```

После того как появится приглашение

```
Type any key to continue
```

нужно нажать клавишу <Enter>.

Для перезагрузки системы из режима суперпользователя нужно набрать команду

```
#init 6
```

После того как появится приглашение

```
Type any key to continue
```

следует нажать клавишу <Enter>.

Если система не отвечает на ввод команд, то единственный выход из этой ситуации — нажать кнопку Reset на системном блоке или использовать кнопку включения/выключения питания.

5.4.3. Запуск и останов Linux

Операционная система Linux может быть запущена как с дискет, так и с жесткого диска. При включении компьютера BIOS производит тестирование оборудования, а затем запуск операционной системы. Сперва выбирается устройство, с которого будет производиться запуск. Таким устройством может быть первый дисковод или первый жесткий диск, если он установлен. Первоначально считывается самый первый сектор, который называется загрузочным (Master Boot Record, MBR), т. к. у жесткого диска может быть несколько разделов и у каждого — свой загрузочный сектор.

В загрузочном секторе находится небольшая программа, которая загружает и запускает операционную систему. При загрузке с

дискеты в загрузочном секторе находится код, который обеспечивает только считывание ядра системы в определенную заранее область памяти. Загрузочная дискета для Linux не содержит никаких файловых систем. Ядро записано на дискете как последовательность блоков, т. к. это значительно упрощает процесс загрузки. Однако можно загружаться и с дискеты, на которой установлена какая-нибудь файловая система, используя загрузчик LILO.

При загрузке с жесткого диска код, расположенный в MBR, проверяет таблицу разделов (также расположенную в MBR), определяет активный раздел (раздел, используемый при загрузке), считывает загрузочный сектор этого раздела и запускает считанный код. Код, расположенный в загрузочном секторе активного раздела жесткого диска, выполняет те же функции, что и код, находящийся в загрузочном секторе дискеты: он считывает ядро из выбранного раздела, а затем запускает его. Однако здесь существует много тонкостей, т. к. использование отдельного раздела диска только для хранения кода ядра неэффективно, поэтому код, расположенный в загрузочном секторе раздела, не просто последовательно считывает информацию с диска, а выполняет считывание по секторам. Существует несколько способов решения этой проблемы, но наиболее простым из них является использование LILO-загрузчика (информацию по его установке и настройке можно просмотреть в документации по LILO).

При загрузке с использованием LILO обычно сразу же загружается и запускается ядро, заданное по умолчанию, однако можно сконфигурировать LILO так, чтобы можно было загрузить одно из нескольких возможных ядер или даже другую операционную систему, отличную от Linux.

Можно указать требуемое ядро или операционную систему во время загрузки. При нажатии клавиши <Alt>, <Shift> или <Ctrl> (после загрузки LILO) будет выдан запрос, где можно указать ядро или систему. Однако при конфигурировании можно установить опцию, при которой LILO будет всегда выдавать такой запрос, а также указать время, по истечении которого загружается ядро, установленное по умолчанию.

Существуют и другие загрузчики, подобные LILO, однако у него есть несколько полезных функций, которых нет в других загрузчиках, т. к. он был написан специально для Linux. Например, имеется возможность передачи ядру параметров во время загрузки или изменения некоторых опций, встроенных в ядро. Среди подобных загрузчиков (bootlin, bootactv) LILO является наилучшим выбором.

Загрузка системы как с жесткого диска, так и с дискет имеет свои преимущества, хотя загрузка с жесткого диска предпочтительнее, поскольку она позволяет избежать неудобств, связанных со сменой дискет. Однако в некоторых случаях загрузка с дискет более приемлема, например, при установке системы или при повреждении файловой системы.

После того как ядро системы загружено в память (с жесткого диска или с дискет) и запущено, выполняется определенная последовательность действий, описанная далее.

Поскольку ядро Linux находится в упакованном виде, то выполняется его распаковка. Это осуществляет небольшая программа, расположенная в самом начале программного кода ядра.

Если на компьютере установлена нестандартная видеокарта, выдается запрос для уточнения требуемого режима. При компиляции ядра можно сразу указать используемый режим, чтобы система не запрашивала его во время загрузки. Режим также может быть установлен при помощи LILO или утилиты `rdev`.

После этого ядро тестирует аппаратную часть компьютера (жесткие диски, дисководы, сетевые адаптеры) и конфигурирует соответствующие драйверы устройств. Во время данного процесса на экран выдаются сообщения как показано далее:

```
. . .
LILO boot:
Loading linux.
Console: colour EGA+ 80x25, 8 virtual consoles
Serial driver version 3.94 with no serial options enabled
tty00 at 0x03f8 (irq = 4) is a 16450
```

```
tty01 at 0x02f8 (irq = 3) is a 16450

lp_init: lp1 exists (0), using polling driver

Memory: 7332k/8192 available (300k kernel code, 384k re-
served, 176k data)

Floppy drive(s): fd0 is 1.44M, fd1 is 1.2M

Loopback device init

Warning WD8013 board not found at i/o = 280

Math coprocessor using irq13 error reporting

Partition check:

  hda: hda1 hda2 hda3

VFS: Mounted root (ext filesystem)

. . .
```

Тексты сообщений могут отличаться на разных системах и зависят от аппаратного обеспечения, версии Linux и конфигурации.

Далее ядро пытается смонтировать файловую систему для суперпользователя root. Место, куда она будет смонтирована, задается во время компиляции с помощью утилиты `rdev` или LILO. Тип файловой системы определяется автоматически. Если система root не монтируется, например, из-за того, что ядро не содержит драйвер соответствующей файловой системы, то система зависает.

Файловая система root обычно монтируется в режиме read-only, причем это устанавливается таким же образом, как и узел монтирования. Это делает возможным проверку файловой системы, когда она смонтирована, хотя проверка файловой системы, установленной в режиме read-write, не рекомендуется.

Затем ядро запускает программу `init` в фоновом режиме. Программа расположена в каталоге /sbin и именно она становится главным процессом. Процесс `init` выполняет различные функции, требуемые при установке системы.

Наконец, `init` запускает программу `getty` для виртуальных консолей и сериальных устройств. Эта программа позволяет подключаться к системе посредством виртуальных консолей и терминалов, подключенных через последовательные порты.

Программа `init` может быть сконфигурирована также для запуска и других программ.

После этого процесс запуска системы считается завершенным и система готова к работе.

При выключении системы Linux необходимо выполнить некоторые процедуры. Если этого не сделать, то файловые системы и файлы могут быть повреждены. Это происходит по причине наличия в Linux дискового кэша, информация из которого записывается на диск только через некоторые промежутки времени. Это значительно повышает производительность системы, но не позволяет просто выключить питание, поскольку в дисковом кэше может находиться большое количество информации, и файловая система может быть частично повреждена, т. к. на диск обычно сбрасывается только часть информации.

Другой причиной является мультизадачность системы, где одновременно способно выполняться несколько процессов, и выключение питания может быть гибельным для системы. Особенно это касается компьютеров, на которых одновременно работает несколько пользователей.

Существуют команды, предназначенные для правильного выключения системы. К ним относятся команды `shutdown` и `halt`, расположенные в каталоге /sbin. Есть два обычных способа их применения.

Если система установлена на компьютере, где работает один пользователь, то для правильного выключения выполняют такие действия:

1. Завершают работу всех программ.
2. Завершают работу всех виртуальных консолей.
3. Входят в систему как суперпользователь root (или остаются подключенными под этим пользователем).
4. Выполняется команда `halt` или `shutdown -h now` (при желании можно определить задержку, которая устанавливается заменой параметра `now` на знак `+` и число минут, по истечении которых будет завершена работа системы).

Если на компьютере, на котором установлена система, работает одновременно несколько пользователей, то возможно использование команды `shutdown` в следующем формате:

```
shutdown -h +время сообщение
```

Здесь *время* — это время, по истечении которого работа системы будет завершена, а *сообщение* — сообщение, в котором объясняется причина выключения, например:

```
root# shutdown -h +10 'We will install a new disk.  System
should be back on-line in three hours.'
```

Выполнение этой команды предупредит каждого пользователя, работающего в системе, что она будет выключена через 10 минут. Сообщение выдается на каждый терминал, где работают пользователи, включая xterm:

```
Broadcast message from root (ttyp0) Wed Aug 2 01:03:25
1995...


We will install a new disk. System should

be back on-line in three hours.

The system is going DOWN for system halt in 10 minutes !!
```

Выдача сообщения автоматически повторяется несколько раз перед прекращением работы системы, и каждый раз с более коротким интервалом. При использовании команды `halt` нельзя установить задержку, поэтому данная программа редко применяется в многопользовательских системах.

После запуска процесса прекращения работы системы демонтируются все файловые системы (кроме системы root), завершается выполнение всех процессов и программ-демонов, затем демонтируется файловая система root, и вся работа завершается. Далее выдается сообщение, в котором говорится, что можно отключить питание. Только после этого питание компьютера может быть отключено.

В некоторых случаях невозможно завершить этот процесс соответствующим образом. Например, при повреждении кода ядра в

памяти нарушается его работа или система зависает, и нет возможности ввести новую команду.

В этом случае нужно выключить питание, но после перезагрузки проверить операционную систему. Если же неполадки не такие серьезные (например, вышла из строя клавиатура), а ядро работает нормально, то наилучшим вариантом будет подождать несколько минут, пока программа `update` не сохранит на диске информацию, хранящуюся в кэш-буфере и только после этого выключить питание.

Иногда выключают компьютер после трехкратного выполнения команды `sync`, которая сбрасывает на диск содержимое буфера. Если в момент выключения работа всех программ была завершена, то эта процедура почти идентична выполнению команды `shutdown`. Однако файловые системы не демонтируются, что может привести к некоторым проблемам, связанным с флагом `clean filesystem` системы ext2fs. В любом случае использование этого способа не рекомендуется.

Процесс перезагрузки может выполняться путем прекращения работы системы, выключения питания и включения снова. Более простой способ — это выполнить команду `shutdown` с опцией `-r`, например:

```
shutdown -r now
```

Также можно использовать команду `reboot`.

Команда `shutdown` может также применяться для перевода системы в однопользовательский режим, в котором к системе никто не может подключиться кроме пользователя root, использующего для работы главную консоль. Иногда это применяется для административных целей, для выполнения которых не может быть использована нормально работающая система.

Не всегда имеется возможность загрузки системы с жесткого диска. Например, при неправильных установках в LILO-загрузчике систему невозможно будет загрузить. В таких случаях используется другой способ загрузки. Для персональных компьютеров обычно она выполняется с дискет.

Большинство распространяемых версий Linux позволяет во время установки системы создать загрузочную дискету. Однако многие такие дискеты содержат лишь ядро, и предполагается, что для устранения неполадок будут использоваться программы, находящиеся на установочных дисках. Иногда этих программ бывает недостаточно, например, когда требуется восстановить некоторые файлы, созданные с помощью программ, которых нет на этих дисках.

Поэтому может возникнуть необходимость в создании специально настроенного диска. В документации Linux содержится необходимая информация для создания подобного диска.

При загрузке со специально настроенного диска нельзя использовать привод, на котором смонтирована эта дискета, для каких-либо других целей. Это может создать некоторые неудобства, если в компьютере имеется только один дисковод. Однако если компьютер имеет достаточный объем памяти, можно загрузить этот диск в RAM-диск (для этого ядро, расположенное на дискете, должно быть сконфигурировано соответствующим образом). Это позволяет использовать дисковод для других целей.

Глава 6

Взаимодействие пользователя с операционной системой: командные интерпретаторы

Система UNIX, как и любая другая развитая современная операционная система, имеет огромное количество команд, позволяющих осуществлять множество операций с объектами системы (пользователи, файлы, устройства), управлять их поведением и настраивать систему по своему усмотрению (в определенных пределах, разумеется). В предыдущих главах мы уже использовали многие команды UNIX, предполагая, что все они вводятся из так называемой командной строки. Интуитивно этот процесс понятен: из приглашения командной строки вводится команда и на экране консоли отображается или не отображается результат ее выполнения. С их помощью можно выполнять большинство действий по настройке, диагностированию и управлению системой. Команды операционной системы являются "кирпичиками", из которых строятся любые, часто очень сложные и изощренные программы.

В этой главе мы рассмотрим более подробно, как выполняются команды операционной системы и как на их основе создавать программы, которые чаще всего называются *командными файлами* или *командными сценариями*. Если провести аналогию с опе-

рационной системой MS-DOS, то командные файлы там имеют расширение bat и состоят из команд, интерпретируемых командным процессором command.com определенным образом.

Команды UNIX не выполняются сами по себе, а только в контексте определенной командной оболочки, которую называют *интерпретатором команд*. Интерпретатор команд проверяет и анализирует введенные команды и их аргументы, анализирует их синтаксис, корректность введенных ключей и т. д. После успешного завершения проверки интерпретатор запускает соответствующую программу, т. е. создает в UNIX процесс и передает ему управление.

Командные интерпретаторы имеют общее название shell. Помимо исполнения команд, интерпретатор shell выполняет и другие операции:

- конвейеризацию команд;
- переназначение ввода/вывода;
- генерацию имен файлов;
- контроль среды окружения.

Значение интерпретатора для пользователей UNIX трудно переоценить. Командный интерпретатор позволяет выполнять не только отдельные команды, но и решать намного более сложные задачи, которые реализуются на основе командных файлов. В дальнейшем выражения "командный файл", "командный скрипт" и "командный сценарий" будут использоваться как синонимы и обозначать исполняемый файл, состоящий из команд в виде текстовых строк.

В настоящее время наиболее часто используются четыре основные разновидности shell:

- Bourne shell (sh) — является оригинальным командным интерпретатором и включается во все без исключения дистрибутивы операционной системы UNIX;
- C shell (csh) — разработан в Калифорнийском университете (г. Беркли). Особенностью этого интерпретатора является возможность интерактивной обработки shell-окружения;

- Korn shell (ksh) — разработан Дэвидом Корном и включает целый ряд дополнительных возможностей по сравнению с Bourne shell;
- Bourne Again shell (bash) — разработан Фондом свободно распространяемых программных продуктов (Free Software Foundation) и аккумулирует в себе возможности оболочек C shell и Korn shell, что обусловило его широкую популярность среди системных администраторов. Он является наиболее продвинутым интерпретатором по сравнению с остальными и служит очень мощным инструментом программирования задач системного администрирования. В этой главе мы будем подробно рассматривать именно возможности bash, хотя большинство командных файлов без каких-либо изменений будут работать и в других оболочках.

В дальнейшем мы будем использовать наряду с термином "интерпретатор" другое определение — "командная оболочка". Оба эти выражения являются синонимами и употребляются в одинаковом контексте.

Одна и та же операционная система позволяет работать сразу с несколькими командными оболочками и переходить от одной командной оболочки к другой. Возможна и одновременная работа пользователя в нескольких экранах со своими командными оболочками.

Любой пользователь, работающий в операционной системе UNIX, оперирует командами того интерпретатора, который для него установлен либо в файлах инициализации, или по умолчанию в файле /etc/passwd (рис. 6.1).

Как видно из рисунка, для пользователя user1 при входе в систему по умолчанию будет определен интерпретатор bash, в то время как для пользователя user2 интерпретатором по умолчанию будет ksh. Если нужно изменить тип командного интерпретатора, то следует модифицировать соответствующим образом файлы инициализации пользователей user1 и user2.

В практическом аспекте командный интерпретатор shell является языком программирования очень высокого уровня. Работу с ко-

мандным интерпретатором можно начинать после входа пользователя в систему, при этом оболочка отображает приглашение (prompt) к вводу команды в виде одного из символов $, #, > или др., за которым обычно следует пробел. Команды пользователя вводятся сразу же после этого пробела.

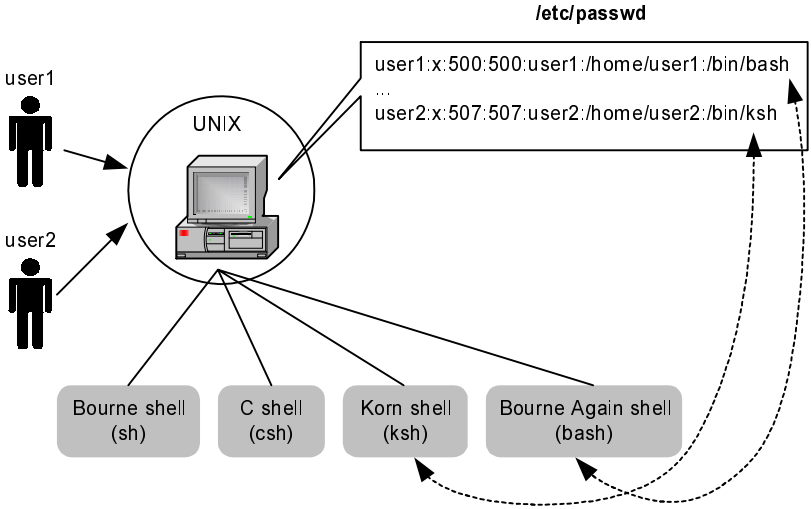


Рис. 6.1. Пример установок по умолчанию интерпретатора shell

Анализ особенностей командного интерпретатора проведем на примерах, разработанных в командном интерпретаторе bash. Исходные тексты командных файлов, представленных здесь, легко адаптируются для работы в любой из версий UNIX.

6.1. Элементы языка shell

Как и любой язык высокого уровня, командный интерпретатор shell имеет определенный синтаксис. В состав языка входит целый ряд операторов и символов, имеющих специальное значение.

В табл. 6.1 приводится полный перечень всех специальных символов, используемых в командных интерпретаторах.

Таблица 6.1. *Специальные символы командных интерпретаторов*

Символ	Функция
*	Используется в шаблонах поиска файлов
?	Используется в шаблонах поиска файлов
[ ]	Используется в шаблонах поиска файлов
&	Использование символа амперсанда после вводимой команды позволяет выполнить команду в фоновом режиме, освободив терминал для выполнения других программ
;	Точка с запятой служит разделителем команд, вводимых в одной строке
\	Обратный слэш отменяет назначение специальных символов *, ?, [ ], &, ;, >, < и |
' '	Используемый в паре прямой апостроф отменяет значение пробела как разделителя символов и назначение специальных символов
` `	Используемый в паре обратный апостроф замещает вывод команды
" "	Кавычки отменяют значение пробела как разделителя символов и назначение специальных символов, кроме $ и `
>	Переназначает вывод команды в файл, при этом предыдущее содержимое файла уничтожается
<	Символ переназначения ввода позволяет команде читать ввод с файла
>>	Позволяет добавить вывод команды в конец файла
|	Символ программного канала переназначает вывод одной команды на вход другой в конвейере команд
$	Символ доллара используется в позиционных параметрах *(см. далее)* и в переменных, определенных пользователем. Кроме этого, данный символ обозначает приглашение к вводу командного интерпретатора

Таблица 6.1 *(окончание)*

Символ	Функция
#	Указывает на то, что следующие за ним символы являются обычным текстом (комментарием)

Рассмотрим использование специальных символов на практических примерах. Следующая командная строка демонстрирует использование точки с запятой:

```
# who;ls -l
root     :0           Sep 29 09:32
root     pts/0        Sep 29 09:57 (:0.0)
total 34
drwxr-xr-x    8 root     root   4096  Oct 11 23:43 tmp
-rw-r--r--    1 root     root   20633 Sep  3 12:50 in-
stall.log
-rw-r--r--    1 root     root   20633 Sep  3 12:50 in-
stall.log.syslog
-rw-r--r--    1 root     root   141   Sep  2 14:13 textfile
```

В этом примере последовательно выполняются две команды — `who` и `ls -l`, первая из которых отображает на экране пользователей, работающих в системе, а вторая — список файлов и каталогов.

Последовательность символов `&&` (двойной амперсанд) читает код завершения первой команды, и если он равен 0 (успешное завершение), то будет выполнена вторая команда. Например:

```
# test -d ./tmp && echo tmp is a directory
tmp is a directory
```

При этом сообщение выводится только в том случае, если объект файловой системы /tmp является каталогом (в данном случае это так).

Если изменить условие и выполнить проверку объекта tmp как файла, то команда и результат будут другими:

```
# test -f ./tmp && echo tmp is a directory
```

Никакого вывода на экран консоли в этом случае мы не получим, поскольку результатом выполнения команды `test` является число, отличное от нуля (/tmp не является файлом).

Командная строка

```
# test -e install.log && echo File install.log exists
File install.log exists
```

выполняет проверку на существование файла install.log (опция `-e`) в текущем каталоге и, если это так, выводит соответствующее сообщение.

Следующая команда выводит на экран содержимое файла textfile только в том случае, если этот файл имеет установленный атрибут для чтения:

```
# test -r textfile && cat textfile
```

Результат, противоположный тому, который получается при использовании `&&`, дает оператор `||`. Если из двух команд, разделенных этим оператором, первая команда завершается с ненулевым результатом (т. е. неудачно), то выполняется вторая команда. Успешно выполненная первая команды (с нулевым кодом завершения) запретит выполнение второй. Например:

```
# test -f /.tmp || echo tmp is a directory
```

Здесь на экран консоли будет выведена строка

```
tmp is a directory
```

Аналогично, команда

```
# test -x textfile || cat textfile
```

отображает содержимое файла textfile, поскольку для этого файла не установлен атрибут исполнения, следовательно, результат выполнения команды `test -x` будет ненулевым.

Комбинация операторов `&&` и `||` позволяет создавать довольно замысловатые алгоритмы обработки, например:

```
# test -e textfile && test -r textfile && cat textfile
```

Эта команда выводит на экран содержимое файла textfile (`cat textfile`) только при одновременном выполнении двух условий:

- файл существует (`test -e`);
- файл имеет доступ по чтению (`test -r`).

Альтернативой приведенной может служить команда:

```
# test -e textfile && test -x textfile || cat textfile
```

Поскольку команда `test -e` завершается успешно, то будет выполнена следующая за ней команда `test -x`, которая завершается с ненулевым кодом, что разрешает выполнение команде `cat textfile`.

Следующая командная строка ничего на экран не выводит:

```
# test -e textfile && test -x textfile && cat textfile
```

В этой цепочке вторая команда `test -x` завершится неудачно, не позволяя выполниться команде `cat textfile`.

Создавая такие программные конструкции, следует помнить, что конечный результат зависит от результатов выполнения каждой из команд такой цепочки.

Если команда должна выполняться как фоновый процесс, то в ее конце нужно указать символ амперсанда `&`. Обычно запуск в фоновом режиме нужен для того, чтобы освободить консоль для запуска других программ, особенно если запускаемая программа выполняется достаточно длительное время.

Выполнение команды в фоновом режиме сопровождается выводом на экран идентификатора (PID) процесса, соответствующего выполняемой команде, при этом система, запустив фоновый процесс, вновь выходит на диалог с пользователем. Допускается запускать в фоновом режиме несколько команд, разделенных точкой с запятой.

Например:

```
# who;ls -l;pwd&
root     :0           Jan 17 09:32
root     pts/0        Jan 17 09:57 (:0.0)
[1]+  Done
total 56
-rw-r--r--    1 root     root   1544 Dec  1 12:54 install.log
-rw-r--r--    1 root     root   479  Jan 17 09:40 textfile
[1] 5211
```

После выполнения указанной цепочки команд в системе будет работать фоновый процесс с идентификатором 5211.

Вот еще один пример выполнения цепочки команд в фоновом режиме:

```
# test -e textfile && test -x textfile || cat textfile&
```

Очень мощным средством командного интерпретатора shell является переназначение ввода/вывода, расширяющее возможности стандартного ввода/вывода.

Стандартный ввод обозначается в операционной системе UNIX как stdin (standard input) и осуществляет операцию ввода данных с клавиатуры терминала, а стандартный вывод stdout (standard output) выполняет вывод данных на экран терминала. Кроме того, диагностические сообщения и сообщения об ошибках направляются в стандартное устройство ошибок, обозначаемое как stderr (standard error).

Операционная система UNIX обладает механизмами, позволяющими перенаправить результаты работы любой команды, предназначенные для стандартного ввода/вывода, на другое устройство или в файл.

Вначале рассмотрим операцию *переназначения вывода*. Поскольку любые устройства операционной системы UNIX являются файлами, то принято говорить о переназначении вывода в файл, для чего служит оператор >.

Если, например, нужно записать содержимое текущего каталога в файл с именем list_dir, то следует ввести команду

```
# ls -l > list_dir
```

выполняющую два действия:

- создание, если не существует, и открытие файла с именем list_dir в текущем каталоге процесса;
- запись выводимых командой `ls` данных в файл list_dir, при этом предыдущее содержимое файла (если он был непустой) затирается.

После выполнения команды файл list_dir будет содержать список файлов и каталогов, находящихся в текущем каталоге.

Переназначение вывода используется очень часто. Вот несколько типичных примеров применения этой операции:

- запись текстовых строк в файл;
- операции архивирования данных.

Для записи текста в файл используется команда `echo`, например:

```
# echo String to be written in file > textfile
```

Пример создания архива:

```
# find tmp -print | cpio -ovB > tmp_arch
```

Здесь команда `find` передает по программному каналу данные программе `cpio`, которая записывает их в архивный файл tmp_arch, используя переназначение вывода.

Еще один пример переназначения стандартного вывода:

```
# cpio -it -I tmp_arch > list_arch
```

Здесь команда `cpio` записывает список файлов, содержащихся в архиве tmp_arch, в файл list_arch.

К операторам переназначения вывода относится и `>>`, как и оператор `>`, он используется для записи данных в уже существующий файл, но при этом информация записывается в конец файла, не затирая предыдущую.

С помощью этого оператора можно добавить в файл textfile, созданный в одном из предыдущих примеров командой `echo`, следующую строку:

```
# echo String to be added to the file >> textfile
```

Замечу, что оператор переназначения `>>` можно использовать и как замену `>`.

Вывод на стандартное устройство ошибок также можно переназначить, но при этом следует использовать выражение `2>`. Рассмотрим пример.

Попытаемся выполнить команду `ls` с несуществующей опцией `-y`, перенаправив при этом вывод в файл ls.LOG:

```
# ls -y 2> ls.LOG
```

После выполнения команды файл ls.LOG может содержать примерно следующее:

```
# cat ls.LOG
ls: invalid option -- y
Try `ls --help` for more information.
```

Нужно всегда помнить, что если файл, в который переназначается вывод, уже существует, то его предыдущее содержимое теряется. Если нужно сохранить содержимое файла, лучше использовать оператор `>>`.

Программы, использующие для получения данных стандартный ввод, могут принимать их из файла через оператор *переназначения ввода* `<`.

Например, для подсчета количества строк, слов и символов в файле textfile можно применить команду

```
# wc < textfile
```

Операция переназначения ввода используется и в более сложных операциях, например, разархивирования данных:

```
# cpio -ivB < tmp_arch
```

Здесь команда `cpio` получает ввод из файла архива, созданного в одном из предыдущих примеров командой `find`, восстанавливая полный путь к файлу в процессе разархивирования.

Ввод и вывод одной и той же команды могут быть переназначены одновременно, как в этом примере:

```
# echo String > output
# cat < output > input
```

Команда `echo` записывает текстовую строку в файл output. В следующей строке команда `cat` получает данные из файла output и записывает их в файл input.

После выполнения этих двух команд файл input будет содержать строку `String`.

Чтобы лучше понять, в чем суть операций переназначения ввода/вывода, посмотрим, как реализован этот механизм в операционной системе UNIX. При запуске процесс получает дескрипторы стандартного ввода и вывода, позволяющие стандартным библиотечным функциям языков высокого уровня осуществлять консольный ввод/вывод. Например, хорошо известная библиотечная C-функция `printf()` использует дескриптор стандартного вывода, а функция `gets()` — дескриптор стандартного ввода.

В UNIX процесс может либо с помощью системного вызова `fcntl()`, либо с помощью библиотечной C-функции `dup2()` изменить свои порты ввода/вывода, переназначив их с консоли на текстовые файлы. Механизм переназначения ввода/вывода показан на рис. 6.2.

Проанализируем последовательность операций, показанных на рисунке. Предположим, что процесс должен переназначить направление ввода данных с консоли на файл с именем file. Для этого с помощью системного вызова `open()` процесс получает дескриптор `fdesc` файла file.

Затем полученный дескриптор дублируется при помощи системного вызова `fcntl()` — по завершению функции стандартный

дескриптор ввода, определенный макросом `STDIN_FILENO`, будет так же, как и `fdesc`, указывать на файл file. После этого можно закрыть дескриптор `fdesc` функцией `close()` — весь ввод процесс будет принимать из файла file, на который указывает дескриптор `STDIN_FILENO`.

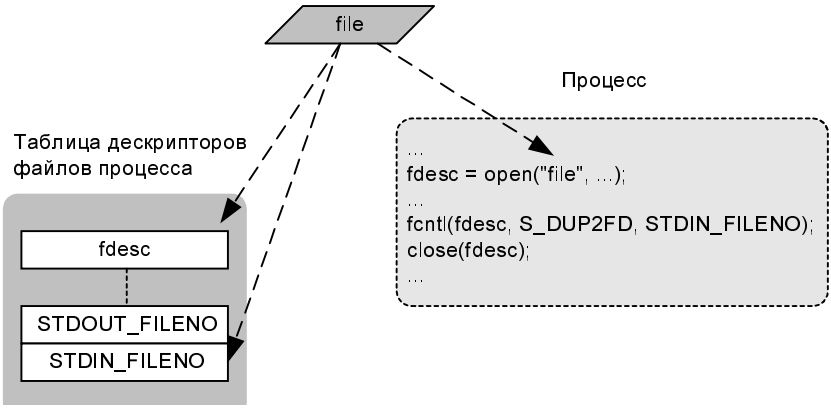


Рис. 6.2. Пример переназначения ввода/вывода в операционной системе UNIX

Точно так же можно переназначить вывод процесса в файл, только в этом случае в качестве третьего параметра функции `fcntl()` нужно указать дескриптор стандартного вывода `STDOUT_FILENO`. В качестве первого параметра должен выступать дескриптор файла, куда переназначается стандартный вывод.

В листинге 6.1 приведен исходный текст программы, принимающей данные из стандартного ввода, переназначенного на файл input и направляющего данные на стандартный вывод, который перенаправлен в файл OUTPUT.

Программа фактически эмулирует ситуацию, которую в терминах интерпретатора shell можно изобразить так:

```
процесс < input > output
```

Листинг 6.1. Перенаправление ввода/вывода с использованием функции `fcntl()`

```
#include <stdio.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

int main(void)
{
  int fin, fout;
  char buf[128];
  int bytesRead, bytesWritten;

  bzero(buf, sizeof(buf));

  fin = open("input", O_RDONLY);
  if (fin != -1)
    {
     fcntl(fin, F_DUP2FD, STDIN_FILENO);
     close(fin);
    }
  else
    return 0;

  bytesRead = read(STDIN_FILENO, buf, sizeof(buf));

  fout = open("OUTPUT", O_RDWR | O_CREAT | O_APPEND);
  if (fout != -1)
   {
    fcntl(fout, F_DUP2FD, STDOUT_FILENO);
```

```
    close(fout);
   }
  else
   {
    close(fin);
    return 0;
   }
  if (bytesRead > 0)
    printf("The OUTPUT_FILE content: %s\n", buf);
  return 0;
}
```

Здесь в качестве дескриптора файла, из которого данные читаются, выступает `fin`, а в качестве дескриптора, в который данные записываются, — `fout`. Оба дескриптора дублируются функцией `fcntl()`, после чего данные будут вводиться из стандартного дескриптора `STDIN_FILENO`, который теперь указывает на файл input. Выводимые данные направляются на стандартное устройство вывода, в качестве которого теперь будет выступать файл OUTPUT.

Вместо системного вызова `fcntl()` в программе можно использовать библиотечную С-функцию `dup2()` со следующими параметрами:

```
dup2 (fin, STDIN_FILENO);
dup2 (fout, STDOUT_FILENO);
```

Операции переназначения ввода/вывода командного интерпретатора shell реализованы, как можно догадаться, посредством функций `fcntl()` и `dup2()`.

Другими, очень мощными средствами командного интерпретатора shell являются фильтры и конвейеры команд (программные каналы, pipes). *Фильтрами* называются программы или команды, которые выполняют чтение со стандартного ввода и записывают результат в стандартный вывод. Многие команды операционной

системы UNIX реализованы в виде фильтров. Фильтры допускают объединение, используя механизм конвейеризации, позволяющий объединять стандартный вывод одной команды со стандартным вводом другой. Оператор, реализующий механизм программных каналов, обозначается вертикальной чертой |.

Командную строку, состоящую из нескольких команд (программ), работающих в *конвейере*, можно представить следующим образом:

программа1 | *программа2* | *программа3*...

Первая команда конвейера создает поток выходных данных, принимаемых на входе *программы2*. *Программа2*, в свою очередь, создает выходной поток для *программы3* и т. д.

Программный канал не следует смешивать с переназначением ввода/вывода, когда выходные данные просто перенаправляются в файл или входные данные читаются из файла — это разные механизмы.

Программный канал в операционной системе UNIX можно реализовать при помощи системных вызовов `pipe()`, `fcntl()` и `exec()`, что и сделано для часто используемых команд (`find`, `grep`, `cpio`, `ls` и т. д.).

Следующая программа на C, исходный текст которой показан далее, показывает технику конвейеризации. Для упрощения в листинге 6.2 показано взаимодействие родительского и порожденного процессов посредством программного канала, в котором используются стандартный ввод и стандартный вывод.

Листинг 6.2. Пример программного конвейера

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/types.h>
#include <fcntl.h>
```

```
int main(void)
{
  int fifo[2];
  char *str = "Test String to be redirected.";
  char buf[128];

  pid_t pid;
  bzero(buf, sizeof(buf));
  pipe(fifo);

  pid = fork();
  if (pid == 0)
   {
    close(fifo[1]);
    fcntl(fifo[0], F_DUP2FD, STDIN_FILENO);
    close(fifo[0]);
    while (gets(buf));
    printf("READ: %s\n", buf);
    exit(0);
   }
  if (pid > 0)
   {
    close(fifo[0]);
    fcntl(fifo[1], F_DUP2FD, STDOUT_FILENO);
    close(fifo[1]);
    printf("%s", str);
   }
 return 0;
}
```

Вначале запущенная программа создает неименованный канал при помощи системного вызова `pipe()`, затем порождается но-

вый процесс посредством системного вызова `fork()`. Родительский процесс связывает с каналом вывода (дескриптор `fifo[1]`) дескриптор стандартного вывода `STDOUT_FILENO`, после чего данные, направляемые на устройство стандартного вывода, будут выводиться в канал вывода. Дескриптор `fifo[1]` должен в этом случае быть закрыт (функция `close(fifo[1])`). После этого данные можно записывать в канал, используя функции вывода на консоль — в данном примере это `printf()`, которая выводит содержимое строки `str`.

Порожденный процесс, в свою очередь, связывает с каналом ввода дескриптор стандартного ввода `STDIN_FILENO`:

```
fcntl(fifo[0], F_DUP2FD, STDIN_FILENO);
```

После этого дескриптор `fifo[0]` следует закрыть — таким образом, вводимые со стандартного ввода данные будут направляться в канал ввода. Для ввода данных теперь можно использовать любую функцию, принимающую данные со стандартного ввода. В порожденном процессе мы будем использовать библиотечную функцию `gets()`.

В результате работы программы строка `str`, переданная из родительского процесса порожденному, будет выведена на экран дисплея.

Реализация программного канала для взаимодействия двух независимых процессов, естественно, будет сложнее, но принцип будет таким, как в этом демонстрационном примере.

Вернемся к функционированию программных каналов в интерпретаторе shell. Пусть требуется определить количество файлов, содержащихся в текущем каталоге. Если использовать переназначение ввода/вывода, то данная задача решается посредством двух команд:

```
# ls -l > list_files
# wc -l < list_files
```

Здесь используются операторы > и < переназначения ввода/вывода, а промежуточный результат хранится в файле list_files. Та

же операция с использованием программного канала решается одной командой:

```
# ls -l | wc -l
```

Нужно четко представлять себе, когда лучше использовать переназначение ввода/вывода, а когда конвейер команд. Оптимальное сочетание того и другого механизмов позволяет реализовать довольно сложные и эффективные алгоритмы обработки данных.

Так, например, если необходимо сохранить список файлов, имена которых начинаются с "text", то это можно сделать одной командой:

```
# ls -l| grep text > text_files
```

Командный интерпретатор shell, подобно языкам высокого уровня, использует переменные. Имя переменной должно начинаться с буквы или с символа подчеркивания, например:

```
string_1
_InputVal
```

Имена переменных наподобие

```
7n
my var
.var1
```

использовать нельзя. Имя `7n` начинается с цифры, `my var` содержит пробел, а `.var1` — недопустимый символ точки.

Переменным можно присвоить определенные значения с помощью оператора присваивания `=`:

```
string_1="String 1"
Var1=1
```

Здесь переменной `string_1` присвоена строка символов "`String 1`", а переменной `Var1` — значение 1. Обратите внимание на то, что строковое значение заключается в кавычки.

В отличие от языков программирования высокого уровня, таких, например, как C, переменные командного интерпретатора shell не связаны с определенным типом данных, поэтому любое присвоенное им значение интерпретируется как строка символов.

Для доступа к переменной нужно непосредственно перед ее именем установить знак доллара `$`. Следующий пример демонстрирует вывод содержимого переменной `myvar` на экран:

```
# myvar=10
# echo $myvar
10
```

Точно так же на экране отображается и строковое значение, присвоенное переменной:

```
# message="String"
# echo $message
String
```

Везде, где встречается символ `$`, предшествующий имени переменной, он заменяется ее значением, например:

```
# myvar=100
# echo myvar = $myvar
myvar = 100
```

Если имя переменной заключить в фигурные скобки, то отображаемое значение будет содержать символы (если есть), находящиеся за фигурными скобками, например:

```
# myvar="String"
# echo ${myvar}100
String100
```

Важное замечание: при записи операции присваивания переменная, оператор = и присваиваемое значение должны быть записаны без пробелов.

Переменной может быть присвоен результат выполнения команды, для чего используются обратные апострофы, например:

```
# DATE=`date`
```

Здесь переменная `DATE` получает результат вывода команды `date`.

Если команда заключена в обратные апострофы, то интерпретатор обязательно выполнит ее, поместив результат на то место в командной строке, где эта команда указана:

```
# users=`who | wc -l`
# echo $users users logged in.
3 users logged in.
```

Для работы со строками, файлами и каталогами в интерпретаторе shell очень часто используются специальные символы (метасимволы):

- `*` — произвольная последовательность символов, может не содержать ни одного символа;
- `?` — один произвольный символ;
- `[...]` — любой из символов в указанном диапазоне, либо указанный в перечислении.

Например, команда

```
# ls -l  r*
```

отображает все файлы каталога, начинающиеся с литеры r.

Команда

```
# ls -l *rt*
```

отображает на консоли все файлы, содержащие в имени rt.

Команда

```
# ls -l t.???
```

выводит файлы текущего каталога, начинающиеся с литеры t и имеющие 3-буквенные расширения, например, tx.doc и tx.out.

Команда

```
# ls -l [a-d]*
```

выводит на экран список файлов, начинающихся с a, b, c, d. Аналогичный результат можно получить с помощью одной из команд

```
# cat [abcd]*
# cat [bdac]*
```

Символ * можно использовать и самостоятельно. Например, команда

```
# echo *
```

отображает имена всех файлов текущего каталога на экране.

Кроме перечисленных метасимволов, командный интерпретатор shell использует еще несколько символов, имеющих специальное назначение:

- двойные кавычки ";
- апостроф ';
- символ обратной наклонной черты \.

Предположим, что текущий каталог содержит файлы data1, prog1 и text1. Введем две команды echo, аргументы которых содержат *, и сравним результаты выполнения:

```
# echo *
data1 prog1 text1
# echo  "*"
*
```

В первом случае интерпретатор shell воспринимает * как список содержимого текущего каталога и отображает список файлов, а во втором двойные кавычки отменяют значение символа *. В этом и заключается смысл использования двойных кавычек — любой символ, имеющий специальное значение (например, *, ?, >, >>, ||), утрачивает свой специальный статус.

Исключениями из этих случаев являются символ $, обратный апостроф (`) и обратный слэш (\), если они предшествуют специальному символу.

Смысл одиночных апострофов демонстрирует следующая команда:

```
# echo '$text'
$text
```

Как это следует из результата, если переменная заключена в апострофы, то ее значение не подставляется при вычислении выражения.

Разницу между двойными кавычками, апострофами и их отсутствием можно увидеть, выполнив следующие команды:

```
# echo *
data1 prog1 text1

# echo * "*" '*'
data1 prog1 text1 * *

# var1=test
echo $var1 "$var1" '$var1'
test test $var1
#
```

Обратная наклонная черта перед символом ликвидирует его специальное значение:

```
# echo \$text
$text
# echo \\
\
```

В командном интерпретаторе можно использовать так называемые предопределенные или, по-другому, системные переменные. Любой процесс выполняется в определенной среде (environment), представляющей собой в первом приближении набор некоторых стандартных переменных и констант, доступных данному процессу. Этот набор выделяется процессу операционной системой и может использоваться, например, для установки параметров по умолчанию для терминальных линий,

ссылок на определенные каталоги, содержащие команды операционной системы, и т. д.

Мы уже рассматривали системные переменные в *главе 4*, сейчас я кратко напомню смысл и назначение важнейших из них:

- `PATH` — указывает, в каких каталогах искать выполняемые файлы;
- `HOME` — это путь к домашнему каталогу пользователя;
- `MAIL` — имя файла электронной почты пользователя;
- `SHELL` — указывает оболочку, в которой работает пользователь.

Эти переменные можно использовать в выражениях и командах интерпретатора shell обычным образом, так же, как и другие переменные.

Большинство командных интерпретаторов в общем случае не предназначены для выполнения сложных математических вычислений, хотя имеют встроенные средства для арифметических операций. Приятным исключением является командная оболочка bash, в которой весьма существенно расширены возможности по обработке математических величин.

Например, командный интерпретатор выполняет приведенные далее операции следующим образом:

```
# n=0
# n=$n+1
# echo $n
0+1
```

Арифметические операции, которые можно выполнить в shell, мы рассмотрим далее при анализе командных файлов.

6.2. Командные файлы

Командный интерпретатор shell наиболее эффективен, когда используется для запуска так называемых *командных файлов* (командных сценариев, скриптов). Командные файлы пред-

ставляют собой обычные текстовые файлы, содержащие последовательность команд. В командных файлах, что очень существенно, можно использовать логические конструкции, подобные тем, что применяются в языках высокого уровня (`while`, `if`, `for`). Такие логические структуры позволяют реализовывать довольно сложные алгоритмы обработки данных и управления системой, которые невозможно реализовать из командной строки (хотя в командной строке и можно обеспечить некоторую гибкость выполнения за счет использования команд `test` и операторов `&&` и `||`).

Вот простой пример командного файла с именем dl, созданного с помощью редактора vi:

```
pwd
ls
echo This is the end of the shell script
```

Для запуска на выполнение файла dl введем строку

```
# sh dl
```

Если текущим каталогом является, например, /home/user1, то результат может выглядеть так:

```
# sh dl
/home/user1
file1
file2
file3
This is the end of the shell script
```

Текстовый файл можно сделать выполняемым (командным) при помощи команды `chmod`:

```
# chmod +x имя_файла
```

Здесь *имя_файла* — имя текстового файла, подлежащего выполнению. Данная команда устанавливает для файла атрибут исполнения (в символьной форме `+x`). Указывая имя файла, помните, что UNIX различает строчные и прописные литеры.

При создании командных файлов следует учитывать несколько важных моментов:

- не начинайте командный файл символом `#`, если планируется выполнение этого файла в командном интерпретаторе csh (C shell);
- ни в коем случае не присваивайте командному файлу имя, совпадающее с именем одной из системных команд, таких, например, как `ls`, `rm` или `mv`, иначе вместо командного файла UNIX выполнит системную команду. Объясняется это просто — система ищет выполняемую команду в каталогах, определяемых системной переменной `PATH`, и только потом в каталоге пользователя.

В командных файлах широко используются так называемые позиционные параметры, задающие аргументы командной строки для выполняемого файла. *Позиционный параметр* — это число, перед которым расположен знак доллара, например, `$1`, `$2`, `$3` и т. д. При запуске командного файла параметр `$1` соответствует первому по порядку аргументу после имени командного файла, параметр `$2` — второму и т. д. Допускается использовать до девяти позиционных параметров.

Командный файл (предположим, он называется test_parms), содержащий операторы:

```
echo 1-st parameter = $1
echo 2-nd parameter = $2
echo 3-d parameter = $3
```

после запуска с параметрами 1, 2, 3 выводит на консоль строки:

```
# ./test_parms 1 2 3
1-st parameter = 1
2-nd parameter = 2
3-d parameter = 3
```

Позиционные параметры позволяют задавать параметры командной строки в командных файлах, включая применение в

конвейере команд. Приведем некоторые примеры использования позиционных параметров.

Пусть командный файл содержит строку

```
ls -l | grep $1
```

Предположим, что файл назван list_file, тогда его выполнение с параметром `text` может дать, например, такой результат (если файл text существует):

```
# ./list_file text
-rw-rw-r--    1 root     root   19 Sep  8 12:18 text
```

Командная строка может содержать до 128 аргументов, однако командный файл может обрабатывать одновременно только девять параметров.

Во многих случаях требуется вводить данные с клавиатуры, присваивая их значения переменным. Это обеспечивает команда `read`, которой часто предшествует команда `echo`, отображающая на экране приглашение к вводу, например:

```
echo "Enter Integer:"
read x
```

Данный фрагмент командного файла после вывода на экран сообщения

```
Enter integer:
```

ожидает ввода значения с клавиатуры.

Одна команда `read` позволяет присвоить значения сразу нескольким переменным, и если параметров команды `read` больше, чем их было введено, то оставшимся переменным присваивается пустая строка. Если количество введенных значений больше, чем параметров в команде `read`, то лишние значения игнорируются.

Следующий пример демонстрирует использование команды `read`:

```
echo Enter string:
read x y
echo You entered: $x + $y
```

Один из возможных результатов работы командного файла показан далее:

```
Enter string:
37 51
You entered: 37 + 51
```

Командный интерпретатор shell обрабатывает и два так называемых специальных параметра — `$#` и `$*`. Параметр `$#` указывает на общее количество параметров выполняемого командного файла. Если сохранить строку

```
echo The number of arguments is $#
```

в файле arg_num и выполнить его, то получим, например, такой результат:

```
# arg_num param1 param2 param3
The number of arguments is 3
```

Параметр `$*` представляет собой строку, содержащую все аргументы командного файла. Если сохраним строку

```
echo The parameters are: $*
```

в файле arg_content и запустим его с параметрами `Three arguments here`, то получим такой результат:

```
# arg_content Three arguments here
The parameters for this command are: Three arguments here
```

Командные файлы, в общем случае, не так часто используют какие-либо математические вычисления. Тем не менее, для обработки целых чисел (если такое необходимо) можно воспользоваться командой `expr`. Использование этой команды лучше всего показать на нескольких примерах.

Первый пример демонстрирует вывод разности двух целых чисел на дисплей:

```
echo Enter two integers
read x y
echo The difference = `expr $x - $y`
```

Обратите внимание на запись выражения справа от команды `expr` — чтобы получить правильный результат, необходим пробел между переменными и знаком операции.

Команда `expr`, кроме сложения и вычитания, позволяет выполнять умножение и деление чисел. В следующем примере командного файла находится произведение двух переменных, значения которых введены с клавиатуры:

```
echo Enter two numbers:
read x y
echo 1-st num: $x
echo 2-nd num: $y
echo Multiplying $x and $y = `expr $x '*' $y`
```

Обратите внимание на оператор

```
`expr $x '*' $y`
```

Здесь для выполнения операции умножения символ `*` следует взять в апострофы, иначе командный интерпретатор выдаст ошибку.

Операцию деления двух целых чисел можно выполнить, если в рассмотренном выше примере заменить символ `*` символом `/`.

Вот пример более сложных вычислений:

```
echo Enter two numbers:
read x y
echo First number: $x
echo Second number: $y
echo $x '*' $y + $x '/' $y = `expr $x '*' $y + $x '/' $y`
```

Если сохранить указанные команды в файле с именем multi_ops и запустить его на выполнение, то можно получить результат такого вида:

```
# ./multi_ops
Enter two numbers:
5 3
```

```
First number: 5
Second number: 3
5 * 3 + 5 / 3 = 16
```

Нередко требуется передавать значения переменных, используемых в одном процессе, другому процессу. Все переменные какого-либо процесса по умолчанию являются локальными для него и недоступными остальным процессам.

Командный интерпретатор shell имеет механизм, позволяющий данные одного процесса передавать другому — механизм экспортирования, реализованный при помощи команды `export`, которая имеет такой синтаксис:

```
# export список_переменных
```

Здесь *список_переменных* представляет собой список переменных, разделенных пробелами, причем символ `$` перед именами переменных не ставится.

Любой процесс, который выполняется после этой команды, получает доступ к экспортированным переменным, но изменить их значений он не может. Напомню, что с экспортированием переменных окружения мы уже сталкивались в *главе 4* во время анализа рабочей среды пользователя.

Запишем в командный файл строку:

```
echo $X
```

и сохраним его под именем demo_exp.

Этот командный файл при выполнении ничего не выводит на экран, поскольку команда `echo` ничего "не знает" о переменной `X`.

Присвоим переменной `X` значение 777 и экспортируем ее:

```
# x=777
# export x
```

Повторно запущенный командный файл demo_exp теперь выведет на экран значение 777.

Механизм экспортирования обладает одной весьма полезной особенностью — если присвоить переменной `X` другое значение, то это значение будет доступно другим процессам без повторного выполнения команды `export`.

6.3. Логические структуры командного интерпретатора

До сих пор мы рассматривали командные файлы, в которых команды выполнялись последовательно. В подавляющем большинстве случаев так не бывает — почти все программы (и командные файлы в этом плане не являются исключением) реализуют алгоритмы с ветвлениями и переходами. Во всех языках высокого уровня, включая командный интерпретатор shell, есть программные конструкции, позволяющие изменять последовательность выполнения команд в зависимости от тех или иных условий. Их часто называют логическими структурами, и благодаря им язык командного интерпретатора приобретает особую гибкость и мощь.

Большинство логических конструкций shell напоминает аналогичные им логические структуры, имеющиеся в таких языках высокого уровня, например, как C или Fortran. Практически все логические структуры командного интерпретатора используют коды завершения команд для определения дальнейшей последовательности выполнения операторов.

Код завершения показывает, с каким результатом выполнилась команда: в случае удачного завершения команда возвращает 0, а в случае неудачи — значение, отличное от нуля. Код завершения команды по умолчанию не выводится на экран — чтобы его увидеть, нужно задать специальный параметр `$?`, например:

```
# pwd
/root
# echo $?
0
```

```
# cat file1
cat: file1: No such file or directory
# echo $?
1
#
```

Как видно из примера, первая команда (`pwd`) выполнилась без ошибок, поэтому вернула значение кода ошибки, равное 0, в то время как вторая команда завершилась неудачно, поэтому код ошибки равен 1.

Следует сказать, что почти все логические структуры предназначены специально для использования в командных файлах и не работают непосредственно в командной строке, за исключением команды `test`, которую мы рассматривали в начале главы.

Анализ логических структур командного интерпретатора shell начнем с оператора цикла `for`. В общем виде цикл `for` можно представить так:

```
for переменная in список_переменных
do
  команда_1
  команда_2
  . . .
  команда_n
done
```

В этой конструкции жирным шрифтом выделены ключевые слова, которые обязательно должны присутствовать в цикле `for`. Цикл `for` последовательно выполняет команды, находящиеся между ключевыми словами `do` и `done`.

При этом выполняется последовательный проход по списку *`список_переменных`* и очередное выбираемое значение из этого списка присваивается переменной *`переменная`*, имя которой указано слева от ключевого слова `in`. Список переменных может задаваться как символьными константами, так и дискретными значениями, разделенными пробелами.

Следующий фрагмент программного кода демонстрирует свойства цикла `for`:

```
for x in  100 1000 10000
do
 echo "$x"
done
echo
```

Результатом выполнения такого цикла будет такая последовательность чисел на экране:

```
100
1000
10000
```

Следующий фрагмент программного кода выводит на экран список файлов текущего каталога, размер которых отличен от нуля:

```
for x in *
do
  test -s $x && ls -l $x
done
```

Переместить файлы из каталогов, указанных в списке переменных, в другой каталог можно с помощью следующего программного кода:

```
echo enter the directory path
read path
test -e $path || mkdir $path
for file in dir1 dir2 dir3
 do
   mv $file $path/$file
done
```

Команда `test -e $path || mkdir $path` создает каталог с именем, заданным переменной `path`, если таковой отсутствует.

Далее мы рассмотрим логическую структуру высокого уровня, имеющую название оператор условия `if`.

Структуру оператора `if` в общем виде можно представить так:

```
if условие
then
   команда_1
   команда_2
   . . .
   команда_n
fi
```

Здесь *условие* представляет собой определенное выражение, от истинности которого зависит, будут ли выполняться операторы, расположенные между ключевыми словами `then` и `fi`.

Если условие является истинным, то команды будут выполняться, а если ложным — они будут пропущены. Условие представляет собой определенное выражение, заключенное в квадратные скобки, например,

```
[ $I -gt 100 -a $I -le 1000 ]
```

Это выражение будет истинно, если переменная `I` больше 100 и меньше или равна 1000.

Рассмотрим несколько примеров применения оператора `if`. Следующий фрагмент программного кода выполняет поиск слова в текстовом файле:

```
echo Enter the word and filename
read word file
if grep $word $file
  then
    echo $word found in $file
fi
```

Выполняющийся процесс ожидает ввода с консоли слова, которое следует искать, и имени файла, после чего анализирует содержимое файла. При обнаружении слова на экран выводится соответствующее сообщение.

Логическая структура `if` имеет еще одну форму, в которой используется оператор `else`:

```
if условие
then
   команда_1
   команда_2
   . . .
else
   команда_3
   команда_4
   . . .
fi
```

Предыдущий пример можно модифицировать таким образом, чтобы в нем использовался оператор `if...else`:

```
echo Enter the word and filename
read word file
if grep $word $file >/dev/null
  then
    echo $word found in $file
  else
    echo $word is NOT found in $file
fi
```

Если слово обнаружено, то выводится сообщение

```
echo $word found in $file
```

иначе сообщение

```
echo $word is NOT found in $file
```

В этом примере используется устройство /dev/null, являющееся приемником ненужной или нежелательной информации, которая может появляться в результате выполнения команд — переназначение вывода в /dev/null применяется для "удаления мусора".

Альтернативой оператору `if` в ряде случаев может служить команда `test`, с которой мы уже сталкивались ранее. Как и `if`, данная команда проверяет истинность определенных условий и полезна при организации ветвлений в программе. Во многих случаях команда `test` применяется вместе с оператором `if`.

Команда `test` может принимать целый ряд опций:

- `-r` *файл* — условие истинно, если файл существует и доступен для чтения;
- `-w` *файл* — условие истинно, если файл существует и доступен для записи;
- `-x` *файл* — условие истинно, если файл существует и доступен для выполнения;
- `-s` *файл* — условие истинно, если файл существует и его размер больше нуля;
- *переменная_1* `-eq` *переменная_2* — истинно, если *переменная_1* равна *переменная_2*;
- *переменная_1* `-ne` *переменная_2* — истинно, если *переменная_1* не равна *переменная_2*.

Следующий фрагмент программного кода, в котором используется как оператор `if`, так и команда `test`, отображает на экране дисплея содержимое текстового файла, имя которого будет задано в командной строке:

```
if test -f $1
   then cat $1
fi
```

Следующий тип логических структур, который мы рассмотрим, — операторы цикла `while` и `until`.

Оператор `while` можно представить следующим образом:

```
while условие
do
  команда_1
  команда_2
  . . .
  команда_n
done
```

Оператор `while` повторяет заданную группу команд, если условие выполнения соответствует истине, при этом условие проверяется перед выполнением самой первой команды из списка, поэтому вполне возможно, что список не будет выполнен ни разу.

Рассмотрим пример использования оператора `while`. Предположим, что необходимо записать символьные данные в файл с именем textfile. Для окончания записи данных нужно нажать комбинацию клавиш <Ctrl>+<D>, после чего содержимое файла будет отображено на дисплее. Исходный текст командного файла показан далее:

```
echo Please enter one or more words and press Enter
echo Please end the list of words by pressing Control-D
while read x
do
 echo $x >> textfile
done
echo textfile contains the following words:
cat textfile
```

Что же касается оператора `until`, то он представляет собой разновидность оператора `while`. Основное отличие состоит в том, что в `while` цикл выполняется, пока условие является истинным, в то время как в `until` повторение цикла осуществляется до тех пор, пока условие ложно.

Далее приводится пример использования оператора `until`. Следующий фрагмент программного кода каждые 300 секунд проверяет присутствие на диске файла newtmp и, когда он создан, делает его архивную копию с именем newtmp.arch, удаляя оригинал:

```
until test -r newtmp
do
  sleep 300
  echo Waiting until file is created...
done
echo File created. Making archive...
echo newtmp|cpio -ovB > newtmp.arch
rm newtmp
```

Фрагмент программного кода, показанный далее, может помочь пользователю очистить каталог от файлов нулевого размера:

```
for x in *
do until test -s "$x"
  do
    rm $x
    echo File $x is deleted
    break
 done
done
```

В этом фрагменте оператор `for` обеспечивает проход по всем файлам каталога, а команда `test` с опцией `-s` проверяет существование файла и его размер. Если файл имеет нулевой размер, команда `test -s` устанавливает код завершения, равный 1, что приводит к выполнению команды удаления файла `rm $x`. Далее цикл повторяется, пока не будут проверены все файлы каталога.

Последний оператор, который мы рассмотрим и который позволяет создавать программы с множественными ветвлениями, — это оператор `case`. Он удобен для организации ветвления про-

граммы и работает по принципу совпадения шаблонов. Вообще, оператор `case` можно заменить группой операторов `if...else`, однако во многих случаях использование `case` удобнее.

Оператор `case` имеет следующий формат:

```
case переменная
in
    шаблон_1) команда_11
              команда_12
              . . .
              команда_1N;;
    шаблон_2) команда_21
              команда_22
              . . .
              команда_2N;;
    шаблон_N) команда_N1
              команда_N2
              . . .
              команда_NN;;
esac
```

Здесь *переменная* сравнивается со всеми шаблонами, причем первое же совпадение приводит к выполнению соответствующей группы команд из списка. Образец *шаблон_N* может включать обычные метасимволы командной оболочки shell. Рассмотрим пример командного файла, исходный текст которого показан далее:

```
echo Enter:
echo 1 - show current directory
echo 2 - show content
while read i
 do
 case $i in
```

```
   1)
      pwd
      continue;;
   2)
      ls -l
      continue;;
   *) break;;
 esac
done
```

Данный командный файл позволяет, в зависимости от выбранной опции, либо просмотреть текущий каталог, либо содержимое этого каталога. Здесь нам встретились две команды — `break` и `continue`. Команда `break` выполняет безусловный выход из тела цикла, а `continue` досрочно завершает текущую итерацию цикла `while` или `for` и переходит к следующей.

Последний пример — самый сложный. Фрагмент программного кода, представленный далее, позволяет выполнять операции резервного копирования и восстановления файлов.

В качестве исходного каталога выбран $HOME/user, а в качестве каталога, где создается копия — backup.

В процессе архивирования выполняется копирование всего дерева каталогов, начиная с текущего:

```
echo "Select choice:"
echo "1 - backup files"
echo "2 - restore files"
echo "q - exit"
while read x
do
 case $x in
  1)cd $HOME
    rm -r backup
```

```
    cd developer
    find . -print -depth|cpio -pdm $HOME/backup
    continue;;
  2)cd $HOME
    test -d backup && cd backup && find . -print -depth|cpio
-pdm $HOME/user
    continue;;
  q)break;;
 esac
done
echo Done.
```

Исходный текст данного командного файла можно использовать в качестве шаблона для разработки собственной программы резервного копирования.

Глава 7

Сетевые настройки UNIX

Изначально UNIX разрабатывалась как сетевая операционная система, и именно в этом и заключается вся ее мощь. Анализ сетевых возможностей UNIX неосуществим без знания основ сетевого взаимодействия, поэтому вначале рассмотрим некоторые наиболее существенные аспекты этого вопроса.

Вначале уточним сам термин "сеть". В протоколах сетевого уровня термин *"сеть"* означает совокупность компьютеров, соединенных между собой в соответствии с одной из стандартных типовых топологий и использующих для передачи пакетов общую базовую сетевую технологию, причем внутри сети сегменты не разделяются устройствами маршрутизации.

В обычном смысле под сетью понимают комплекс аппаратно-программных средств, позволяющих обмениваться информацией между компьютерными системами, физически расположенными в разных местах. Именно на это определение сети мы и будем опираться при анализе сетевого взаимодействия. Аппаратно-программные средства сетевого взаимодействия, вообще говоря, могут быть реализованы и для взаимодействия процессов, выполняющихся на одной машине, но эти случаи рассматриваться не будут.

Сеть, в общем случае, кроме компьютеров, включает целый ряд дополнительных устройств, обеспечивающих ее функционирование (маршрутизаторы, коммутаторы и т. д.) (рис. 7.1).

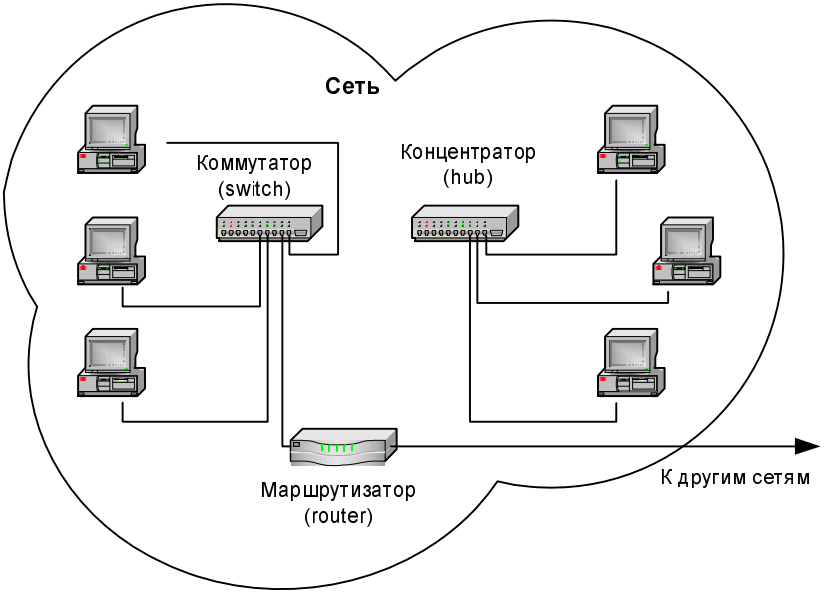


Рис. 7.1. Пример сети

Данные между компьютерами сети передаются посредством так называемых *пакетов*, а их доставка нужному адресату и оптимизация трафика обеспечиваются посредством сетевого оборудования. Очень часто сетевые устройства, независимо от их физической природы, называют *узлами сети*, а для компьютеров, входящих в состав сети, используется термин *"хост"*. Иногда под термином хост подразумевают компьютерную систему, выполняющую функции обслуживания запросов сетевых клиентов.
В любом случае путаницы между использованием этих терминов обычно не возникает.

Функционально сети разделяются на локальные (Local Area Network, LAN) и глобальные (Wide-Area Networks, WAN). Локальные сети обычно используются в организациях, коммерческих структурах и предприятиях, географически расположенных в пределах определенного региона, с небольшими расстояниями между узлами сети. Глобальные сети используются обычно в

разветвленной инфраструктуре, которая включает, например, разные подразделения одной и той же организации, расположенные в географически удаленных регионах. Как правило, глобальная сеть обеспечивает передачу данных между несколькими удаленными локальными сетями.

Собственно, основное отличие LAN от WAN состоит в том, что для передачи данных между удаленными узлами сетей требуются иные программные средства и иное оборудование, чем то, которое используется в локальных сетях. Наиболее ярким примером глобальной сети является знакомый всем Интернет.

7.1. Топология сетей

Любая сеть имеет определенную топологию. Под *топологией* (компоновкой, конфигурацией, структурой) *компьютерной сети* обычно понимается физическое расположение компьютеров сети друг относительно друга и способ соединения их линиями связи. Важно отметить, что понятие топологии относится, прежде всего, к локальным сетям, в которых структуру связей можно легко проследить. В глобальных сетях структура связей обычно скрыта от пользователей и не слишком важна, т. к. каждый сеанс связи может производиться по собственному пути.

Топология определяет фундаментальные характеристики сети:

- требования к оборудованию и тип используемых физических соединений;
- допустимые и наиболее подходящие механизмы управления обменом;
- надежность работы;
- возможности дальнейшего расширения сети.

Пользователям операционных систем, в том числе и UNIX, вряд ли нужны глубокие знания топологии сетей, но представлять особенности тех или иных основных топологий все же нужно, поскольку они существенно влияют на эффективность сетевого обмена данными. Знание топологии сети помогает правильно

выбрать аппаратно-программные настройки сетевых интерфейсов UNIX-системы.

Существуют три базовые топологии сети: общая шина, звезда и кольцо.

Общая шина — все хосты подключаются параллельно к общей для всех линии связи, при этом данные от каждого компьютера одновременно передаются всем остальным компьютерам (рис. 7.2).

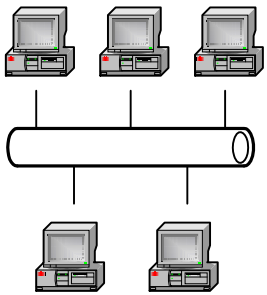


Рис. 7.2. Топология "общая шина"

Топология "общая шина" обеспечивает также одинаковые права доступа всех абонентов к сети. Передача данных в такой топологии осуществляется по очереди, поскольку линия связи в данном случае является единственной для всех компьютеров. Если несколько компьютеров передают информацию одновременно, то могут возникать так называемые *коллизии* (конфликты передачи данных). Общая шина всегда реализует режим так называемого полудуплексного обмена, когда данные передаются в обоих направлениях не одновременно, а по очереди.

Надежность сети при такой топологии высока, поскольку отсутствует центральный хост или сетевое устройство, посредством которого передается вся информация — при отказе одного из хостов сеть продолжает функционировать, в отличие от централизованного варианта, в котором при отказе центрального хоста или сетевого устройства перестает функционировать вся управ-

ляемая им система. Подключение новых абонентов к шине несложно и может быть выполнено при работающей сети.

Топология "звезда" — к одному центральному хосту или сетевому устройству подключаются остальные компьютеры, входящие в сеть, причем каждый из них использует отдельную линию связи (рис. 7.3).

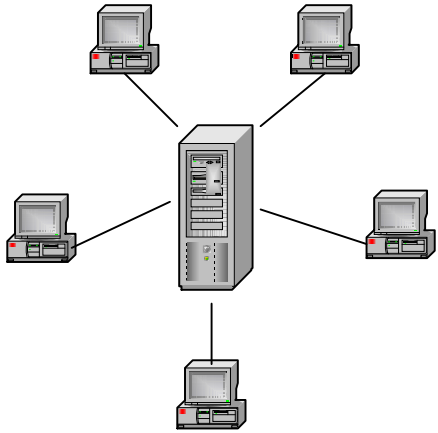


Рис. 7.3. Топология "звезда"

При этой топологии информация от периферийного хоста передается только центральному хосту, а от центрального — одному или нескольким периферийным. Здесь в качестве центрального хоста может выступать как сервер UNIX с соответствующим программным обеспечением, так и сетевое устройство, реализующее эту топологию.

Поскольку обмен информацией осуществляется исключительно через центральный хост, на него ложится большая нагрузка. На центральном хосте должно быть установлено специальное программное обеспечение, выполняющее передачу пакетов между компьютерами такой сети. В сети с такой топологией конфликты в принципе невозможны, поскольку управление полностью централизовано.

Топология "звезда" гарантирует работу сети при выходе из строя периферийного компьютера или его сетевого оборудования — это никак не отражается на функционировании оставшейся части сети. В то же время любой отказ центрального компьютера делает сеть полностью неработоспособной.

Если сравнивать с общей шиной, то, в отличие от нее, в звезде на каждой линии связи находятся только два хоста: центральный и один из периферийных. Для их соединения используются две линии связи, каждая из которых передает информацию в одном направлении — на каждой линии связи имеется только один приемник и один передатчик (point-to-point, передача "точка — точка").

К серьезному достоинству топологии "звезда" следует отнести то, что все точки подключения собраны в одном месте — это позволяет легко управлять функционированием сети, а в случае возникновения неисправностей быстро их локализовывать путем простого отключения от центрального хоста абонентов сети (это невозможно в случае шинной топологии).

Кольцо — это топология, в которой каждый компьютер соединен линиями связи с двумя другими: от одного он получает информацию, а другому передает. На каждой линии связи, как и в случае звезды, работает только один передатчик и один приемник (связь типа "точка — точка"). Передача информации в кольце всегда производится только в одном направлении. Каждый из компьютеров передает информацию только компьютеру, следующему за ним в цепочке, а получает информацию только от предыдущего в цепочке компьютера (рис. 7.4).

В кольцевой топологии нет центрального хоста, и все компьютеры являются одинаковыми и равноправными. Тем не менее, довольно часто в кольце выделяется специальный хост, управляющий обменом. Наличие такого хоста снижает надежность сети, поскольку выход его из строя делает сеть неработоспособной.

Вообще говоря, хосты кольца не являются полностью равноправными, поскольку один из них обязательно получает инфор-

мацию от компьютера, передающего информацию в данный момент раньше, чем другие.

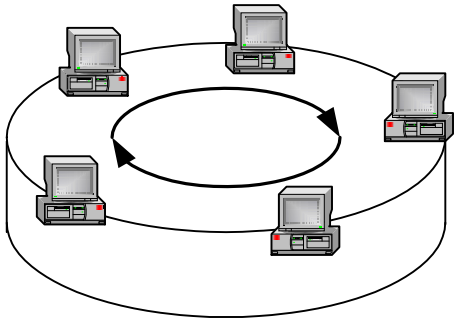


Рис. 7.4. Топология "кольцо"

Эта особенность топологии кольца используется для организации управления обменом по сети — право на следующую передачу (или, по-другому, на захват сети) переходит последовательно к следующему по кольцу компьютеру. Подключить нового абонента в кольцо несложно, хотя при этом необходимо остановить работу всей сети на время подключения.

Кольцевая топология обычно обладает высокой устойчивостью к перегрузкам, обеспечивая надежную передачу больших потоков передаваемой по сети информации при, практически, отсутствии конфликтов (в отличие от топологии "общая шина"). В этой топологии не требуется центральный хост, который, например, в топологии "звезда" может быть перегружен большими потоками информации.

В современных сетях ни одна из топологий не используется в чистом виде. Чаще всего применяется модифицированный вариант звезды, например, такой, как на рис. 7.5.

На этом рисунке показана простая сеть, в которой используется модифицированный вариант звезды. В качестве центрального узла выступает коммутатор, управляющий потоком данных по всей сети. Отдельные концентраторы обеспечивают передачу

данных на локальные компьютеры. Серверы сети реализуют программный интерфейс высокого уровня, обеспечивая управление потоком данных в сети на уровне операционных систем.

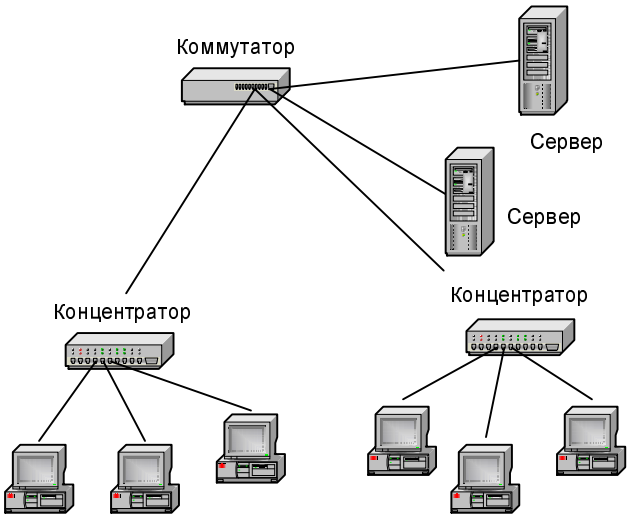


Рис. 7.5. Пример топологии сети

Топология сети, как мы видим, определяет способы физической передачи данных между узлами сети, но не указывает формат самих передаваемых данных. Все компьютеры, взаимодействующие по сети, должны оперировать одним и тем же форматом данных. Форматы данных и способы их формирования определяются так называемыми *сетевыми протоколами передачи данных*. В настоящее время используется много различных протоколов передачи данных, но наибольшее распространение получила группа (стек) протоколов, известных под названием TCP/IP. Стек протоколов TCP/IP используется в глобальной сети Интернет и является основой для построения подавляющего большинства локальных и глобальных сетей во всем мире, хотя существует и используется и целый ряд других протоколов передачи данных по сети.

Для передачи данных по сети от одного компьютера к другому передающая сторона должна знать аппаратный адрес принимающего хоста. Аппаратный адрес хоста представляет собой аппаратный адрес сетевой карты (адаптера), посредством которой осуществляется подключение данного компьютера к сети. Аппаратный адрес представляет собой уникальное 48-разрядное двоичное число, присваиваемое сетевому адаптеру производителем — его часто называют MAC-адресом (Media Access Control address). MAC-адрес обычно записывают в побайтовой форме, например, 00:0A:5E:40:E3:2C. Хочу особо акцентировать внимание на том, что конечной точкой передачи данных сети является MAC-адрес получателя.

7.2. Модели сетевого взаимодействия

Чтобы лучше понять, как осуществляется процесс передачи данных от одного компьютера в сети к другому, нам придется рассмотреть модель передачи данных OSI (Open System Interconnection, взаимодействие открытых систем). Это международная программа стандартизации обмена данными между компьютерными системами различных производителей на основе семиуровневой модели протоколов передачи данных в открытых системах, предложенная ISO.

7.2.1. Модель OSI

Модель OSI отображает все этапы передачи данных, представляя их в виде уровней. Для того чтобы понять принцип построения такой модели, представим процесс передачи данных на чисто интуитивном уровне (рис. 7.6).

Здесь приложения пользователя, передающие и/или принимающие данные по сети, осуществляют взаимодействие с операционной системой, которая, в свою очередь, формирует последовательности блоков данных, называемые *сетевыми пакетами*. Эти данные передаются посредством сетевой карты далее в сеть, где они принимаются адресатом.

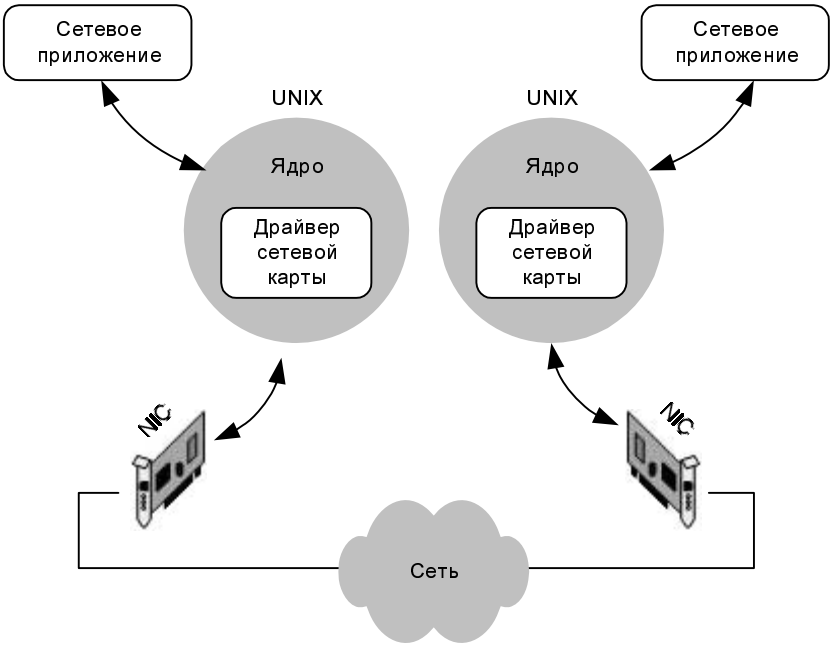


Рис. 7.6. Упрощенная схема передачи данных по сети

Таким образом, данные, прежде чем будут отправлены по сети, проходят несколько этапов преобразований. Цель модели OSI — формализовать сложный процесс формирования и передачи данных по сети, что позволяет использовать четкие критерии при разработке сетевых систем, а также обеспечить стандарт сетевой связи компьютеров независимо от производителя оборудования компьютеров и/или сети.

Процесс передачи данных по сети обычно представляют в виде семиуровневой модели ISO/OSI, которая разделяет сетевые функции на семь областей или уровней (рис. 7.7).

Все функции физического, канального и сетевого уровней тесно связаны с используемым в данной сети оборудованием: сетевыми адаптерами, концентраторами, мостами, коммутаторами и маршрутизаторами. Функции прикладного (уровень приложений), сеансового (уровень сессий) и уровня представлений реализуются операционными системами и системными приложениями конеч-

ных узлов. При этом транспортный уровень выступает посредником между этими двумя группами протоколов.

Рис. 7.7. Семиуровневая модель OSI стека протоколов TCP/IP

Каждый уровень модели OSI предоставляет определенный набор сервисов для более высокого уровня и, в свою очередь, использует сервисы, предоставляемые ему нижним уровнем. Несмотря на то, что данные передаются в стеке вертикально от уровня к уровню, логически каждый уровень одного узла взаимодействует с соответствующим логическим уровнем другого узла в сети, что отображено на рис. 7.7.

В качестве адресов отправителя и получателя в составной сети используется не MAC-адрес, а пара чисел — номер сети и номер компьютера в данной сети. В протоколах канального уровня поле "номер сети" обычно отсутствует — предполагается, что все

узлы принадлежат одной сети. Явная нумерация сетей позволяет протоколам сетевого уровня составлять точную карту межсетевых связей и выбирать рациональные маршруты при любой их топологии, используя альтернативные маршруты, если они имеются, что невозможно сделать с помощью мостов.

Таким образом, внутри сети доставка сообщений регулируется канальным уровнем, а доставка пакетов между сетями выполняется на сетевом уровне. Рассмотрим смысл уровней модели OSI более детально.

- *Физический уровень*. На этом уровне по физическим каналам (коаксиальный кабель, витая пара, оптоволоконный кабель и т. д.) сети передаются неструктурированные данные в виде потока битов информации. Уровень определяет характеристики физических сред передачи данных, параметры электрических сигналов, а также среду передачи, например, Ethernet.
- *Канальный уровень* обеспечивает передачу кадра данных между любыми узлами в сетях с типовой топологией либо между двумя соседними узлами в сетях с произвольной топологией. На этом уровне работают драйверы сетевых устройств, а также выполняется коррекция ошибок, возникающих на физическом уровне. Протоколы канального уровня используют физические (MAC) адреса, поэтому на этом уровне невозможно создавать сети с развитой структурой, объединяющие, например, несколько сетей предприятия в единую сеть, или высоконадежные сети с избыточными связями между узлами.
- *Сетевой уровень* отвечает за доставку данных между любыми двумя узлами в сети с произвольной топологией, не гарантируя при этом надежности передачи. Смысл использования сетевого уровня состоит в том, чтобы, не затрагивая технологий, используемых в объединяемых сетях, добавить в кадры канального уровня дополнительную информацию в виде заголовка пакета сетевого уровня. Заголовок пакета позволяет находить адресата в сети с любой базовой технологией, причем он имеет унифицированный формат, не зависящий от форматов кадров канального уровня сетей, входящих в объединенную сеть. В заголовке сетевого уровня содержится адрес

назначения и другая информация, необходимая для успешного перехода пакета из сети одного типа в сеть другого типа. На этом уровне функционируют протоколы так называемого разрешения адресов, отвечающие за отображение адреса узла в локальный адрес сети. Такие протоколы имеют название ARP — Address Resolution Protocol.

- *Транспортный уровень.* На этом уровне обеспечивается передача данных между любыми узлами сети с требуемым уровнем надежности, поэтому транспортный уровень предполагает наличие средств для установления соединения, а также нумерации, буферизации и упорядочивания пакетов. На этом уровне работают протоколы TCP/IP или UDP/IP, выполняющие фрагментацию (разбивку) сообщений на пакеты при передаче и сборку (дефрагментацию) полного сообщения из пакетов при приеме так, что для старших уровней модели эти процедуры являются прозрачными. Кроме этого, на этом уровне выполняется посылка и обработка подтверждений и, при необходимости, повторная передача.
- *Сеансовый уровень (уровень сессий)* предоставляет средства управления диалогом между взаимодействующими узлами, позволяющие синхронизировать операции в рамках процедуры обмена сообщениями. На этом уровне осуществляется управление переговорами взаимодействующих транспортных уровней. В UNIX этот уровень используется, например, для реализации механизма вызовов удаленных процедур (RPC).
- *Уровень представлений* управляет представлением информации, выполняя при необходимости преобразование данных, например, компрессию/декомпрессию или шифровку/дешифровку данных. В операционной системе UNIX используется при работе с сетевой файловой системой (NFS), реализуя механизм внешнего представления данных, используемого всеми компьютерами, входящими в сеть.
- *Уровень приложений (прикладной уровень).* На этом уровне работают различные сетевые сервисы, предоставляющие услуги конечным пользователям и приложениям, например, электронная почта, передача файлов, подключение удаленных

терминалов к компьютеру по сети. Выполняет функции интерфейса с такими сетевыми приложениями, как telnet, rlogin, mail и т. д.

Следует сказать, что семиуровневая модель OSI является скорее теоретической абстракцией, дающей хорошее представление о сетевом взаимодействии на разных уровнях, но не реализована в практическом плане для большинства приложений.

Во всем мире наибольшее распространение получила группа или, по-другому, стек протоколов TCP/IP, который используется в большинстве локальных и глобальных сетей, а также является основой функционирования Интернета.

7.2.2. Стек протоколов TCP/IP

Этот стек протоколов получил свое название от двух важнейших и взаимосвязанных между собой протоколов — TCP (Transmission Control Protocol) и IP (Internet Protocol).

Изначально TCP/IP был задуман как универсальное средство взаимодействия между сетями, имеющими разную физическую структуру и разные коммуникационные протоколы. При этом архитектура физических сетей должна быть скрыта от пользовательских приложений.

Еще одна цель, которую преследовали разработчики стека протоколов, — обеспечить единый интерфейс пользователя независимо от того, в какой сети он работает. Эта цель была реализована посредством глобальной сети, известной больше под названием Интернет.

Для представления стека протоколов TCP/IP используется 4-уровневая модель, совместимая с OSI, из которой исключены или объединены некоторые уровни OSI (рис. 7.8).

Все четыре уровня в этой практической модели выполняют определенные функции, включая в себя те или иные протоколы. При этом некоторые уровни в этой модели объединяют функциональность нескольких уровней в классической семиуровневой модели OSI.

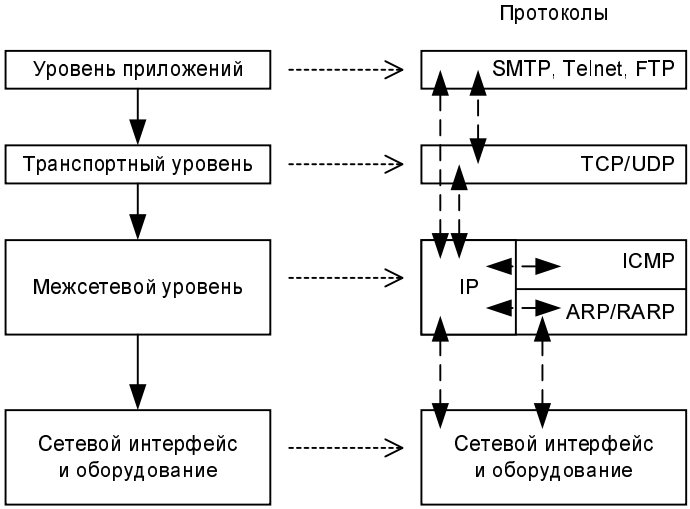


Рис. 7.8. Стек протоколов TCP/IP

Функции каждого из четырех уровней описаны далее.

- *Сетевой интерфейс.* Этот уровень отвечает за установление сетевого соединения в конкретной физической сети, к которой подсоединен компьютер. На этом уровне работают драйвер устройства в операционной системе и соответствующая сетевая плата компьютера. На самом нижнем уровне сетевого интерфейса используются специальные протоколы разрешения адресов ARP (Address Resolution Protocol) и RARP (Reverse Address Resolution Protocol). Они применяются только в определенных типах физических сетей (Ethernet и Token Ring) для преобразования адресов сетевого уровня в адреса физической сети и обратно.
- *Межсетевой уровень.* На этом уровне реализуется ненадежная служба доставки пакетов по сети от системы к системе без установления соединения, в частности, маршрутизация пакетов по Интернету. Это означает, что будут выполнены все необходимые операции для доставки пакетов, хотя сама доставка не гарантируется: пакеты могут быть потеряны, переданы

в неправильном порядке, продублированы и т. д. На этом уровне принимается решение о маршрутизации пакета по межсетевым соединениям, в которых используется протокол IP — основной протокол сетевого уровня. Он используется протоколами TCP и UDP, функционирующими на транспортном уровне. Протокол IP определяет базовую единицу передачи данных в Интернете — IP-дейтаграмму, указывая точный формат всей информации, проходящей по сети TCP/IP. Программное обеспечение IP выполняет функции маршрутизации при выборе пути данных по физическим сетям. Для выполнения таких функций поддерживаются специальные таблицы, а выбор осуществляется на основе адреса сети, в которой находится компьютер-адресат. Протокол IP определяет маршрут отдельно для каждого пакета данных, не гарантируя надежной доставки в нужном порядке. Он задает непосредственное отображение данных на нижележащий физический уровень передачи, реализуя тем самым доставку пакетов. Кроме протокола IP, на сетевом уровне используются также протоколы ICMP (Internet Control Message Protocol) и IGMP (Internet Group Management Protocol). Протокол ICMP отвечает за обмен сообщениями об ошибках и другой важной информацией с сетевым уровнем на другом хосте или маршрутизаторе, а IGMP используется для отправки IP-дейтаграмм множеству хостов в сети.

- *Транспортный уровень*. Этот уровень реализует надежную передачу данных, при этом работающие на данном уровне два основных протокола, TCP и UDP, осуществляют связь между машиной-отправителем пакетов и машиной-адресатом. При анализе протоколов транспортного уровня часто возникает вопрос: зачем нужны два транспортных протокола TCP и UDP? Дело в том, что каждый из этих протоколов предоставляет определенные услуги прикладным процессам. Как правило, большинство прикладных программ используют только один из них.

  - Протокол TCP (Transmission Control Protocol) обеспечивает надежную передачу данных между двумя хостами. Он позволяет клиенту и серверу приложения устанавливать ме-

жду собой логическое соединение и затем использовать его для передачи больших массивов данных, как если бы между ними существовало прямое физическое соединение. Протокол позволяет осуществлять фрагментацию потока данных, подтверждать получение пакетов данных, задавать тайм-ауты (которые позволяют подтвердить получение информации), организовывать повторную передачу в случае потери данных и т. д. Поскольку данный транспортный протокол реализует гарантированную доставку информации, то использующие его приложения получают возможность игнорировать все детали такой передачи.

- Протокол UDP (User Datagram Protocol) реализует гораздо более простой сервис передачи, обеспечивая, подобно протоколам сетевого уровня, ненадежную доставку данных без установления логического соединения, но, в отличие от IP, для прикладных систем на хост-компьютерах. Он просто посылает пакеты данных, дейтаграммы (datagrams) с одной машины на другую, но не предоставляет никаких гарантий их доставки. Все функции надежной передачи должны встраиваться в прикладную систему, использующую UDP. Протокол UDP имеет и некоторые преимущества перед TCP. Для установления логических соединений нужно время, и они требуют дополнительных системных ресурсов для поддержки на компьютере информации о состоянии соединения. UDP занимает системные ресурсы только в момент отправки или получения данных. Поэтому если распределенная система осуществляет непрерывный обмен данными между клиентом и сервером, связь с помощью транспортного уровня TCP окажется для нее более эффективной. Если же коммуникации между хостами осуществляются редко, предпочтительней использовать протокол UDP. Среди известных приложений, использующих TCP, такие, как telnet, ftp и smtp. Протоколом UDP пользуется, в частности, протокол сетевого управления SNMP.

❒ *Уровень приложений.* На этом уровне выполняются приложения типа "клиент-сервер", базирующиеся на протоколах ниж-

них уровней. В отличие от протоколов остальных трех уровней, протоколы этого уровня работают непосредственно с приложением, не вдаваясь в детали передачи данных по сети. К протоколам, работающим на этом уровне, можно отнести telnet, протокол передачи файлов FTP, протокол электронной почты SMTP, протокол управления сетью SNMP, используемый в World Wide Web, протокол передачи гипертекста HTTP и др.

7.3. Сетевые приложения

Взаимодействие двух и более процессов в операционной системе UNIX можно описать в терминах модели "клиент-сервер". Процесс-сервер отвечает на запросы процесса-клиента, выполняя те или иные действия, например, передавая клиенту блоки данных или принимая от него данные для обработки. Один и тот же процесс может выступать одновременно как сервером для одних процессов, так и клиентом для других. Взаимодействие между клиентом и сервером осуществляется при помощи механизмов межпроцессных коммуникаций, одним из которых является обмен данными с использованием сетевых протоколов TCP/IP.

Особенностью сетевого протокола TCP/IP является то, что он позволяет осуществлять обмен данными как между отдельными процессами, выполняющимися на разных хостах (компьютерах) в сети, так и между процессами в одной и той же UNIX-системе.

Обычно процесс-сервер может обслуживать одновременно несколько запросов, т. е. может работать с несколькими клиентами одновременно. Взаимодействие клиентов и сервера иллюстрирует рис. 7.9.

Как клиенты, так и серверы могут использовать протоколы TCP/IP непосредственно или протоколы более высокого уровня, например, уровня приложений (telnet, FTP и т. д.).

Применительно к сетевому взаимодействию очень часто используются термины "сетевой сервис" или, что то же самое,

"сетевая служба". Например, когда упоминается "сетевой сервис telnet", то предполагается, что в операционной системе UNIX имеется определенный набор программных или иных (в частности, конфигурационных) файлов, посредством которых реализуется обмен данными по протоколу telnet. Программный файл, обслуживающий любые поступающие запросы по telnet из многочисленных источников и в произвольные моменты времени, обычно запускается как процесс-демон (обычно это `telnetd`) — его очень часто называют сервером telnet. По аналогии, демон `ftpd`, реализующий сервис FTP, принято называть сервером FTP и т. д.

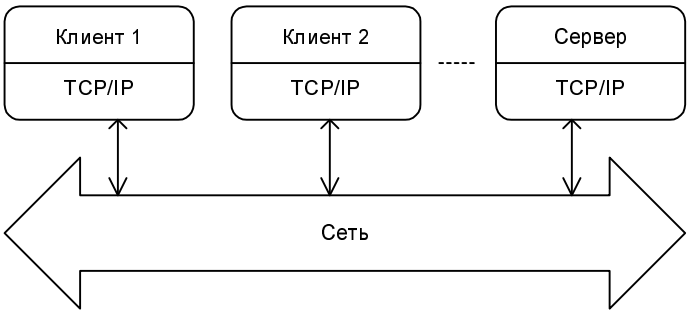


Рис. 7.9. Пример сетевого взаимодействия "клиент-сервер"

Для того чтобы два и более сетевых приложения взаимодействовали между собой, нужен какой-то способ, идентифицирующий каждое приложение в сети. Таким механизмом являются гнезда или, по-другому, *сокеты* (sockets). Во всех операционных системах (не только UNIX) сокет реализован как программная конструкция, представляющая собой пару "IP-адрес — порт". IP-адрес мы уже рассматривали, что же касается порта, то он представляет собой целочисленное значение из диапазона 0—65 536. Таким образом, каждое сетевое приложение может быть однозначно идентифицировано по IP-адресу сетевого интерфейса и по номеру порта. Сокет является таким же объектом файловой системы, как и дисковый файл или программный канал, поэтому лю-

бое сетевое приложение TCP/IP должно получать каким-то образом дескриптор (описатель) сокета, который далее может быть использован в операциях сетевого обмена.

Механизм сокетов впервые был реализован в 1982 году в UNIX BSD 4.1 в качестве развитого средства межпроцессных взаимодействий. Это средство, вообще говоря, позволяет любому процессу обмениваться сообщениями с любым другим процессом, независимо от того, выполняются они на одном компьютере или на разных машинах, соединенных сетью.

Механизм сокетов является обязательным компонентом всех версий операционных систем UNIX, однако в различных системах он реализован по-разному. В BSD-совместимых системах сокеты реализованы как функции ядра, и пользователи могут применять специальные системные вызовы `socket()`, `bind()`, `listen()`, `connect()` и `accept()` при разработке сетевых приложений.

В операционных системах, совместимых с System V, механизм сокетов реализован не в ядре системы, а представлен набором функций в библиотеке /usr/lib/libsocket.a.

Серверы сетевых приложений обычно имеют заранее известные номера портов, которые используются программами-клиентами для запросов на обслуживание. Например, в каждой реализации TCP/IP, которая поддерживает сервер FTP, этот протокол передачи файлов получает для своего сервера номер TCP-порта 21, telnet-сервер имеет TCP-порт 23, а сервер протокола TFTP (Trivial File Transfer Protocol) — UDP-порт 69. Сервисам, которые поддерживаются в любой реализации TCP/IP, назначаются номера портов в диапазоне от 1 до 1023. Назначение номеров портов находится в ведении организации IANA (Internet Assigned Numbers Authority).

Перейдем к обсуждению наиболее широко используемых протоколов уровня приложений и связанных с ними сетевых сервисов. Начнем с протокола telnet.

Протокол telnet реализован на базе протокола TCP и позволяет работать с удаленным компьютером через стандартный вирту-

альный терминал строчного типа в кодировке ASCII. При этом обеспечивается выполнение довольно сложных функций, включая локальный или удаленный эхо-контроль, страничный режим и т. д. На серверной стороне протокол telnet реализован как процесс-демон `telnetd` (`in.telnetd` в операционных системах Solaris), к которому обращается программа-клиент.

Демон `telnetd` обычно запускается системой посредством суперсервера `inetd` (хотя может быть запущен и вручную) и прослушивает TCP-порт 23 в ожидании соединений. При поступлении запроса от клиента на обслуживание `telnetd` открывает для каждого удаленного клиента виртуальный терминал (псевдотерминал), с которым связывает стандартный ввод/вывод, а также стандартное устройство ошибок.

В процессе взаимодействия с клиентом `telnetd` посылает ему команды настройки (эхо, обмен двоичной информацией, тип терминала, скорость обмена, переменные окружения). Обмен данными происходит с использованием стандартных команд протокола telnet.

Для вызова сервиса `telnet` клиент (пользователь) может воспользоваться либо командной строкой, либо установить связь в режиме удаленного терминала. Работа в режиме удаленного терминала позволяет использовать как буферизованный, так и посимвольный ввод/вывод. При работе в посимвольном режиме каждый введенный символ немедленно отправляется на удаленную машину, откуда приходит эхо-ответ. Если используется буферизованный ввод/вывод, то введенные символы накапливаются в буфере памяти клиентской программы и отправляются на удаленную машину в виде пакета данных.

В практическом плане для начала сеанса `telnet` пользователь вводит с консоли командную строку

`telnet` *`host`*

Здесь *`host`* — имя удаленного компьютера (хоста) или его IP-адрес. Если соединение было успешным, пользователь получает на экране приглашение к входу в машину *`host`*.

Протокол FTP (File Transfer Protocol, протокол передачи файлов), так же как и telnet, распространен очень широко и в качестве транспортного протокола использует TCP. С помощью FTP можно просмотреть, например, каталог удаленной машины, перейти из одного каталога в другой, а также скопировать один или несколько файлов. Архивы FTP представляют собой один из основных информационных ресурсов Интернета и содержат огромное количество различной информации, включая текстовые документы, программное обеспечение, аудио- и видеоинформацию, хранящуюся в виде файлов на компьютерах во всем мире.

Как и многие другие виды сервиса, FTP осуществляет обмен данными по принципу "клиент-сервер". Программа-сервер реализована в виде демона `ftpd` (`in.ftpd` в Solaris), который принимает входящие запросы на TCP-порт 21. Демон `ftpd` запускается суперсервером `inetd`, хотя может выполняться и как изолированное приложение-сервер. Для большей безопасности доступ к FTP-серверу можно установить для определенных пользователей, потребовав при открытии сеанса пароль. Тем не менее, одним из самых распространенных видов FTP-серверов является так называемый анонимный FTP-сервер. При подключении к нему для соединения и получения файлов не нужно знать имя пользователя и его пароль. В качестве имени пользователя на анонимном FTP-сервере обычно используется "anonymous" или "ftp", а в качестве пароля — пустая строка или адрес электронной почты клиента.

Протокол SMTP (Simple Mail Transfer Protocol, простой протокол передачи почты) обеспечивает передачу сообщений электронной почты между произвольными узлами Интернета. Имея механизмы промежуточного хранения почты и механизмы повышения надежности доставки, протокол SMTP допускает использование различных транспортных служб. При этом он может функционировать даже в тех сетях, которые не используют протоколы TCP/IP. Протокол SMTP обеспечивает группирование сообщений, предназначенных для отдельного получателя, а также создание нескольких копий сообщения для передачи в разные адреса. Этот протокол будет рассмотрен более подробно при анализе работы электронной почты.

Еще один протокол SNMP (Simple Network Management Protocol, простой протокол управления сетью) работает на базе UDP и применяется для управления сетевыми управляющими станциями. Рабочие станции, использующие этот протокол, могут собирать информацию о состоянии сети в определенном формате, после чего можно обрабатывать и интерпретировать полученные данные.

Протокол TCP лежит в основе функционирования системы X Window, позволяющей работать с графикой и текстом и являющейся основой графического интерфейса пользователя в подавляющем большинстве операционных систем.

Сетевая файловая система NFS (Network File System) использует транспортные услуги протокола UDP, позволяя монтировать файловые системы нескольких UNIX-машин и работать с ними как с локальными ресурсами.

Работу многочисленных хостов в Интернете обеспечивает еще один протокол — HTTP, которому мы уделим достаточно внимания далее в этой главе, и который также базируется на стеке протоколов TCP/IP.

Кроме упомянутых сетевых сервисов чрезвычайно важным является так называемый сервис имен или *служба разрешения имен* (DNS), лежащая в основе работы как локальных сетей, так и Интернета. Очень широко используется протокол динамического присвоения адресов (DHCP), который позволяет автоматизировать сетевые настройки компьютеров, находящихся в локальных или глобальных сетях.

Для операционной системы UNIX имена большинства сетевых сервисов и порты, на которых они работают, описаны в файле /etc/services. Так, для `telnet` и `ftp` полученная информация может выглядеть так:

```
# cat /etc/services|grep telnet;cat /etc/services|grep ftp
telnet          23/tcp
ftp-data        20/tcp
ftp             21/tcp
tftp            69/udp
```

Из результата видно, что на данном хосте служба `telnet` для обмена данными использует TCP-порт 23, а служба `ftp` — TCP-порт 21.

Подробную информацию о работающих сетевых приложениях можно получить и при помощи утилиты `netstat` (ее мы рассмотрим далее в этой главе), которая реализована во всех версиях операционной системы UNIX.

7.4. Адресация в Интернете

Узлы сети должны обладать методом идентификации, позволяющим однозначно определить, каким образом можно получить доступ к каждому из них. Для стека протоколов TCP/IP выбрана схема идентификации, при которой каждому сетевому интерфейсу (сетевой карте либо последовательному порту) присваивается уникальный 32-разрядный адрес (IP-адрес), используемый для всех коммуникаций с этим интерфейсом в Интернете.

IP-адрес компьютера позволяет задавать идентификатор сети, к которой подсоединен компьютер, а также уникальный идентификатор самого компьютера. IP-адрес представляет собой четырехбайтовое число, записываемое либо в шестнадцатеричном виде, например, C0A10D02, либо в десятичном виде, где байты разделены точками, например, 192.161.12.2 (в обоих случаях показан один и тот же IP-адрес).

Далее показаны примеры записи IP-адресов в десятичной нотации:

```
193.124.148.73
128.8.2.1
```

Кроме IP-адреса, обязательным атрибутом является *маска сети*, которая определяет размер сети, т. е. число адресов в сети. Она представляет собой четырехбайтовое число, старшие биты которого, начиная с определенной позиции, всегда установлены в единицу, а все младшие — в ноль. Маска задается либо количе-

ством битов (например, 8 битов — 256 адресов, 6 битов — 64 адреса), либо их последовательностью, например:

```
111...11100...00
```

Ее можно записать и в десятичной нотации, например:

- 255.255.255.192 — маска на 64 адреса (192 = 256 − 64);
- 255.255.255.0 — маска на 256 адресов;
- 255.255.0.0 — маска на 64 Кбайт адресов.

В отдельной сети все хосты имеют IP-адреса с одинаковым адресом сети и одинаковой маской. В одной локальной сети можно совместить две и более разных IP-сетей (они называются подсетями), при этом они даже могут "не знать" о существовании друг друга и, тем не менее, обмениваться данными.

Самый старший адрес в сети используется для широковещательной (broadcast) рассылки, например

```
128.8.255.255
```

Если нужно указать сочетание номера и маски, то можно использовать запись "номер/число_установленных_битов_в_маске". Так, сочетание IP-адреса 192.168.14.5 и маски 255.255.255.0 будет записано в виде 192.168.14.5/24.

Сети делятся на несколько классов, которые имеют свои обозначения:

- класс A — так называемые "большие" сети. Адреса этих сетей находятся в диапазоне 1—126, маска равна 255.0.0.0, а сами сети могут содержать до 16 777 216 адресов (256×256×256). Адреса хостов в этих сетях имеют вид "125.*.*.* ";
- класс B — так называемые "средние" сети. Адреса этих сетей лежат в диапазоне 128.0—191.255, маска равна 255.255.0.0, а сама сеть может содержать до 65 536 адресов (256×256). Адреса хостов в этих сетях имеют вид "136.12.*.* ";
- класс C — "небольшие" сети. Их адреса лежат в диапазоне 192.0.0—255.254.255, маска сети равна 255.255.255.0, а сама сеть содержит 254 адреса. Адреса хостов в этих сетях имеют вид "195.136.12.* ".

Большинство сетевых приложений автоматически определяют класс сети по адресу, а сами сети при необходимости можно разбивать на две и более подсети с любыми масками.

Особым случаем сети класса A является сеть с номером 127 — так называемый "интерфейс обратной связи" (loopback interface) — такая сеть предназначена для настройки и проверки сетевого интерфейса самого хоста. В ряде случаев интерфейс обратной связи используют для межпроцессных коммуникаций на локальной машине.

Сети класса C используются в небольших организациях, объединяющих до 255 машин, с адресами из диапазона 192.0.0.0— 223.255.255.255. Для идентификатора сети отводится 21 бит, а для идентификатора хоста — 8 битов.

В более крупных организациях используются сети класса B, способные обслужить до 256 сетей, каждая из которых может содержать до 64 тыс. рабочих станций. Что же касается сетей класса A, то они используются в глобальных сетях очень большого масштаба, например, ARPANET.

Адресом (номером) сети является число, полученное с помощью поразрядной операции логическое И над номером сетевого интерфейса и маски. При этом в номере интерфейса обнуляются биты на тех позициях, где находятся нулевые биты в маске.

Каждая сеть имеет свой уникальный номер, например,

```
194.15.28.0
```

который не может быть присвоен ни одному конкретному хосту.

Последний номер в сети предназначен для передачи широковещательных сообщений (broadcasting), которые передаются всем узлам данного сегмента сети, например, `194.15.28.255`. Следовательно, при выделении группы адресов в сеть два адреса — сети и широковещательной рассылки — становятся недоступными для конкретных хостов. С помощью широковещательных адресов можно обращаться ко всем хостам сети, при этом такие адреса имеют специальный идентификатор, состоящий только из единиц. Механизм широковещательной передачи существенно зависит от характеристик конкретной физической сети. В сети

Ethernet, например, широковещательная передача может выполняться так же, как и обычная передача данных, хотя есть сети, которые вообще не поддерживают такой тип передачи или имеют для этого ограниченные возможности.

Существует еще один специальный тип адресов — так называемые *групповые адреса*. Групповые адреса, которые иногда называют адресами класса D, позволяют отправлять сообщения определенному множеству адресатов (multicasting). Многие приложения используют групповую адресацию: программы интерактивных конференций, отправки почты или новостей группе получателей. Для поддержки групповой передачи хосты и маршрутизаторы используют протокол IGMP, предоставляющий всем системам в физической сети информацию о принадлежности хоста к той или иной группе.

Хосты могут находиться сразу в нескольких физических сетях: в этом случае им присваивается несколько IP-адресов — по одному для каждой сети.

Напомню, что каждая сетевая карта или, в терминологии UNIX, каждый сетевой интерфейс имеет уникальный 32-разрядный адрес. В настоящее время, в связи с бурным ростом Интернета, 32-битовая схема адресации нынешней версии IPv4 протокола IP уже не удовлетворяет потребности Всемирной сети. В связи с этим была разработана новая версия протокола IP, известная как IPv6, проект которой был обнародован в 1991 году. Стандарт IPv6 предполагает использование 128-битового формата IP-адреса и, кроме того, поддерживает автоматическое назначение адресов.

В данной главе мы ограничимся рассмотрением особенностей сетевых настроек UNIX только для версии IPv4.

Установка IP-адреса для одной машины не является сложной задачей, но для большой корпоративной сети, в которую входит большое число компьютеров, которые могут постоянно перемещаться, управление IP-адресами вручную представляет собой довольно сложную задачу. Если, например, компьютер перемещается в другую подсеть, то следует изменить его сетевую идентификацию таким образом, чтобы часть адреса с номером хоста

не конфликтовала с адресами узлов новой подсети. Для автоматизации задач управления сетями существуют определенные решения, способствующие повышению эффективности использования адресов, а также средства организации виртуальных частных сетей.

Для автоматизации управления сетями был разработан протокол динамической конфигурации хоста, известный под названием DHCP (Dynamic Host Configuration Protocol). В соответствии с этим протоколом одна из машин сети настраивается как сервер DHCP, а все остальные становятся его клиентами. Протокол DHCP поддерживает динамическое назначение IP-адресов, позволяя присвоить хосту IP-адрес из заданного при конфигурации сервера диапазона IP-адресов. При этом адрес выделяется на ограниченный период времени или до отказа клиента от его использования, что позволяет эффективно использовать лимитированное число IP-адресов.

Протокол DHCP, кроме поддержки автоматического присвоения постоянного IP-адреса, позволяет назначать адреса вручную, при этом сервер DHCP только извещает клиента о назначении. Реализация клиентской части протокола поддерживается большинством операционных систем. Более того, TCP/IP дает пользователям возможность работать не с адресами хостов, а с их символьным представлением (именами), что намного удобнее для восприятия. Такой механизм отображения IP-адресов на имена хостов реализован в виде распределенной базы данных DNS (Domain Name System). При этом любая программа может вызвать стандартную библиотечную функцию для преобразования IP-адреса в соответствующее имя хоста или наоборот.

База данных DNS является распределенной и включает информацию только об ограниченном числе хостов в пределах определенного региона. Многие организации и компании поддерживают свою базу данных DNS, причем к ней могут обращаться другие клиенты сети. Кроме того, распределенная база DNS, содержащаяся на сервере, взаимодействует с клиентами посредством определенного протокола.

Современные сети очень редко бывают изолированными от остального мира — в подавляющем большинстве случаев узлы сети должны взаимодействовать не только между собой, но и с хостами, находящимися в других сетях. Такое взаимодействие возможно только при использовании механизма маршрутизации, который мы рассмотрим в следующем разделе.

7.5. Маршрутизация

Термином "маршрутизация" определяют механизм взаимодействия хостов (узлов), находящихся в разных сетях. Маршрутизация необходима, например, при подключении компьютера к Интернету или к другой сети, поэтому практически каждый пользователь рано или поздно столкнется с проблемой настройки маршрутизации на своей машине.

Функции маршрутизации выполняют специальные устройства — маршрутизаторы (routers), которые нередко называют *шлюзами*. Маршрутизаторы соединяют две и более сети, позволяя выполнять пересылку пакетов между ними (IP-forwarding).

Применительно к физической структуре сети можно сказать, что в общем случае все сети TCP/IP включают два типа функциональных единиц — хосты, куда входят компьютеры и прочее сетевое оборудование, и маршрутизаторы. Во всех сетях обязательно присутствуют хосты, но далеко не в каждой есть маршрутизаторы.

Функции маршрутизатора может выполнять как специальное устройство, так и обычный компьютер, имеющий две сетевые карты и настроенный соответствующим образом. Маршрутизаторы должны быть сконфигурированы так, чтобы можно было пересылать пакеты не только между соседними сетями, но и передавать пакеты узлам, не входящим в такие сети.

Вот простейший пример взаимодействия двух сетей посредством маршрутизатора (рис. 7.10).

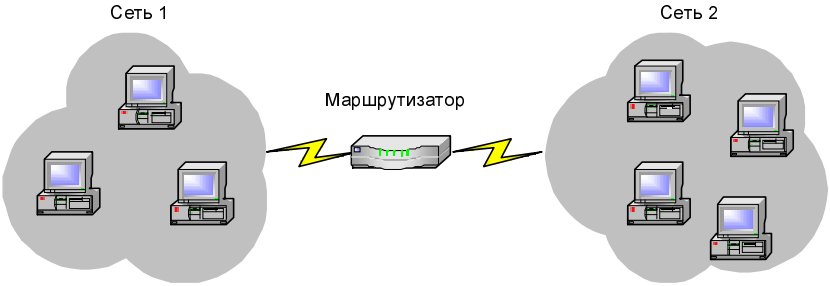


Рис. 7.10. Взаимодействие двух сетей посредством маршрутизатора

Как видно из рис. 7.10, сеть 1 связана с сетью 2 посредством маршрутизатора, что делает возможным обмен данными между этими сетями.

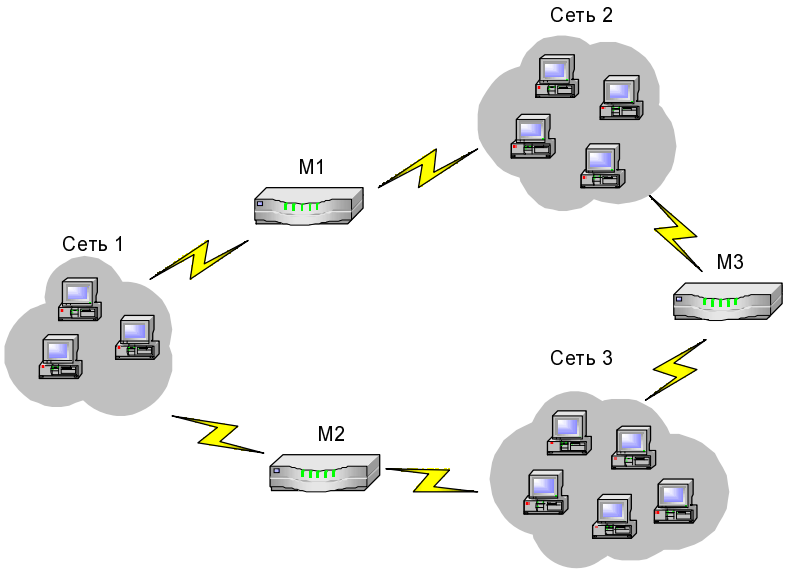


Рис. 7.11. Пример взаимодействия трех сетей посредством маршрутизаторов

В более сложных случаях, когда необходимо связывать между собой несколько сетей, используется несколько маршрутизаторов, причем для доступа к одной и той же сети могут устанавливаться альтернативные маршруты, как показано на рис. 7.11.

На этой схеме маршрутизатор M1 напрямую соединяет сеть 1 с сетью 2, маршрутизатор M2 соединяет сеть 1 и сеть 3, а маршрутизатор M3 — сеть 2 с сетью 3. Если, например, канал связи на маршруте M2 по какой-либо причине становится недоступным, то сеть 1 будет иметь доступ к сети 3 посредством маршрутизаторов M1 и M3.

Если канал связи на маршруте M3 становится недоступным, то и в этом случае все сети будут видеть друг друга.

Маршрутизаторы определяют дальнейший маршрут пакета, используя так называемые *таблицы маршрутизации* (routing tables). Эти таблицы содержат IP-адреса хостов и маршрутизаторов в тех сетях, с которыми данный маршрутизатор имеет связь, а также указатели на эти сети. При получении пакета маршрутизатор просматривает таблицу маршрутизации, пытаясь определить, содержит ли таблица IP-адрес, указанный в заголовке IP-пакета. Если маршрутизатор не обнаружит этот адрес, то он переправляет этот пакет другому маршрутизатору, указанному в таблице маршрутизации.

Процесс маршрутизации можно проиллюстрировать рис. 7.12. Здесь показана топология трех сетей, соединенных двумя маршрутизаторами.

Маршрутизатор M1 соединяет сети 194.15.26 и 194.15.27, а маршрутизатор M2 соединяет сети 194.15.27 и 194.15.28. Предположим, что хост A, находящийся в сети 194.15.26, отправляет данные хосту B из сети 194.15.28. В этом случае выполняется следующая последовательность действий:

1. Хост A посылает пакет через сеть 194.15.26. Заголовок отправляемого пакета содержит IPv4-адрес получателя (194.15.28.10). Поскольку ни один из хостов сети 194.15.26 не имеет IPv4-адреса 194.15.28.10, то пакет принимает маршрутизатор M1.

2. Маршрутизатор М1 проверяет свои таблицы маршрутизации и обнаруживает, что ни один из компьютеров сети 194.15.27 не имеет адрес 194.15.28.10. Поскольку таблицы маршрутизации содержат адрес следующего маршрутизатора, каким является маршрутизатор М2, то именно ему маршрутизатор М1 и переправляет пакет.
3. Маршрутизатор М2 соединяет сети 194.15.27 и 194.15.28, поэтому располагает информацией о компьютере хост В. На последнем шаге маршрутизатор 2 передает пакет в сеть 194.15.28 и хост В его принимает.

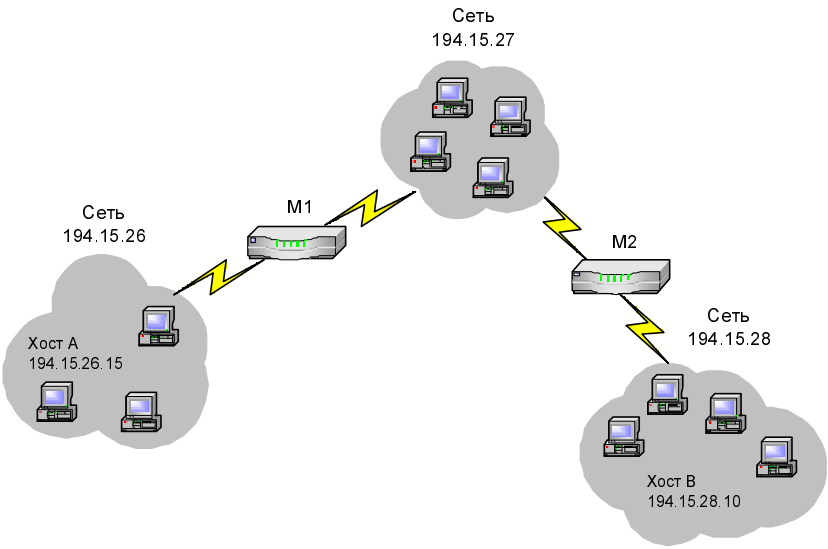


Рис. 7.12. Пример маршрутизации между тремя сетями

Как хост узнает, что нужно передавать пакеты в другую сеть, и как определяет, какой маршрутизатор для этого использовать? IP-адрес получателя, включаемый в заголовок пакета, определяет дальнейший маршрут пакета. Если IP-адрес узла доставки включает номер локальной сети, пакет будет доставлен адресату с этим IP-адресом, находящемуся в данной сети. Если же номер

сети не соответствует локальной сети, то пакет направляется маршрутизатору, находящемуся в данной сети.

Для определения маршрута передачи используется протокол ARP, а сам процесс передачи данных показан на рис. 7.13.

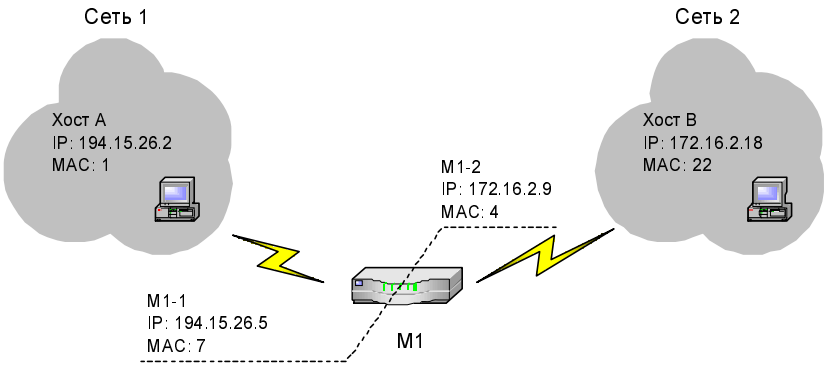


Рис. 7.13. Процесс передачи пакета посредством маршрутизатора

Рассмотрим более подробно передачу пакета данных из сети 1 в сеть 2. Пусть хост A, подключенный к сети 1 через сетевой интерфейс с IP-адресом 194.15.26.2 и аппаратным (MAC) адресом 1, пытается передать пакет данных хосту B с IP-адресом 172.16.2.18 и MAC-адресом 22. Напомню, что для доставки сетевого пакета узлу необходимо знать, кроме IP-адреса, также и его MAC-адрес — доставка пакета узлу невозможна, если неизвестен его аппаратный адрес.

Вот примерная последовательность шагов для передачи пакета:

1. Хост A отправляет широковещательный ARP-запрос об аппаратном адресе узла с адресом 172.16.2.18. Если бы данный узел находился в сети 1, то он бы ответил на запрос хоста A, однако хост B находится в другой сети, и ответа от него не будет.
2. С помощью ARP хост A определяет MAC-адрес маршрутизатора по умолчанию (default gateway). IP-адрес 194.15.26.5

маршрутизатора должен быть известен хосту А (он указывается при настройке сетевых параметров для данного хоста), но аппаратный адрес маршрутизатора должен определяться при помощи протокола ARP (в данном случае MAC-адрес маршрутизатора по умолчанию равен 7).

3. Хост А отправляет пакет данных маршрутизатору с IP-адресом 194.15.26.5 (M1-1), причем заголовок каждого пакета содержит следующую информацию:
   - MAC-адрес отправителя: 1;
   - IP-адрес отправителя: 194.15.26.2;
   - аппаратный адрес узла-получателя: 7;
   - IP-адрес узла-получателя: 172.16.2.18.
4. Маршрутизатор, имеющий IP-адрес 194.15.26.5 и MAC-адрес 7, по заголовку пакета определяет, что пакет предназначен для дальнейшей передачи в сеть 172.16.2.
5. Маршрутизатор выполняет ARP-запрос для определения аппаратного адреса узла 172.16.2.18, причем аппаратный адрес сохраняется в кэше для последующего использования.
6. Маршрутизатор M1-1 отправляет пакет в сеть 172.16.2, поместив следующую информацию в заголовок пакета:
   - аппаратный адрес отправителя: 7;
   - IP-адрес отправителя: 194.15.26.2;
   - MAC-адрес получателя: 22;
   - IP-адрес узла-получателя: 172.16.2.18.
7. Данные передаются по сети 172.16.2, причем сетевая карта узла адресата распознает свои IP- и MAC-адрес и получает данные.

Обратите внимание на то, что IP-адрес узла-отправителя (194.15.26.2) сохраняется в заголовке пакета, когда пакет доходит до узла адресата. В то же время аппаратные (MAC) адреса в процессе передачи пакета будут изменяться таким образом, чтобы указывать на последний шлюз, через который прошел пакет. Когда пакет "перепрыгивает" из одной сети в другую, IP-адреса

узла-отправителя и узла-адресата не изменяются, но MAC-адрес будет соответствовать устройствам, посредством которых передавался пакет.

Протокол ARP реализован в виде набора системных вызовов, используемых сетевыми программами для получения аппаратных адресов сетевых интерфейсов. Обширную информацию можно получить при помощи команды `arp`, которая запускается из командного интерпретатора shell.

Для настройки маршрутизации на уровне операционной системы можно использовать статическую или динамическую маршрутизацию.

Для статической маршрутизации используются так называемые таблицы маршрутизации, маршруты в которые добавляются и удаляются вручную с помощью команды `route`. Статическая маршрутизация лучше всего подходит для небольших сетей, если существует только одна точка подключения к другим сетям и нет запасных маршрутов (запасные маршруты могут быть использованы в случае выхода из строя основных). Если хотя бы одно из трех приведенных выше условий невыполнимо, лучше использовать динамическую маршрутизацию.

Динамическая маршрутизация позволяет выбрать наилучший маршрут из имеющихся — ее удобно использовать при наличии двух и более маршрутов к определенному узлу сети. Здесь используются так называемые таблицы динамической маршрутизации, которые создаются и поддерживаются одним из протоколов динамической маршрутизации, программно реализованным в виде демона маршрутизации. Наиболее часто в UNIX-системах используются демоны маршрутизации `routed` и `gated`, которые обычно запускаются при старте системы и выполняются все время, пока система работает. Протоколы маршрутизации, используемые на конкретном хосте, определяют, как будет происходить обмен информацией о маршрутах с удаленными маршрутизаторами.

Данные о маршрутах включаются в таблицы динамической маршрутизации на основе информации, полученной от маршру-

тизаторов сети протоколом динамической маршрутизации. Кроме этого, протокол выбирает оптимальные маршруты доставки информации по указанному адресу. Естественно, что маршрутизаторы сети должны быть настроены для поддержки протокола динамической маршрутизации.

Демон маршрутизации обновляет таблицу маршрутизации примерно один раз каждые 30 секунд. Кроме того, таблица маршрутизации обновляется при приеме ICMP-сообщения о необходимости перенаправления (ICMP redirect). Обновление таблицы маршрутизации происходит и при загрузке системы, позволяя установить некоторые исходные маршруты.

Нужно отличать маршрутизацию от протоколов маршрутизации: маршрутизация осуществляет передачу данных и базируется на информации, находящейся в таблицах маршрутизации, в то время как протоколы маршрутизации реализованы в виде программ, обновляющих информацию в таблицах маршрутизации. Можно настроить маршрутизацию так, что не будут использоваться никакие протоколы, но при этом придется создавать таблицы маршрутизации вручную.

Для получения информации о существующих маршрутах в операционной системе UNIX имеется несколько команд, из которых наиболее часто используются `route` и `netstat`. Следует сказать, что эти команды присутствуют во всех реализациях UNIX, хотя и имеют некоторые отличия в используемых опциях этих команд, а также в форме выводимой информации.

Например, в операционной системе Solaris для получения информации о маршрутах применяется утилита `netstat`, а в Linux — `netstat` и `route`, причем `route` используется чаще.

Статическую маршрутизацию в UNIX-системах настроить несложно — для этого используется команда `route`, позволяющая явным образом указывать маршрут к узлу назначения. Вот пример добавления и изменения в операционной системе Solaris (System V):

```
# route add 194.15.26.0 194.15.28.11 255.255.255.0
add net 194.15.26.0: gateway 194.15.28.11
```

```
# route change 194.15.26.0 194.15.28.12 255.255.255.0
change net 194.15.26.0: gateway 194.15.28.12
```

Первая команда `route` устанавливает маршрут к сети 194.15.26.0 через маршрутизатор с IP-адресом 194.15.28.11. Первым параметром этой команды является ключевое слово `add`, означающее, что нужно добавить маршрут. Следующим идет IP-адрес пункта назначения, маршрут к которому нужно установить, — в данном случае таким пунктом назначения является сеть 194.15.26.0. Вместо IP-адреса сети можно указать ее имя, но для этого оно должно быть указано в файле /etc/networks.

В качестве пункта назначения может выступать не только сеть, но и отдельный хост, который можно задать либо через его IP-адрес, либо, указав его имя, если таковое присутствует в файле /etc/hosts. Следующим параметром после пункта назначения является IP-адрес маршрутизатора (194.15.28.11).

В командной строке после адреса маршрутизатора можно (но не обязательно) указать метрику маршрута (routing metric). Данный параметр иногда используется в некоторых операционных системах при оценке маршрута (прямой-непрямой, используется ли внешний маршрутизатор).

Вторая команда `route` изменяет параметры маршрута, указывая другой IP-адрес (194.15.28.12) маршрутизатора, поэтому первым параметром этой команды является ключевое слово `change`.

Проверить результат внесенных изменений можно с помощью команды `netstat`:

```
# netstat -r

Routing Table: IPv4
  Destination     Gateway       Flags  Ref   Use   Interface
-------------------------------------------------------------
---------
194.15.28.0       yuryhost        U     1     0    rtls0
```

```
194.15.26.0       194.15.28.12    UG    1     0
224.0.0.0         yuryhost        U     1     0    rtls0
localhost         localhost       UH    4     88   lo0
```

Выводимая информация имеет дополнительное поле `Flags`, в котором устанавливаются специальные флаги, смысл которых объясняется далее:

- `U` — маршрут активен;
- `G` — маршрут подключен к шлюзу (маршрутизатору). Если флаг не установлен, то узел назначения подключен непосредственно, минуя маршрутизатор;
- `H` — указанный маршрут имеет пунктом назначения хост, а не сеть;
- `D` — маршрут был создан посредством перенаправления (ICMP Redirection);
- `M` — указывает на то, что маршрут был модифицирован (протоколом маршрутизации или перенаправителем).

Так, например, установленный флаг `U` свидетельствует о том, что все маршруты в данный момент активны, флаг `G` указывает на то, что на данном маршруте используется маршрутизатор (194.15.28.12). Наконец, флаг `H` показывает, что маршрут ведет к хосту.

Кроме того, имеется поле `Interface`, в котором показана привязка сетевых интерфейсов к IP-адресам.

Для удаления маршрута из таблицы маршрутизации нужно использовать команду `route` с ключевым словом `delete`:

```
# route delete 194.15.26.0 194.15.28.12
delete net 194.15.26.0: gateway 194.15.28.12
```

Команду `route` можно использовать и в операционной системе Linux, хотя форма записи параметров и вывод результата несколько отличаются от тех, что мы видели для Solaris.

Кроме того, с помощью этой команды в Linux можно просмотреть таблицу маршрутизации, задав опцию `-n`:

```
# route -n

Kernel IP routing table

Destination   Gateway      Genmask       Flags  Metric  Ref
Use Iface

194.15.26.0   194.15.28.12 255.255.255.0 U      0       0
0   eth0

127.0.0.0     0.0.0.0      255.0.0.0     U      0       0
0   lo
```

Из результата выполнения можно сделать вывод, что команда `route` в Linux предоставляет ту же информацию, что и `netstat` в Solaris. Кроме того, выводятся также некоторые дополнительные поля:

- `Ref` — отображает информацию о количестве использований каждого маршрута;
- `Use` — отображает количество пакетов, прошедших по этому маршруту. Если, например, запустить программу `ping`, которая отправит 5 пакетов, значение счетчика станет равным 5 (при условии, что никто больше не использует этот маршрут).

Последнее поле `Interface`, как и в Solaris, отображает имя сетевого интерфейса.

Чтобы добавить маршрут к сети 194.15.26.0 через маршрутизатор 194.15.28.12, в Linux следует выполнить команду

```
route add -net 194.15.26.0 netmask 255.255.255.0 gw
194.15.28.12 1
```

При загрузке большинства операционных систем необходимо, чтобы маршруты устанавливались автоматически. Если в системе используется статическая маршрутизация, то следует внести соответствующие изменения в стартовые командные файлы (/etc/rc.local в Red Hat Linux и /etc/init.d/inetinit в Solaris). Обычно командные строки дописывают в конец этих файлов.

Динамическая маршрутизация используется для передачи информации между маршрутизаторами, которая включает данные о сетях, подключенных в настоящее время к каждому из них.

Динамическая маршрутизация реализуется посредством одного из протоколов динамической маршрутизации. К таким протоколам относится, в первую очередь, широко используемый RIP (Routing Information Protocol, протокол обмена информацией о маршрутизации), который присутствует практически во всех версиях TCP/IP. Кроме этого, достаточно широко применяются еще два протокола маршрутизации — OSPF (Open Shortest Path First) и BGP.

Чаще других в операционных системах UNIX используется протокол RIP, который в большинстве операционных систем запускается демоном `routed` (в Solaris — `in.routed`) во время загрузки системы. Демон `routed` используется, в основном, в сетях малого и среднего размеров. Альтернативой `routed` является `gated` — этот демон, кроме RIP, поддерживает и другие протоколы, такие как IGP и EGP.

Протокол маршрутизации отвечает за политику маршрутизации, выбирая, какие маршруты необходимо включить в таблицу маршрутизации. При обнаружении нескольких маршрутов к одному и тому же пункту назначения демон маршрутизации выбирает лучший маршрут, добавляя его в таблицу маршрутизации. Если демон определил, что канал не существует или отключен, он может удалить соответствующие маршруты или добавить альтернативные, чтобы обойти возникшую проблему.

Динамическая маршрутизация не меняет способы, с помощью которых ядро осуществляет маршрутизацию на уровне протокола IP — изменяется только способ помещения информации в таблицу маршрутизации: вместо запуска команды `route` или использования загрузочных файлов маршруты добавляются и удаляются динамически демоном маршрутизации, который работает постоянно.

Маршрутизаторы сети с динамической маршрутизацией должны обязательно поддерживать протокол RIP, иначе процесс обмена

данными между ними и демоном RIP будет невозможен, что сделает применение протокола бесполезным.

Очень важной задачей, помимо установки требуемых маршрутов, является проверка их доступности, а также поиск и устранение неисправностей. Хорошо, если поиск в таблице маршрутизации заканчивается успешно, и будет обнаружено соответствие хотя бы с одним маршрутом. Что произойдет, если маршрут по умолчанию отсутствует, и не будет найдено совпадение с указанным пунктом назначения? Для выявления таких неисправностей в подобных случаях можно использовать команды `ping`, `traceroute` и `netstat`, которые будут рассмотрены более подробно далее в этой главе.

7.6. Электронная почта и Интернет

Этот раздел посвящен одному из наиболее важных средств коммуникаций — электронной почте. Операционные системы UNIX обладают широкими возможностями по обработке сообщений электронной почты, используя три важнейших программных компонента электронной почты:

- *пользовательский агент* (Mail User Agent, MUA). MUA читает входящие сообщения, которые приходят на почтовый ящик пользователя, и отправляет исходящие сообщения транспортному агенту для доставки адресатам. Функции пользовательского агента выполняют такие известные программы, как `mail`, `elm`, `pine`, `mutt`. В настоящее время в программах этого класса широко используется стандарт, позволяющий включать в почтовые сообщения объекты мультимедиа — так называемый MIME (Multipurpose Internet Mail Extensions);
- *транспортный агент* (Mail Transfer Agent, MTA), пересылающий сообщения с одной машины на другую. MTA принимает почтовые сообщения, переданные ему или пользовательским агентом (MUA), или другим MTA, анализирует заголовок почтового сообщения и определяет, какой метод доставки ему использовать. Далее он передает сообщение соответст-

вующему агенту доставки. Помимо этого, транспортный агент должен принимать входящую почту от других транспортных агентов. Большинство транспортных агентов работает при наличии программной поддержки протокола SMTP (стандарт RFC 821). В операционных системах UNIX используется несколько транспортных агентов (`postfix`, `qmail`, `zmailer`, `smail`, `upas` и др.), но самым мощным и распространенным является `sendmail`;

- *агент доставки* (Mail Delivery Agent, MDA), принимающий сообщения от MTA и выполняющий фактическую их доставку пользователям. К агентам доставки можно отнести программу `procmail`.

В каждом конкретном случае почтовая система может содержать как все компоненты, так и часть из них, например, развитые почтовые системы имеют в своем составе все компоненты. Наиболее важным компонентом является MTA, выполняющий основную работу по передаче сообщения. Хотя транспортный агент и не осуществляет непосредственную доставку сообщения в почтовый ящик, тем не менее, он служит интерфейсом, соединяющим вместе все компоненты почтовой системы.

Файлы почтовых ящиков для операционных систем обычно хранятся в каталоге /var/spool/mail, хотя для получения почты можно выбрать и другие каталоги.

Для приема, отправки и обработки сообщений в большинстве современных почтовых систем используются протоколы SMTP, POP3 и IMAP4. Они определяют процедуры доставки, приема и обработки почтовых сообщений. Например, протокол POP3 предоставляет конечному пользователю доступ к поступившим ему электронным сообщениям. Обычно POP-клиенты при запросе пользователя на получение почты требуют ввести пароль, что повышает конфиденциальность переписки, при этом запросы программ-клиентов отслеживаются POP-сервером, реализованным как демон `pop3d`.

Для адресации электронной почты применяются два вида адресов — маршрутно-зависимые и маршрутно-независимые. Первый способ адресации предполагает, что отправителю известны

промежуточные хосты, через которые передается сообщение, для того чтобы попасть в пункт назначения. В адресе второго вида просто указывается пункт назначения.

В большинстве систем электронной почты используются так называемые интернет-адреса, не зависящие от маршрута и имеющие формат:

```
пользователь@хост
```

Здесь символ `@` отделяет имя пользователя от имени хоста.

Символы, стоящие справа от `@`, определяют домен, гарантируя однозначное описание клиента. Составные части домена разделяются точками, причем самая правая часть домена может обозначать код страны адресата (например, для Российской Федерации кодом страны является **ru**). Могут использоваться и обозначения сетей. Например, сети высших учебных заведений используют сокращения **edu**, правительственные учреждения — сокращение **gov**, коммерческие — **com** и т. д.

Для успешной доставки сообщения электронной почты адресату необходимо соблюдать правила оформления электронной почты, определенные международными стандартами (документом RFC 822). Кроме того, электронное письмо должно иметь стандартизованный почтовый электронный адрес.

Сообщение электронной почты кодируется в текстовом файле произвольной формы. При передаче нетекстовых данных (программ, графической информации) применяется перекодировка сообщений, которая выполняется соответствующими программными средствами.

7.6.1. Программа *mail*

Самой простой и самой распространенной программой подготовки и отправки почты в операционных системах UNIX является `mail`. Она относится к классу пользовательских агентов (MUA) и обеспечивает отправку и чтение почты. Данная программа работает даже в тех случаях, когда другие, более современные программы клиентов электронной почты в силу каких-

то причин отказываются функционировать. Несмотря на то, что она считается устаревшей, имеет смысл изучить принципы ее работы, поскольку функционирование данной программы более понятно и очевидно по сравнению с современными приложениями.

Кроме того, преимуществом программы `mail` является то, что ее удобно использовать в командных файлах, что часто и делается. Вот синтаксис программы:

```
mail [-o] [-s] [-w] [-t] адресат
mail [-e] [-h] [-p] [-q] [-r] [-f файл] [-F адресат ...]
```

Опции программы `mail` расшифровываются следующим образом:

- `-o` — подавляется оптимизация адреса;
- `-s` — символ перевода строки не включается в начало отправляемого сообщения;
- `-w` — при отправке сообщения удаленному пользователю программа не ожидает завершения пересылки;
- `-t` — в сообщение включается строка "`To:` *адресаты*", позволяя тем самым получателю иметь информацию обо всех адресатах сообщения.

Получатель сообщения обычно задается в форме имени пользователя, после чего выполняется отправка почты, если только не указана опция `-F`. Текст отправляемого сообщения читается со стандартного ввода, пока не будет обнаружен символ конца файла (<Ctrl>+<D>) либо не будет введена строка, состоящая из единственной точки. По завершению ввода программа `mail` добавляет сообщение к почтовому файлу каждого из адресатов.

В начало сообщения включается заголовок, состоящий из одной или нескольких строк "`From ...`", за которыми идет пустая строка (если только не была использована опция `-s`). Если во время ввода нажать клавишу прерывания, то сообщение будет записано в файл dead.letter, что позволяет отредактировать и отправить сохраненное в этом файле сообщение позже.

Если сообщение не было доставлено по каким-либо причинам, то оно возвращается отправителю с указанием информации о причине неудачи.

При приеме почтовых сообщений можно использовать следующие опции:

- `-e` — устанавливает код завершения без вывода почты (0 означает, что у пользователя имеются входящие сообщения, 1 — их отсутствие);
- `-h` — позволяет отображать заголовки без вывода текстов самих сообщений;
- `-p` — позволяет выводить тексты всех сообщений без приглашений;
- `-q` — позволяет завершить работу команды `mail` после получения прерывания. В противном случае прерывание вызывает лишь прекращение вывода текста сообщения;
- `-r` — указывает, что вывод текстов сообщений выполняется в порядке поступления сообщений;
- `-f` *файл* — в качестве почтового файла используется *файл* вместо принятого по умолчанию;
- `-F` *адресат* — вызывает переадресацию вновь поступающих сообщений указанному клиенту. Опция допустима только при отсутствии почтовых сообщений у пользователя.

Программа `mail` выводит тексты сообщений в порядке, обратном их поступлению, т. е. вначале выдаются последние из поступивших сообщений. После обработки последнего сообщения происходит выход из `mail` или выдается приглашение `?`, позволяющее считывать очередную команду со стандартного ввода.

Для обработки и просмотра почты можно использовать такие команды:

- *перевод_строки*, `+` или *n* — переход к следующему сообщению;
- `d` или `dp` — удалить сообщение и перейти к следующему. Фактическое удаление выполняется только по окончанию сеанса работы с `mail`;

- `d n` — удалить сообщение с номером *n* (сообщения нумеруются, начиная с 1, в порядке поступления) и перейти к следующему;
- `dq` — удалить сообщение и выйти из `mail`;
- `h` — отобразить заголовки ближайших к текущему сообщений;
- `h n` — отобразить заголовок сообщения с номером *n*;
- `h a` — отобразить заголовки всех сообщений из почтового файла;
- `h d` — отобразить заголовки сообщений, отмеченных к удалению;
- `p` — вновь отобразить содержимое текущего сообщения;
- `-` — отобразить предыдущее сообщение;
- `a` — отобразить сообщение, поступившее во время сеанса работы с `mail`;
- *n* — отобразить сообщение с номером *n*;
- `r [`*адресат* `...]` — ответить отправителю сообщения, после чего удалить сообщение;
- `s [`*файл* `...]` — сохранить сообщение в указанных файлах; при этом из почтового файла сообщение удаляется;
- `y [`*файл* `...]` — см. предыдущую опцию;
- `u [`*n*`]` — снять отметку об удалении с сообщения под номером *n* (по умолчанию используется номер последнего прочитанного);
- `w [`*файл* `...]` — сохранить в указанных файлах только текст (без заголовка сообщения);
- `m [`*адресат* `...]` — переслать текущее сообщение указанным адресатам;
- `q` или <Ctrl>+<D> — оставить в почтовом файле только неудаленные сообщения и завершить сеанс работы;
- `x` — оставить почтовый файл без изменения и завершить сеанс работы;

- `!` *`команда`* — выполнить команду интерпретатора shell;
- `?` — отобразить список команд.

О поступлении новых сообщений пользователь уведомляется при входе в систему, а также на протяжении всего сеанса работы. Программа `mail` допускает изменение атрибутов доступа к почтовому файлу, что позволяет обеспечить необходимый уровень защиты. Если режим доступа к почтовому файлу отличается от стандартного, файл будет сохранен, даже если станет пустым. Если первая строка почтового файла выглядит как

```
Forward to получатель
```

то предназначенная для данного пользователя почта будет пересылаться пользователю. При этом к заголовку сообщения добавляется строка `"Forwarded by..."`, что позволяет собирать почту на одной машине и получать информацию о пересылке писем. Установка и отмена режима переадресации выполняется посредством опции `-F`:

```
mail -F "список_адресатов"
mail -F ""
```

Первая из команд устанавливает переадресацию на клиентов из *`списка_адресатов`*, а вторая отменяет переадресацию (список адресатов пуст). Элементы *`списка_адресатов`* должны разделяться запятыми или пробелами, а сам список должен быть заключен в кавычки. Максимальная длина списка не должна превышать 1024 байта.

Опцию `-s` следует использовать осторожно, т. к. без включения промежуточного символа перевода строки сообщение может быть воспринято как часть заголовка сообщения, что нарушит работу команды `mail`.

Следует заметить, что от системы к системе синтаксис команды `mail` может незначительно меняться, поэтому перед началом работы с этой программой имеет смысл обратиться к соответствующему руководству или к man-страницам.

7.6.2. Программа *sendmail*

Одним из основных и наиболее распространенных средств рассылки почты в Интернете является программа `sendmail`. Программа `sendmail` по своим функциональным характеристикам является транспортным агентом (MTA), выполняющим функции интерфейса между пользовательскими агентами и агентами доставки. Кроме того, при обмене почтовыми сообщениями через Интернет программа сама является агентом доставки.

Программа `sendmail` позволяет выполнять:

- управление сообщениями, созданными пользователями;
- анализ адресов получателей сообщений;
- выбор соответствующего транспортного агента или агента доставки;
- преобразование адресов в форму, понятную агенту доставки;
- переформатирование заголовков, если это необходимо;
- передачу преобразованного сообщения агенту доставки.

При возникновении ошибок программа `sendmail` генерирует соответствующие сообщения, возвращая отправителю сообщения, которые не могут быть доставлены.

Программа `sendmail` тесно связана с программами подготовки и просмотра почтовых сообщений и позволяет организовать почтовую службу локальной сети таким образом, чтобы можно было обмениваться почтой с другими серверами почтовых служб посредством специальных шлюзов. Наконец, программу `sendmail` можно настроить для работы с различными почтовыми протоколами, такими, например, как UUCP и SMTP.

Универсальность и мощь программы `sendmail` проявляется и в том, что она может выполнять разнообразные функции, являясь вспомогательной для других почтовых программ. Например, `sendmail` может использоваться программой подготовки сообщений для отправки уже подготовленных сообщений, программой получения почты для пересылки полученной почты или самим пользователем для отправки по почте файла или сообщения.

Программа `sendmail` использует протокол SMTP, причем может функционировать и как клиент, и как сервер. Напомню, что протокол SMTP в настоящее время является стандартом для приложений, занимающихся обработкой электронной почты в Интернете. Если `sendmail` выполняет функции сервера SMTP, то он работает как процесс-демон, прослушивая TCP-порт 25, и, в случае получения сообщения, устанавливает соединение с удаленным клиентом SMTP. Как правило, таким клиентом является другая программа `sendmail`.

Программа подготовки почты на локальной машине также может использовать SMTP, при этом `sendmail` открывает программный канал для межпроцессного обмена.

Отправка сообщений электронной почты может выполняться как на удаленный хост, так и на локальный (местный) компьютер. В случае оправки сообщения на удаленный хост программа `sendmail` открывает программный канал и запускает программу рассылки, командная строка которой описана в файле конфигурации. В этом случае `sendmail` записывает заголовок и тело сообщения в программный канал.

7.6.3. World Wide Web

Как и любая современная операционная система, UNIX обеспечивает широкие возможности для работы пользователя в Интернете. Термином "Интернет" в настоящее время обозначают весьма широкий круг технологий, включающих передачу, прием и обработку данных на компьютерах, входящих как в локальные, так и глобальные сети.

Часть таких технологий мы уже рассмотрели (telnet, FTP, электронную почту) в предыдущих разделах, сейчас же обратим внимание на один из наиболее распространенных сервисов, благодаря которому Интернет и стал таким популярным — World Wide Web (WWW). В большинстве случаев, когда речь заходит об Интернете, подразумевают, как правило, именно World Wide Web. Идея WWW появилась у физиков, работающих в центре ядерных исследований CERN, — необходимо было передавать

друг другу результаты экспериментов в виде рисунков и диаграмм, а не только текстов. В начале 90-х годов прошлого столетия идея нашла свое воплощение в технологии WWW, а к настоящему времени без Интернета трудно вообще представить жизнь современного человека.

Анализ возможностей, предоставляемых Интернетом, может занять не один десяток, а иногда и не одну сотню страниц; более того, этой тематике посвящена не одна сотня книг. В данном разделе мы рассмотрим базис, на котором построено взаимодействие пользователей в Интернете, а именно протокол HTTP.

В практическом плане взаимодействие компьютеров (хостов) в Интернете осуществляется, как обычно, по схеме "клиент-сервер". Когда говорят, что пользователь подключается к Интернету, то в подавляющем большинстве случаев подразумевают установку соединения компьютера, на котором работает пользователь, с одним из серверов, предоставляющих услуги Интернета.

Запрашиваемая информация располагается обычно на Web-сервере и в простейшем варианте представляет собой одну или несколько страниц, разработанных посредством языка HTML (HyperText Markup Language). Совокупность таких HTML-страниц, а также других ресурсов (текстовых, мультимедийных файлов, баз данных и т. д.), связанных определенным образом, называют *Web-сайтом*.

Обмен данными между клиентом и Web-сервером осуществляется по протоколу HTTP (Hypertext Transfer Protocol, протокол передачи гипертекстовых файлов). Этот протокол используется на уровне приложений для распределенных информационных систем и позволяет общаться системам с разной архитектурой посредством передачи HTML-страниц. По этой причине очень часто Web-серверы называют HTTP-серверами, и эти термины используются как синонимы.

Для входа в Интернет пользователь запускает браузер (просмотрщик), после чего набирает адрес сетевого ресурса. При успешном соединении Web-сервер возвращает клиенту ответ в виде HTML-страницы. На одном Web-сервере может располагать-

ся одновременно несколько сайтов, а сами Web-серверы в подавляющем большинстве случаев реализованы в операционных системах UNIX. HTTP-сервер работает по протоколу TCP, принимая запросы на порт 80 (по умолчанию).

Каким образом осуществляется взаимодействие клиента и сервера или, что более точно, браузера и Web-сервера? Для понимания этого процесса мы рассмотрим более подробно протокол HTTP. В его основу, как и для других протоколов, положен принцип "запрос — ответ". Обмен по протоколу HTTP осуществляется посредством специальных методов (мы их рассмотрим далее), которые предписывают выполнить последовательность тех или иных действий. Иными словами, метод можно представить как команду на выполнение определенных действий. Программа-клиент устанавливает связь с обслуживающей программой-получателем (сервером) и посылает запрос серверу в формате, который включает:

- метод запроса;
- универсальный идентификатор ресурсов (Universal Resource Identifier, URI);
- версию протокола, за которой следует MIME-подобное сообщение, содержащее управляющую информацию запроса, информацию о клиенте и, иногда, тело сообщения.

В ответ сервер посылает клиенту сообщение, содержащее строку статуса (включая версию протокола и код статуса), за которой следует сообщение, схожее по структуре с описанным в спецификации MIME (Multipurpose Internet Mail Extension, многоцелевое расширение почты Интернета). Данное сообщение включает в себя информацию о сервере, информацию о содержании ответа и само тело ответа. Следует отметить, что одна программа может быть как клиентом, так и сервером.

Как уже упоминалось, для выполнения запроса к серверу используется программа, имеющая название *браузер*. В настоящее время существует достаточно много программ данного класса, предназначенных для работы в операционных системах UNIX (Netscape, Mozilla, Lynx и т. д.). Несмотря на различие в деталях

реализации, функционально эти браузеры похожи друг на друга. Это означает, что если вы хотите перейти с одного браузера на другой, то никаких сложностей при этом не возникнет.

Функционирование браузера основано на использовании так называемого *гипертекста* (hypertext). В основе языка гипертекста лежит концепция ссылки (link), о которой было упомянуто ранее. Ссылки выделяются либо цветом (обычно синим), либо подчеркиванием (например, как здесь) и указывают на определенный ресурс (например, на отдельный документ или на его часть). Если щелкнуть мышью на ссылке, то пользователь попадает на указанный ресурс (если, конечно, ссылка является правильной). Для определения названия ресурса, на который указывает ссылка, достаточно поместить курсор мыши поверх ссылки.

Ссылки формируются на основе URL-адресов, представляющих собой определенную комбинацию IP-адреса компьютера и пути к ресурсу, находящемуся в его файловой системе.

Вот примеры URL-адресов:

- **http://csep10.phys.utk.edu/webcourse/browser/textfile.html** — ссылка на HTML-страницу;
- **http://www.techcorps.org.org/webcourse/browser/usa2.gif** — ссылка на рисунок;
- **http://www.techcorps.org/webcourse/browser/goldgate.mpg** — ссылка на видеофайл.

Процедуры настройки доступа в Интернет в разных реализациях операционной системы UNIX приблизительно одинаковы и включают несколько шагов:

1. Настройка коммуникационного оборудования (модема, сетевого устройства и т. д.). На этом шаге требуется установить драйвер устройства (если он еще не установлен) и указать параметры обмена данными (тип обмена, скорость, опции инициализации и т. д.). Например, при использовании модема для подключения к Интернету нужно, как правило, указать скорость обмена, наличие/отсутствие контроля четности, способы синхронизации. Если система при инсталляции ус-

тановила драйвер модема, то лучше при первоначальной настройке никаких изменений не проводить, а сделать это позже, когда соединение будет гарантированно работать. Если подключение к Интернету осуществляется посредством сетевой карты, то ее нужно настроить с помощью программы `ifconfig` или из графической консоли, выбрав соответствующие опции.

2. Настройка доступа к серверу провайдера услуг Интернета.
   В большинстве случаев сервер провайдера выполняет автоматическое конфигурирование клиента (присвоение динамического IP-адреса, установку атрибутов безопасности и т. д.), так что от клиента требуется минимум настроек. Если, например, подключение осуществляется посредством модема, то обычно необходимо указать номер дозвона, имя и/или IP-адрес сервера Интернета, а также атрибуты учетной записи клиента (имя и пароль). Если же используется соединение через сетевой интерфейс, то номер дозвона не понадобится. Более сложная настройка потребуется, если используется туннеллирование или иные способы повышения безопасности соединения.

7.7. Сетевые интерфейсы

Первое, что нужно сделать при настройке сети в UNIX — сконфигурировать сетевой интерфейс. Напомню, что под *сетевым интерфейсом* в UNIX-системах понимают аппаратно-программный интерфейс, включающий как оборудование сети (сетевой адаптер или иное устройство), так и программное обеспечение (драйверы и т. д.), которое управляет этим оборудованием.

Во всех реализациях операционной системы для настройки сетевых интерфейсов применяется утилита `ifconfig`. Утилита имеет множество опций, отличающихся в различных операционных системах, и используется на этапе загрузки для настройки интерфейсов или, в случае необходимости, для отладки и диагностики неисправностей.

В общем виде синтаксис утилиты таков:

```
ifconfig интерфейс [семейство_адресов] опции | адрес ...
```

Заданная без параметров, программа выдает информацию о состоянии активных интерфейсов. Если указан один параметр *интерфейс*, отображается информация только о состоянии этого интерфейса; если указан один параметр `-a`, выдается информация о состоянии всех интерфейсов, даже если они в данный момент отключены.

Синтаксис утилиты `ifconfig` различен для разных версий UNIX, хотя эти различия и незначительны. Вот синтаксис для FreeBSD:

```
ifconfig интерфейс inet IP-адрес netmask маска
```

В Solaris утилита `ifconfig` задается с такими опциями:

```
ifconfig интерфейс inet IP-адрес/длина_префикса
```

В операционной системе Linux допустимы следующие опции команды `ifconfig`:

```
ifconfig интерфейс inet IP-адрес netmask маска broadcast широковещательный_адрес
```

Смысл наиболее часто используемых опций, допустимых во всех реализациях UNIX, приведен далее:

- *интерфейс* — имя интерфейса. В этом поле обычно указывают имя драйвера, за которым идет номер устройства, например, `eth0` для первого интерфейса Ethernet;
- `up` — позволяет активизировать интерфейс. Опция задается неявно при присвоении адреса интерфейсу;
- `down` — позволяет остановить работу драйвера для данного интерфейса;
- `mtu N` — этот параметр устанавливает максимальный размер пакета (Maximum Transfer Unit, MTU) для интерфейса;
- *адрес* — IP-адрес, присваиваемый *интерфейсу*.

Список интерфейсов, их текущие настройки и статус выводятся по команде `ifconfig`, введенной без параметров (в Solaris — с ключом `-a`).

Рассмотрим несколько примеров использования команды `ifconfig`.

Вот как может выглядеть информация о сетевых интерфейсах для операционной системы Linux:

```
# ifconfig

eth0      Link encap:Ethernet  HWaddr 00:10:4B:D1:D2:B4

          inet addr:194.15.28.10  Bcast:194.15.28.255
          Mask:255.255.255.0

          UP BROADCAST MULTICAST  MTU:1500  Metric:1

          RX packets:0 errors:0 dropped:0 overruns:0 frame:0

          TX packets:4 errors:0 dropped:0 overruns:0 car
          rier:4

          collisions:0 txqueuelen:100

          RX bytes:0 (0.0 b)  TX bytes:240 (240.0 b)

          Interrupt:11 Base address:0xc000

lo        Link encap:Local Loopback

          inet addr:127.0.0.1  Mask:255.0.0.0

          UP LOOPBACK RUNNING  MTU:16436  Metric:1

          RX packets:13055 errors:0 dropped:0 overruns:
          0 frame:0

          TX packets:13055 errors:0 dropped:0 overruns:
          0 carrier:0

          collisions:0 txqueuelen:0

          RX bytes:890244 (869.3 Kb)  TX bytes:890244
          (869.3 Kb)
```

Для получения информации по одному интерфейсу следует указать его имя в командной строке, как в этом примере:

```
# ifconfig eth1

eth1      Link encap:Ethernet  HWaddr 00:0D:87:08:43:64

          inet addr:194.15.28.11  Bcast:194.15.28.255
          Mask:255.255.255.0
```

```
UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:4 errors:0 dropped:0 overruns:
0 carrier:0
collisions:0 txqueuelen:100
RX bytes:0 (0.0 b)  TX bytes:240 (240.0 b)
Interrupt:10 Base address:0x2000
```

В операционной системе Solaris для получения информации о сетевых интерфейсах нужно выполнить команду `ifconfig -a`:

```
# ifconfig -a
lo0:
flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL>
mtu
     8232 index 1 inet 127.0.0.1 netmask ff000000
rtls0: flags=1000803<UP,BROADCAST,MULTICAST,IPv4> mtu 1500
index 2
       inet 194.15.28.10 netmask ffffff00 broadcast
       194.15.28.255
       ether 0:d:87:8:43:64
```

Каждый интерфейс системы должен иметь свой IP-адрес, который может быть присвоен такой командой:

```
# ifconfig eth1 194.15.28.10
```

В этом примере интерфейсу `eth1` присвоен адрес `194.15.28.10`. После установки параметров интерфейсов следует перезапустить демон `inetd` или перезагрузить систему. Проверку адреса можно выполнить с помощью команды `ping`, указав IP-адрес сетевого интерфейса:

```
# ping 194.15.28.10
PING 194.15.28.10 (194.15.28.10) 56(84) bytes of data.
64 bytes from 194.15.28.10: icmp_seq=1 ttl=64 time=0.039 ms
64 bytes from 194.15.28.10: icmp_seq=2 ttl=64 time=0.019 ms
```

Опции команды `up` и `down` позволяют запустить или остановить интерфейс, например:

```
# ifconfig eth1 up
# ifconfig eth0 down
```

В операционной системе Solaris для конфигурирования сетевого интерфейса необходимо указывать команду `ifconfig` с дополнительной опцией `plumb`, например:

```
# ifconfig eri0 plumb
```

Указанная опция позволяет связать сетевой интерфейс с устройством и инициализировать потоковые модули, требуемые для обмена данными по IP-протоколу. Для полной активизации сетевого интерфейса в Solaris следует ввести команду:

```
# ifconfig eri0 up plumb
```

Для удаления сетевого интерфейса в Solaris и закрытия устройства, связанного с данным интерфейсом, можно использовать опцию `unplumb`:

```
# ifconfig eri0 up unplumb
```

Особенности функционирования программы `ifconfig` довольно подробно описаны в документации на операционные системы, а также в man-страницах.

7.8. Статистика работы сети

Контроль функционирования сети и сетевых интерфейсов в операционной системе UNIX осуществляется с помощью утилиты `netstat`. В разных модификациях она присутствует во всех реализациях UNIX, позволяя отобразить содержимое различных структур данных, связанных с сетью, в разных форматах в зависимости от указанных опций.

Если утилита используется в форме

```
netstat [-Aan] [-f семейство_адресов]
        [-I интерфейс] [-p имя_протокола] [система] [core]
```

то она отображает список активных сокетов для каждого протокола.

Вторая форма программы `netstat`

```
netstat [-n] [-s] [-i | -r]
        [-f семейство_адресов]
        [-I интерфейс] [-p имя_протокола] [система] [core]
```

выбирает одну из нескольких других сетевых структур данных и отображает их на консоли.

Третья форма

```
netstat [-n] [-I интерфейс] интервал [система] [core]
```

показывает динамическую статистику пересылки пакетов по сконфигурированным сетевым интерфейсам. Аргумент *интервал* определяет количество секунд, в течение которых выполняется сбор информации между двумя последовательными выводами результата. Значение по умолчанию для аргумента *система* — /unix; для аргумента `core` в качестве значения по умолчанию используется /dev/kmem.

Вот смысл опций команды `netstat`:

- `-a` — позволяет отобразить состояние всех сокетов, кроме тех, что используются серверами;
- `-A` — позволяет отобразить адреса любых управляющих блоков протокола, связанных с сокетами, и предназначена, в основном, для отладки;
- `-i` — позволяет отобразить состояние автоматически сконфигурированных интерфейсов. При этом состояние интерфейсов, статически сконфигурированных в системе, но не обнаруженных во время загрузки, не выводится;
- `-n` — позволяет отображать сетевые адреса как числа вместо их символьного представления, используемого по умолчанию. Опция может быть указана с любым форматом вывода;

- `-r` — позволяет вывести таблицы маршрутизации. Заданная вместе с опцией `-s`, показывает статистику маршрутизации;
- `-s` — отображает статистическую информацию по протоколам. Заданная вместе с опцией `-r`, показывает статистику маршрутизации;
- `-f` *`семейство_адресов`* — ограничивает вывод статистики или адресов управляющих блоков только указанным семейством адресов:
  - `inet` — для семейства адресов `AF_INET`;
  - `unix` — для семейства адресов `AF_UNIX`;
- `-I` *`интерфейс`* — позволяет выводить информацию об указанном сетевом интерфейсе в отдельной колонке;
- `-p` *`имя_протокола`* — позволяет ограничить вывод статистики или адресов управляющих блоков только протоколом с указанным именем, например, `tcp`.

Команда `netstat` для каждого активного сокета выдает сведения о протоколе, размере очередей приема, локальном и удаленном адресах, а также состоянии протокола.

Символьный формат обычно применяется для отображения адресов сокетов и выводится в форме:

`хост.порт`

если имя хоста указано, либо:

`сеть.порт`

если адрес сокета задан сетью, а не конкретным хостом. Имена хостов и сетей берутся из соответствующих записей в файле /etc/hosts или /etc/networks.

Если имя сети или хоста неизвестно (или если задана опция `-n`), адрес отображается в числовом виде, при этом не указанные адреса и порты выводятся в виде символа `*`. Состояния сокетов кодируются следующими обозначениями:

- `CLOSED` — сокет закрыт;
- `LISTEN` — ожидание входящих соединений;
- `SYN_SENT` — попытка установить соединение;

- SYN_RECEIVED — выполняется начальная синхронизация соединения;
- ESTABLISHED — соединение установлено;
- CLOSE_WAIT — удаленная сторона отключилась, ожидание закрытия сокета;
- TIME_WAIT — ожидание после закрытия и подтверждения отключения удаленной стороны.

Вывод результатов во второй форме команды netstat зависит от выбора опции: -i или -r. Если указаны обе опции, netstat использует -i. В этом случае отображаются все имеющиеся маршруты и статус каждого из них. Каждый маршрут состоит из целевого хоста или сети и шлюза, который используется для пересылки пакетов. Столбец Flg (флаги) показывает статус маршрута.

Вот примеры использования команды netstat в операционной системе Linux:

```
# netstat -i

Kernel Interface table

Iface MTU Met RX-OK RX-ERR RX-DRP RX-OVR  TX-OK TX-ERR TX-DRP
TX-OVR Flg

eth0  1500 0   0     0      0      0       9      0      0
0    BMU

eth1  1500 0   0     0      0      0       17     0      0
0    BMRU

lo   16408 0   91    0      0      0      988     0      0
0    LRU
```

Информацию о прослушивающих сокетах можно получить, задав опцию -l:

```
# netstat -l

Active Internet connections (only servers)

Proto Recv-Q Send-Q Local Address   Foreign Address    State

tcp        0      0 *:32768         *:*                LISTEN

tcp        0      0 yuryhost:32769  *:*                LISTEN
```

```
tcp        0      0 *:imaps         *:*                 LISTEN
tcp        0      0 *:pop3s         *:*                 LISTEN
tcp        0      0 *:time          *:*                 LISTEN
```

Как видно из этих примеров, `netstat` является очень информативной командой и позволяет получить довольно подробную статистику по используемым протоколам и соединениям.

7.9. Диагностика сети и поиск неисправностей

С необходимостью проверки работоспособности сетевых настроек сталкивается любой пользователь, чей компьютер подключен к сети. Современные операционные системы включают, как правило, развитые программы диагностики с графическим интерфейсом, позволяющие решить многие проблемы, связанные с сетью.

Тем не менее, стандартные утилиты UNIX для проверки сетей, такие как `ping` и `traceroute`, позволяют эффективно выполнить большинство проверок.

Программа `ping` выполняет проверку доступности удаленного хоста, посылая специальные пакеты `ECHO_REQUEST` протокола ICMP и ожидая от них ответа. Команда реализована в двух вариантах:

```
/usr/sbin/ping хост [таймаут]
/usr/sbin/ping -s [-drvRlfnq] [-i время_ожидания] [-p шаблон]
                хост [размер_данных [кол_пакетов]]
```

Алгоритм работы утилиты `ping` таков: команда посылает дейтаграмму (пакет) `ECHO_REQUEST`. В течение определенного интервала времени команда ожидает ответа "`ICMP ECHO_RESPONSE`" от указанного хоста или сетевого шлюза. Если хост отвечает, `ping` выдает сообщение о его доступности (`host is alive`) в стандартный выходной поток и завершает работу. Если по истече-

нию определенного интервала времени (таймаута) хост не ответил, ping выдает сообщение о его недоступности (no answer from host). Стандартное значение таймаута равно 20 секундам.

Если указана опция -s, утилита ping посылает дейтаграмму каждую секунду и отображает одну строку вывода для каждого полученного ответа ECHO_RESPONSE. В этом случае ping вычисляет времена обхода (round trip times) и статистику потери пакетов. После завершения или по истечении тайм-аута команда отображает итоговую информацию.

Если указана необязательная опция *кол_пакетов*, то ping посылает указанное количество запросов хосту или шлюзу, после чего прекращает работу. Если количество пакетов не указано, команда будет выполняться бесконечно и прервать ее можно клавишей <Delete>.

Стандартный размер посылаемого пакета дейтаграммы равен 64 байтам, но можно изменить его, указав параметр *размер_данных*. Следует учитывать, что ping автоматически добавляет 8-байтовый заголовок к каждой посылаемой дейтаграмме, поэтому размер пакета, отображаемый при использовании опции -s с аргументом *размер_данных*, всегда будет на 8 байтов больше, чем указанное значение.

Перед тем как проверить работоспособность удаленного хоста, следует убедиться в работоспособности сетевого интерфейса на локальной машине, для чего выполнить команду ping, указав IP-адрес сетевого интерфейса на локальном хосте.

Рассмотрим более подробно опции команды ping:

- -d — режим отладки. Поставщику передается опция SO_DEBUG;
- -f — лавинный ping. Выдает пакеты сразу после возвращения, причем для каждого посланного ECHO_REQUEST печатается точка (.), а для каждого полученного ECHO_REPLY печатается забой (backspace). Это позволяет быстро оценить процент потери пакетов, но при этом существенно увеличивается загрузка сети, поэтому использовать ее надо осторожно;

- `-i` *`время_ожидания`* — устанавливает время ожидания в секундах между посылками пакетов. По умолчанию этот интервал равен одной секунде. Эту опцию нельзя использовать с опцией `-f`;
- `-p` *`шаблон`* — указанный шаблон используется для заполнения посылаемых пакетов определенными данными. Шаблон задается, как шестнадцатеричная строка байтов длиной до 16 байтов, и повторяется для заполнения раздела данных пакета. Например, указание опции `-p ff` вызывает заполнение пакетов единицами. Опция полезна при поиске проблем сети, связанных с передаваемыми данными;
- `-q` — сокращенный вывод. Не выдается ничего, кроме суммарных строк при запуске и завершении работы;
- `-r` — не использовать обычные таблицы маршрутизации, посылая пакеты напрямую указанному хосту в подключенной сети. Если хост не находится в непосредственно подключенной сети, возвращается ошибка. Опция используется для проверки локального хоста через интерфейс, удаленный демоном маршрутизации;
- `-R` — записывать маршрут в заголовок IP-пакета. В этом случае содержимое записи маршрута выдается на консоль, если указана опция `-v`;
- `-s` — посылать дейтаграмму каждую секунду и отображать строку вывода для каждого полученного ответа `ECHO_RESPONSE` (при отсутствии ответа ничего не выдается);
- `-v` — детальный вывод. Выдает все полученные пакеты ICMP, кроме `ECHO_RESPONSE`.

В следующем примере проверяется доступность хоста **www.perl.org**:

```
# ping www.perl.org

PING x2.develooper.com (198.182.196.56) 56(84) bytes of data.

64 bytes from x2.develooper.com (63.251.223.172): icmp_seq=1
ttl=50 time=419 ms

. . .
```

```
64 bytes from x2.develooper.com (63.251.223.172): icmp_seq=9
ttl=50 time=409 ms

64 bytes from x2.develooper.com (63.251.223.172): icmp_seq=10
ttl=50 time=379 ms


--- x2.develooper.com ping statistics ---

10 packets transmitted, 10 received, 0% packet loss, time
9329ms

rtt min/avg/max/mdev = 359.982/391.913/419.979/19.345 ms
```

Хочу заметить, что стандартный вариант команды `ping` имеет весьма ограниченные возможности по проверке доступности или существования удаленного хоста. Команда использует протокол ICMP, а этого часто бывает недостаточно, чтобы определить доступность хоста для работы, например, с протоколом UDP.

Удаленный компьютер может быть настроен на работу только с TCP, UDP или иными протоколами, поэтому в таких случаях следует применять другие средства для проверки работоспособности сетевых компьютеров.

Для тестирования сетей, особенно при возникновении трудно разрешимых проблем, полезной может оказаться утилита `traceroute`. С ее помощью можно выявить последовательность шлюзов, через которые проходит IP-пакет на пути к пункту своего назначения.

Команда допускает много опций, большинство из которых используется редко. В простейшем случае `traceroute` указывается с одним параметром:

```
traceroute имя_машины
```

Параметр *имя_машины* указывается в символическом или в числовом формате, а выводимая на экран дисплея информация представляет список хостов, через который проходит пакет, начиная с первого шлюза и заканчивая пунктом назначения. Здесь же выводится и полное время прохождения каждого шлюза.

Команда `traceroute` устанавливает поле "времени жизни" (TTL, Time-To-Live) исходящего пакета так, чтобы это время истекало до достижения пакетом пункта назначения. По истечению времени жизни текущий шлюз отправит сообщение об ошибке на машину-источник. Каждое приращение поля времени жизни позволяет пакету пройти на один шлюз дальше.

`traceroute` посылает для каждого значения поля времени жизни три пакета. Если промежуточный шлюз распределяет трафик по нескольким маршрутам, то пакеты могут возвращаться разными хостами, и их значения будут выведены на консоль. Некоторые системы не посылают уведомлений о пакетах, время жизни которых истекло, другие посылают уведомления, поступающие на машину-источник после того, как истекло время их ожидания командой `traceroute`. Такие шлюзы обозначаются рядом символов `*`. Если конкретный шлюз неизвестен, команда `traceroute` в большинстве случаев все же определит следующие за ним узлы маршрута.

Вот пример работы команды `traceroute`:

```
# traceroute kernig

traceroute to kernig (192.252.13.65), 30 hops max, 40 byte
packets

1 richi (140.252.13.35)  20 ms  10 ms  10 ms

2 kernig (140.252.13.65)  110 ms  110 ms  110 ms
```

Первая строка без номера содержит имя и IP-адрес пункта назначения и указывает на то, что величина TTL не может быть больше 30, размер дейтаграммы установлен в 40 байтов, из которых 20 байтов отводится на IP-заголовок, 8 байтов на UDP-заголовок и 12 байтов на пользовательские данные. Следует отметить, что в 12 байтах пользовательских данных сохраняется номер последовательности, который увеличивается на единицу при отправке следующей дейтаграммы, копия исходящего TTL и время, когда дейтаграмма была отправлена.

В следующих двух строках вывод начинается с TTL, затем следует имя хоста или маршрутизатора и их IP-адреса. Для каждого значения TTL отправляются 3 дейтаграммы, для каждого воз-

вращенного ICMP-сообщения рассчитывается и отображается время возврата (round-trip). При отсутствии ответа в течение пяти секунд на любую из трех дейтаграмм, выводится звездочка, затем отправляется следующая дейтаграмма. В данном примере первые три дейтаграммы имели TTL, установленные в единицу, а ICMP-сообщения вернулись через 20, 10 и 10 миллисекунд.

Следующие три дейтаграммы были отправлены с TTL, равным 2, а ICMP-сообщения вернулись с задержкой 110 миллисекунд. Поскольку TTL со значением 2 достигло конечного пункта назначения, программа прекратила свою работу.

Команда `traceroute` имеет некоторые особенности, которые следует учитывать при диагностике сетей:

- не существует гарантии, что маршрут, который используется в данный момент, будет использоваться и дальше, даже если две последовательные дейтаграммы были отправлены к одному и тому же пункту назначения;
- не существует гарантии того, что путь, по которому вернется ICMP-сообщение, совпадет с путем, по которому `traceroute` отправила дейтаграмму. Это означает, что время возврата, отображенное программой, может не совпадать со временем, необходимым для передачи исходящей дейтаграммы и возвращения сообщения. Возможен вариант, что UDP-дейтаграмма дойдет от источника до маршрутизатора за 1 секунду, однако ICMP-сообщение проделает обратный путь за 3 секунды, при этом время возврата будет напечатано как 4 секунды.

Кроме рассмотренных программ, в различных версиях операционных систем UNIX имеются собственные утилиты для настройки и диагностики сетей, такие, например, как команды `sysctl` (FreeBSD, Linux) и `ndd` (Solaris).

В приведенных далее примерах первая команда выводит значение параметра, отвечающего за ретрансляцию дейтаграмм, а вторая команда присваивает этому параметру значение 1, что означает разрешение ретрансляции.

Для Linux командная строка выглядит так:

```
sysctl net.ipv4.ip_forward
sysctl -w net.ipv4.ip_forward=1
```

а для FreeBSD принимает вид:

```
sysctl net.inet.ip.forwarding
sysctl net.inet.ip.forwarding=1
```

Те же самые операции в операционной системе Solaris можно выполнить с помощью команды `ndd`:

```
ndd /dev/ip ip_forwarding
ndd -set /dev/ip ip_forwarding 1
```

7.10. Основы программирования сетевых приложений

В этом разделе будут рассмотрены наиболее общие аспекты программирования сетевых приложений в операционных системах UNIX.

Программирование сетевых приложений — это довольно сложная и обширная тема, требующая достаточно много времени и усилий для изучения. Здесь мы рассмотрим только начальные сведения, которые, тем не менее, помогут при дальнейшем, более глубоком изучении данной темы из других источников.

Вспомним, что в основе программирования сетевых приложений лежит понятие сокета (гнезда). Сокет представляет собой объект файловой системы, обеспечивающий коммуникацию приложения с сетью. При создании сокета с ним связывается дескриптор файла, что позволяет выполнять операции чтения/записи с сокетом так же, как и с обычным файлом. Таким образом, при работе с сокетом можно применять системные вызовы `open()`, `read()`, `write()` и `close()`.

Тем не менее, для операций чтения/записи данных через сокеты лучше использовать специальные библиотечные функции, такие,

например, как `recv()` и `send()` (работают с протоколом TCP), `recvfrom()` и `sendto()` (работают с протоколом UDP).

Сетевые операции выполняются в контексте "клиент-сервер", а это означает, что в любой момент времени один из взаимодействующих процессов является клиентом, а другой — сервером. Дальнейший анализ мы будем проводить применительно к стеку протоколов TCP/IP и рассмотрим, как реализуется программный код для клиента и для сервера.

Стек протоколов TCP/IP обеспечивает взаимодействие приложений посредством предварительного установления соединения, что подразумевает надежную доставку и прием данных. Это выгодно отличает его от протокола UDP, не требующего установления соединения, что упрощает разработку сетевых приложений, но одновременно снижает надежность, поскольку доставка данных адресату не гарантируется.

В общем виде схема взаимодействия клиента и сервера, работающих по протоколу TCP/IP, представлена на рис. 7.14.

На этом рисунке представлены программные шаблоны для реализации как сервера, так и клиента TCP. Вначале о сервере.

Первый шаг, который нужно сделать при разработке сервера, — создать сокет. Это легко выполнить посредством вызова библиотечной функции `socket()`:

```
int fdsock = socket(family, type, protocol);
```

При вызове функции `socket()` необходимо задать целый ряд параметров, смысл которых раскрывается далее:

- *`family`* — задает семейство используемых протоколов. Если используется TCP/IP стандарта IPv4 (как в нашем случае), то следует указать `AF_INET`. Константа `AF_INET`, равно как и другие, описана в файле заголовка sys/socket.h;
- *`type`* — задает тип протокола. Для сокетов TCP/IP этот параметр устанавливается как `SOCK_STREAM`. Это означает, что данные передаются и/или принимаются как непрерывный поток байтов, причем не допускается ни потеря данных, ни их дублирование;

- *protocol* — обычно устанавливается в 0, хотя можно указать одну из констант, например, IPPROTO_IP.

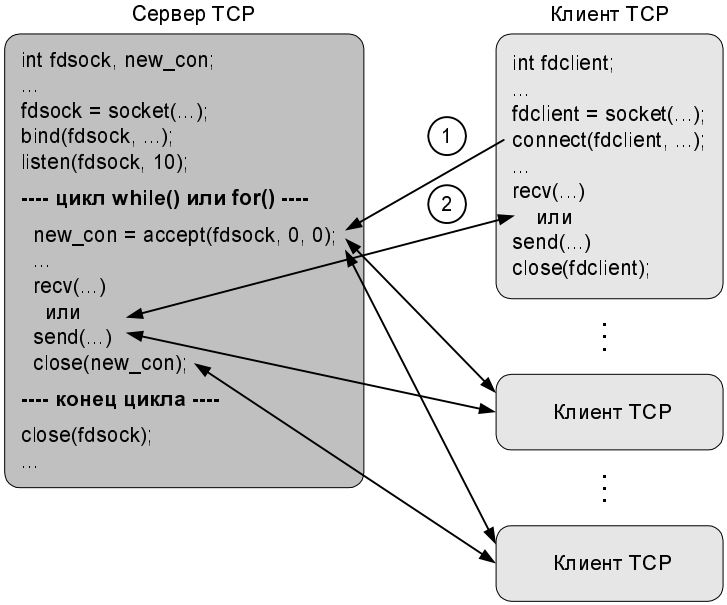


Рис. 7.14. Схема взаимодействия клиентов и сервера TCP/IP

Функция socket() в случае успешного завершения возвращает дескриптор сокета fdsock. В нашем случае для создания потокового TCP/IP-сокета стандарта IPv4 нужно задать функцию socket() со следующими параметрами:

```
int fdsock = socket(AF_INET, SOCK_STREAM, 0);
```

Созданный сокет пока нельзя использовать в операциях обмена данными: удаленный процесс не имеет возможности работать с данным сокетом, поскольку не указан его адрес и порт. Напомню, что сокет характеризуется двумя параметрами — IP-адресом и портом. Иногда для краткости говорят "адрес сокета", подразумевая при этом пару "IP-адрес — порт".

Следовательно, следующий шаг, который необходимо выполнить, — создать локальный адрес и привязать его к сокету, для чего используется функция `bind()`, имеющая синтаксис:

```
bind (fdsock, name, namelen);
```

Здесь *fdsock* — дескриптор сокета, *name* — строка байтов, содержащая IP-адрес и номер порта, которая интерпретируется используемыми протоколами соответствующим образом, а *namelen* — размер данной строки.

Следующий фрагмент программного кода демонстрирует привязку локального адреса к сокету:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in server;

fdsock = socket(AF_INET, SOCK_STREAM, 0);
bzero (&server, sizeof (server));
server.sin_family = AF_INET;
server.sin_addr.s_addr = htonl(адрес);
server.sin_port = htons(порт);
bind(fdsock, (struct sockaddr *) &server, sizeof(server));
```

Здесь в качестве второго параметра функции `bind()` указана структура типа `sockaddr_in`, поля которой должны быть заполнены соответствующими значениями. Так, поле `sin_family` должно содержать название семейства протоколов, поле `sin_addr.s_addr` должно содержать 4-байтовый IP-адрес в сетевом формате, а поле `sin_port` — значение порта в сетевом формате.

Сетевой формат следования байтов отличается от того, который используется на локальной машине, обратным порядком следования байтов. При выполнении сетевых операций всегда необходимо преобразовывать представление адреса *адрес* и порта

порт из локального в сетевой, что и выполняют функции `htonl()` и `htons()`. Первая из них преобразует длинные целые числа, а вторая короткие.

Следующий после привязки адреса шаг — установить только что созданный сокет в режим прослушивания. В этом режиме сокет определяет, какие клиенты требуют соединения с сервером. Установить сокет в режим прослушивания можно при помощи функции `listen()`, имеющей такой синтаксис:

```
listen(fdsock, число_соединений);
```

Здесь *fdsock* — дескриптор сокета, а параметр *число_соединений* определяет длину очереди, т. е. максимальное число одновременно допустимых соединений.

Хочу сделать очень важное замечание — функция `listen()` только отслеживает поступающие запросы клиентов на установление соединения, но не создает их. Второй параметр функции `listen()` определяет количество соединений, которое может обработать сервер. Применительно к схеме, показанной на рис. 7.14, таких соединений может быть 10. Иными словами, 10 клиентов TCP могут одновременно подключиться к серверу, если каждый из них использует только одно соединение.

Наш сервер TCP работает в блокирующем режиме, т. е. процесс может ожидать поступления запроса на соединение теоретически неограниченное время, блокируя работу остальной части программного кода. Такая конфигурация в "реальной жизни" используется редко — в большинстве обычно применяют неблокирующие сокеты, позволяющие выполняться другим частям программы. Перевести сокет в неблокирующий режим можно с помощью функции `fcntl()`, но при этом программный код будет несколько сложнее.

Далее программа-сервер в цикле `while` или `for` может обрабатывать поступающие соединения, вызывая функцию `accept()`:

```
new_con = accept(fdsock, 0, 0);
```

Здесь `fdsock` — дескриптор прослушивающего сокета, остальные два параметра функции `accept()` (байтовая строка с IP-адресом

и портом для вновь созданного клиентского соединения) можно установить в 0, поскольку в данной программе они нигде не используются.

Обращаю ваше внимание на то, что для вновь поступающих запросов функция `accept()` создает динамические сокеты с дескриптором `new_con`. Этим сокетам выделяются временные порты — таким образом, сервер устанавливает уникальный канал обмена данными с каждым клиентом. Дескриптор `fdsock` никогда не используется для обмена данными, его назначение — быть точкой входа для клиентских соединений, и не более, поэтому ошибкой будет чтение/запись данных через такой сокет!

После установления соединения с клиентом функцией `accept()` сервер может принимать данные от клиента или передавать их ему. После окончания обмена данными необходимо закрыть дескриптор рабочего соединения, что выполняет функция `close()`, принимающая в качестве единственного параметра дескриптор сокета.

Сервер должен постоянно обрабатывать входящие запросы, поэтому обработку соединений следует выполнять в бесконечном цикле `while()` или `for()`. Если сервер прекращает работу, то он должен закрыть все открытые дескрипторы сокетов, включая прослушивающий.

Теперь мы можем собрать шаблон нашего приложения-сервера. Вот его примерный исходный текст:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define port MYPORT
int main(void)
{
 int fdsock, new_con;
 struct sockaddr_in server;
```

```
 bzero (&server, sizeof (server));
 server.sin_family = AF_INET;
 server.sin_addr.s_addr = htonl(INADDR_ANY);
 server.sin_port = htons(MYPORT);

 fdsock = socket(AF_INET, SOCK_STREAM, 0);
 bind(fdsock, (struct sockaddr *) &server, sizeof(server));
 listen (fdsock, 10);
 while (1)
  {
   new_con = accept(fdsock, 0, 0);
   if (new_con > 0)
    {
      . . .обработка данных . . .
    }
   close(new_con);
 }
 close(fdsock);
 return 0;
}
```

Этот шаблон программного кода можно использовать для разработки простых серверов TCP. Обратите внимание на строку

```
server.sin_addr.s_addr = htonl(INADDR_ANY);
```

Здесь в качестве параметра функции `htonl` используется `INADDR_ANY`, что означает "любой адрес". Вместо `MYPORT` можно указать любое приемлемое значение для порта из допустимого диапазона. Хочу сказать, что в данном примере для упрощения листинга не анализируются возможные ошибки, но при разработке собственного приложения этот фактор следует учитывать.

Программный код приложения-клиента намного проще (см. рис. 7.14), поскольку клиенту не требуется прослушивать сокет и создавать множественные ветвления, как это необходимо для сервера. Программа-клиент создает сокет с дескриптором

fdclient при помощи функции socket(), после чего устанавливает соединение, вызвав библиотечную функцию connect() (шаг 1 на рис. 7.14). Далее выполняется прием данных от сервера или передача их ему (шаг 2 на рис. 7.14).

Функция connect() имеет такой синтаксис:

```
connect(fdclient, (struct sockaddr *)&server,
sizeof(server));
```

Здесь параметр fdclient является дескриптором сокета, а второй и третий параметры указывают имя, которое должно быть присвоено данному сокету. По завершению обмена данными клиент, так же как сервер, должен закрыть дескриптор сокета.

Вот как выглядит шаблон приложения-клиента:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define port MYPORT
int main(void)
{
 int fdclient;
 struct sockaddr_in server;

 bzero (&server, sizeof (server));
 server.sin_family = AF_INET;
 server.sin_addr.s_addr = htonl(адрес);
 server.sin_port = htons(MYPORT);

 fdsock = socket(AF_INET, SOCK_STREAM, 0);
 connect(fdclient, (struct sockaddr *) &server,
 sizeof(server));
   . . . обработка данных . . .
 close(fdclient);
 return 0;
}
```

Как видно из листинга, программный код клиента намного проще, чем сервера. Кроме того, следует учитывать, что клиент для соединения использует конкретный IP-адрес, поэтому макрос `INADDR_ANY` здесь указывать нельзя. Одним из вариантов является преобразование имени хоста, на котором работает сервер, в IP-адрес посредством функции `gethostbyname()`.

До сих пор мы не рассматривали функции приема/передачи данных через сетевые соединения TCP. Таких функций две: `recv()` и `send()`. Первая из них принимает данные от источника, находящегося в сети, а вторая — передает данные сетевому узлу. Синтаксис этих функций представлен далее:

```
send(fdsock, buf, sizeof(buf), flags);
recv(fdsock, buf, sizeof(buf), flags);
```

Здесь первый параметр *`fdsock`* является дескриптором сокета, второй параметр указывает адрес буфера памяти *`buf`*, куда должны поступать данные (при приеме) или откуда должны извлекаться данные (при передаче). Третий параметр определяет размер буфера данных, и, наконец, четвертый параметр позволяет установить дополнительные флаги, определяющие способ обработки поступающих или передаваемых данных.

Следует сказать, что вместо функций `recv()` и `send()` можно использовать функции `read()` и `write()`, как и для обычных файловых дескрипторов, хотя функции `recv()` и `send()` являются более предпочтительными, поскольку позволяют посредством дополнительных флагов управлять процессом обмена данными по сети.

Заканчивая рассмотрение примеров, хочу добавить, что мы использовали так называемые блокирующие сокеты, когда обработка последующих запросов возможна только после окончания обработки текущего. Современные серверные приложения используют иные подходы, позволяющие выполнять так называемую асинхронную обработку данных, когда обрабатывается одновременно много запросов, но данная тема здесь не рассматривается.

Глава 8

Разработка программного обеспечения в среде UNIX

Современные операционные системы UNIX имеют развитые средства разработки приложений, позволяя создавать как самые простые утилиты, работающие с текстовой консолью, так и многофункциональные приложения с графическим интерфейсом. Несмотря на такое разнообразие инструментов разработки, мы рассмотрим три наиболее популярные из них — языки программирования C/C++, Java и Perl. Такой выбор обусловлен несколькими факторами.

Язык C/C++ является классическим инструментом разработки приложений для UNIX, впрочем, как и для других операционных систем. Помимо гибкости и мощи этого языка следует сказать, что сама операционная система написана на C/C++ и включает в себя многие библиотечные функции, что облегчает разработчику задачу создания приложений — достаточно уметь применять данные функции. Кроме того, ядро операционной системы использует механизм системных вызовов, посредством которых реализован API пользователя, позволяющий создавать приложения, очень тесно взаимодействующие с операционной системой.

Немаловажным преимуществом языка C/C++ является и то, что программы, разработанные для одной операционной системы, можно без особого труда модифицировать для работы в другой,

т. е. С-код является легко переносимым. При этом говорят о переносимости на уровне исходных текстов программ — исходный текст программы на C/C++, разработанной для какой-то одной UNIX-системы, можно с минимальными изменениями откомпилировать в другой. Более того, если в программе используются только функции стандарта POSIX и стандартные библиотеки C/C++, то никаких изменений в исходных текстах вообще не понадобится.

В этой главе мы рассмотрим наиболее широко распространенный компилятор языка C++, известный под аббревиатурой "g++", входящий в состав пакета свободно распространяемого программного обеспечения для разработки приложений под названием GCC (GNU Compiler Collection). В состав дистрибутивного пакета программ GCC, кроме компилятора C++, входят компиляторы языков Java, Fortran и Ada.

Язык Java — это один из наиболее мощных инструментов для разработки как обычных приложений, так и интерактивных продуктов для Интернета.

По сравнению с другими инструментами программирования Java обладает возможностями, недоступными на сегодняшний день в других языках:

- Java позволяет создавать апплеты (applets) — небольшие, надежные, динамичные, не зависящие от платформы активные сетевые приложения, встраиваемые в Web-страницы. Апплеты легко настраиваются и распространяются так же быстро, как любые документы HTML;
- Java использует объектно-ориентированный подход к разработке приложений, что позволяет широкому кругу программистов быстро создавать новые программы и апплеты;
- Java включает большой набор классов объектов для абстрагирования многих системных функций, используемых при работе с окнами, сетью и для ввода/вывода. Ключевая черта этих классов заключается в том, что они обеспечивают создание независимых от используемой платформы абстракций для широкого спектра системных интерфейсов.

Язык Perl (Practical Extraction and Report Language) был создан американским программистом Ларри Уоллом в средине 90-х годов прошлого века. Изначально Perl задумывался как высокоуровневый кроссплатформенный язык системного программирования. С тех пор этот язык вышел далеко за пределы своего исходного предназначения, но продолжает очень широко использоваться в системном программировании как для операционных систем UNIX, так и для других платформ. Для обеспечения максимальной переносимости основное внимание уделялось открытым системам, соответствующим стандарту POSIX, который поддерживает большинство UNIX-систем.

Программистов, использующих Perl, условно можно разделить на две категории: системных администраторов, создающих программы для управления файловой системой и процессами, поиска и составления отчетов, и пользователей, разрабатывающих электронные формы для Web-серверов. Последняя категория находит Perl более удобным и легким, чем C, поскольку Perl имеет развитые возможности для манипулирования данными, включая проверку данных и операции с простыми базами данных.

В настоящее время язык Perl является основным средством администрирования UNIX, которое с успехом может использоваться вместо других традиционных средств администрирования. Интерпретатор этого языка, так же как и командный интерпретатор shell, в настоящее время является встраиваемым инструментальным средством во всех версиях операционных систем UNIX. Универсальность этого языка и способствовала его широкому распространению среди системных администраторов и программистов UNIX, учитывая и то, что он решает задачи обычно быстрее, чем другие аналогичные средства.

Следует подчеркнуть, что Perl может использоваться для системного программирования даже в системах, не соответствующих стандарту POSIX — в таких случаях вам понадобятся специализированные модули для этих систем.

Несмотря на огромную популярность языка Perl как инструмента системного администрирования, в последнее время он стал

очень популярным в области разработки интернет-приложений, таких как CGI-сценарии, системы автоматической обработки электронной почты и поддержки Web-узлов.

Одной из широко применяемых в Интернете технологий является технология CGI-сценариев, с помощью которой реализуются динамические эффекты. Сценарии могут быть разработаны с использованием любого языка программирования, но написанные на Perl получили наибольшее распространение из-за легкости создания и оптимизационных возможностей этого языка при обработке текстовых файлов. Такая популярность привела к тому, что разработчики серверов Интернета, работающих в других операционных системах, стали включать возможность подключения сценариев Perl в свои системы. В настоящее время их можно использовать и на серверах Apache, NCSA и Netscape для операционной системы UNIX.

Следует отметить, что язык Perl очень часто применяется для решения задач, связанных с автоматизацией обработки электронной почты Интернета. Сценарии Perl можно использовать для фильтрации почты на основе адреса или содержимого, автоматического создания списков рассылки и для решения многих других задач. Можно, например, написать сценарий, который обрабатывает входящую почту и добавляет сообщения на заранее созданную страницу новостей, сортируя их по соответствующим тематикам, что позволяет быстро просматривать почту, не тратя времени на чтение каждой полученной корреспонденции.

Еще одна сфера применения языка Perl — поддержка Web-узлов, представляющих собой структурированное хранилище HTML-страниц. Язык Perl оптимизирован для обработки большого количества текстовых файлов, поэтому его использование для анализа и автоматического изменения содержимого Web-узла само собой вытекает из тех задач, для решения которых он специально и создавался. Его, например, можно использовать для решения задачи проверки правильности перекрестных ссылок на страницах Web-узла, а также для проверки правильности ссылок на другие узлы.

Язык Perl обладает развитыми сетевыми возможностями, поэтому его легко можно использовать для обмена информацией на основе протоколов HTTP и FTP. Это позволяет автоматизировать получение файлов с других узлов, а в сочетании с его возможностями обработки текстовых файлов — создавать сложные информационные системы.

Рассмотрим некоторые важные практические аспекты программирования на языках C и Perl и начнем с C.

8.1. Разработка приложений на C++

Разработку программы следует начать с ввода и редактирования исходного текста, для чего подойдет любой из популярных текстовых редакторов (emacs, vi, vim и т. д.). Затем исходный текст программы нужно сохранить в файле, имеющем одно из расширений — c или cpp, при этом необходимо учесть, что все имена в операционной системе UNIX чувствительны к регистру. Например, простейшую программу, отображающую приветствие на экране и имеющую исходный текст

```
#include <stdio.h>
void main(void)
{
  printf("Hello, world!\n");
}
```

можно сохранить в файле hello.cpp, затем откомпилировать его, получив при этом исполняемый файл программы. Для этого следует выполнить команду:

```
# g++ hello.cpp
```

По этой команде компилятор g++ попытается создать исполняемый файл из файла hello.cpp, содержащего исходный текст программы. Создание исполняемого файла программы требует выполнения, как минимум, двух шагов: на первом шаге файл с исходным текстом компилируется в объектный файл, а на втором из объектного файла создается исполняемый файл про-

граммы. Процесс сборки приложения иногда называют линковкой (от англ. *link* — связывать).

В простейшем случае, который здесь представлен, исполняемый файл создается из одного объектного файла, хотя в общем случае объектных файлов может быть несколько. На каждом из описанных этапов (компиляция и сборка) выполняются определенные задачи. На этапе компиляции из исходного текста программы генерируется объектный файл, представляющий собой двоичный код инструкций языка C в форме машинных команд процессора. Во время компиляции вылавливаются ошибки, связанные с неправильным использованием функций или с отсутствием их описания в файлах заголовков, а также различного рода синтаксические ошибки и т. д.

На втором этапе выполняется сборка программы. Здесь определяются взаимосвязи между различными частями программы, а также разрешаются все ссылки на вызываемые из разных объектных файлов функции и данные и т. д. Если не указано имя исполняемого файла, то после сборки ему автоматически присваивается имя a.out. Разработчик программы имеет возможность изменить это имя на более информативное. Если во время компиляции происходят какие-то ошибки, то компилятор g++ выводит соответствующие сообщения на экран дисплея, а исполняемый файл программы, естественно, не создается.

Если сборка программы прошла успешно, то ее можно запустить из командной строки, набрав

```
# ./a.out
```

Для компиляции программы и создания исполняемого файла с именем, отличным от a.out, следует задать опцию `-o`. Например, для создания исполняемого файла hello.out можно воспользоваться командой

```
# g++ -o hello.out hello.cpp
```

В результате компилятор создаст исполняемый файл hello.out.

Иногда требуется вместо двух этапов (компиляции и сборки) выполнить только компиляцию исходного текста. Результатом

компиляции в этом случае будет объектный файл, представляющий программу в виде машинного кода, который затем можно связать с другим объектным файлом для сборки всей программы или использовать в другой программе.

Например, для выполнения только компиляции программы hello.cpp без создания исполняемого файла нужно выполнить команду

```
# g++ -c hello.cpp
```

В результате получим объектный файл hello.o (объектные файлы имеют расширение o).

Для создания исполняемого файла программы из нескольких файлов с исходными текстами нужно просто перечислить их все в командной строке, как в этом примере:

```
# g++ file1.cpp file2.cpp file3.cpp
```

Если компиляция прошла успешно, то будет создан один исполняемый файл a.out. Исполняемый файл программы можно создать из нескольких ранее созданных объектных файлов, например:

```
# g++ file1.o file2.o file3.o
```

После успешной сборки будет создан, как и в предыдущих случаях, исполняемый файл. Следующий пример демонстрирует создание исполняемого файла program.out из нескольких файлов с исходными текстами:

```
# g++ -o program.out file1.cpp file2.cpp file3.cpp
```

Здесь создается исполняемый файл program.out, использующий для сборки файлы file1.cpp, file2.cpp и file3.cpp с исходными текстами.

В процессе создания исполняемого файла кроме файлов исходных текстов могут понадобиться и файлы библиотечных функций или, по-другому, библиотеки, представляющие собой набор объектных файлов, сгруппированных вместе в один файл и проиндексированных. Библиотеки могут содержать конкретный набор функций, как стандартных, так и опреде-

ленных разработчиком. Библиотечные файлы позволяют тиражировать часто применяемые функции для использования их в других приложениях.

Присоединяемые к приложению библиотеки можно задать в командной строке в форме `-lимя_библиотеки`. Например, `-lg++` указывает на библиотеку стандартных функций C++, а `-lm` — на библиотеку, содержащую различные математические функции (синус, косинус, арктангенс и т. д.). Все библиотеки должны быть перечислены в командной строке после объектных файлов или файлов исходных текстов, содержащих вызовы библиотечных функций.

Хочу более подробно остановиться на некоторых, очень важных аспектах программирования. Первый касается разработки сетевых приложений в среде UNIX. В некоторых операционных системах, например, Linux и FreeBSD, сетевые функции включены в ядро и вызываются посредством системных вызовов (`socket()`, `bind()`, `connect()`, `listen()`, `accept()`). В таких случаях компиляция и сборка сетевых приложений выполняется как обычно:

```
# g++ -o program file.cpp
```

Здесь `file.cpp` — файл с исходным текстом, а `program` — исполняемый файл.

Если сетевое приложение разрабатывается, например, в операционной системе Solaris, то здесь ситуация иная. Сетевые функции реализованы не в ядре, а в виде библиотечных функций C++, поэтому командная строка компилятора должна выглядеть так:

```
g++ -o program file.cpp -lsocket -lnsl library
```

Здесь `program` — исполняемый файл, `file.cpp` — файл с исходным текстом, `library` — библиотека сетевых функций.

Кроме того, в исходный текст программы должны быть включены строки:

```
#include <sys/types.h>
#include <sys/socket.h>
```

Например, в операционной системе Solaris 10 командная строка может выглядеть так:

```
# g++ -o program file.cpp -lsocket -lnsl
/usr/lib/libsocket.so
```

Здесь для создания сетевого приложения `program`, исходный текст которого находится в file.cpp, используется библиотека libsocket.so (или libsocket.so.1).

Еще один момент связан с разработкой приложений, использующих программные потоки. Для того чтобы программа была успешно создана, при сборке следует указать на необходимость использования специальной библиотеки, в которой реализованы потоковые функции.

Например, в операционной системе Solaris командная строка для сборки приложения, использующего потоковые функции, будет выглядеть следующим образом:

```
# g++ -o program file.cpp -lpthread /lib/libthread.so
```

В самом исходном тексте программы должно присутствовать объявление файла заголовка:

```
#include <pthread.h>
```

Для операционных систем Linux командная строка может выглядеть так:

```
# g++ -o program file.cpp -lpthread
```

Сложные программы, разрабатываемые на C/C++, могут содержать несколько файлов, включая как файлы с исходными текстами программ (расширение cpp), так и файлы заголовков (расширение h), в которых размещено описание функций. Кроме того, для сборки всего приложения может понадобиться определенная последовательность действий, которую очень сложно, а иногда и невозможно выразить командной строкой.
В таких случаях сборка приложения выполняется посредством так называемого make-файла, содержащего перечень правил и взаимозависимостей между командами, выполняющими созда-

ние исполняемых файлов. Содержимое make-файла обрабатывается утилитой `make`.

Использование make-файла рассмотрим на следующем примере. Предположим, что для сборки приложения необходимо использовать файлы:

```
main1.cpp
mylib.cpp
mylib.h
openfile.cpp
openfile.h
```

Наш исполняемый файл назовем program.out. Для создания make-файла (назовем его program.make) можно воспользоваться одним из редакторов (vi, emacs и т. д.). Если для создания приложений выбрать отдельный каталог, то можно присвоить make-файлу имя "makefile".

Вот как может выглядеть make-файл с именем program.make:

```
# program.make — это комментарий, и он игнорируется
# утилитой make


program.out : main1.o mylib.o openfile.o
	g++ -o program.out main1.o mylib.o openfile.o


# Из двух инструкций, представленных выше, видно,
# что program.out зависит от объектных файлов main1.o,
# mylib.o и openfile.o и должна из них компоноваться.
# Для создания program.out нужно выполнить команду g++
# с необходимыми аргументами (вторая строка).
# Таким образом, исполняемый файл program.out создается
# из трех объектных файлов.


main1.o: main1.cpp openfile.h mylib.h
    g++ -c main1.cpp
```

```
# В этих двух инструкциях определяется, как создавать
# объектный файл main1.o (первая инструкция), и указана
# командная строка для компиляции (вторая строка).
# При этом компиляция выполняется только в том случае,
# если хотя бы один из файлов main1.cpp, openfile.h
# или mylib.h изменился со времени последней компиляции
# main1.o.


mylib.o: mylib.cpp mylib.h
    g++ -c mylib.cpp


# Инструкции, показанные выше, указывают на то, что файл
mylib.o зависит от mylib.cpp и mylib.h (первая инструкция)
# и должен компилироваться в соответствии с командной
# строкой, указанной во второй инструкции. При этом
# компиляция выполняется в том случае, если хотя бы один
# из файлов mylib.cpp или mylib.h изменился со времени
# последней компиляции файла mylib.o.


openfile.o: openfile.cpp openfile.h
       g++ -c openfile.cpp


# Эти инструкции показывают зависимость файла openfile.o
# от openfile.cpp и openfile.h (первая строка) и предписывают
откомпилировать файл openfile.cpp, если файлы openfile.cpp
# и openfile.h изменились со времени последней компиляции
# openfile.o.
clean:
&nbps    rm *.o program.out


# Здесь последние две инструкции указывают правила очистки
# рабочего каталога приложения от ненужных файлов
# (например, при необходимости полной перекомпиляции
# приложения). Из второй инструкции видно, что в процессе
# очистки должны быть удалены все объектные файлы
# и исполняемый файл.
```

После создания make-файла для формирования приложения можно выполнить командную строку

```
# make -f program.make
```

Если все объектные файлы, указанные в make-файле, были созданы успешно, то в результате получим программу program.out.

Выполнить программу можно стандартным образом:

```
# ./program.out
```

Следует отметить, что если хотя бы один из файлов (cpp или h) изменился, то нужно опять выполнить командную строку

```
# make -f program.make
```

Если нужно полностью перекомпилировать приложение, то следует выполнить команду

```
# make -f program.make clean
```

Таким образом, процесс создания исполняемого файла программы с помощью make-файла и утилиты `make` можно представить последовательностью шагов:

- `# emacs program.make` — создаем файл program.make с помощью редактора `emacs`;
- `# make -f program.make` — создаем программу program.out.

Если для разработки приложения используется отдельный каталог, то можно использовать make-файл с именем makefile. Тогда команды для создания исполняемого файла несколько изменятся:

- `# emacs makefile` — создаем файл с именем makefile с помощью редактора `emacs`;
- `# make` — создаем программу program.out — утилита `make`, заданная без параметров, использует по умолчанию файл makefile.

Для очистки рабочего каталога от ненужных объектных файлов следует выполнить команду

```
# make clean
```

Вот еще один пример использования make-файлов для создания приложений:

```
# account.make — это комментарий, и он игнорируется утилитой
make.

# В этом примере создается исполняемый файл account.
# Для сборки приложения используются следующие файлы:

#   исходных текстов: account.cpp и accountmain.cpp

#   заголовков: account.h.

#

account: account.o accountmain.o

       g++ -o account account.o accountmain.o


account.o: account.cpp account.h

       g++ -c account.cpp


accountmain.o: accountmain.cpp account.h

       g++ -c accountmain.cpp


clean:

       rm account account.o accountmain.o
```

На этом рассмотрение возможностей компилятора g++ можно закончить. Дополнительную информацию о компиляторе g++ можно найти на **http://gcc.gnu.org/onlinedocs**.

8.2. Java

Язык Java является объектно-ориентированным языком программирования, с первого взгляда напоминающим усеченную

версию C++. На самом деле он является очень мощным и в то же время достаточно простым. Язык был разработан компанией Sun Microsystems в средине 90-х годов прошлого столетия и быстро завоевал популярность благодаря возможности быстрой разработки высококачественных Web-приложений. Фактически этот язык в настоящее время является стандартом для разработки интернет-программ.

Такая популярность обусловлена несколькими причинами:

- наличием C-подобного синтаксиса, облегчающего изучение Java многими профессиональными программистами, пишущими на языке C/C++;
- возможностью компиляции в промежуточный код (так называемый байт-код), который при выполнении интерпретируется специальной программой. Байт-код может интерпретироваться в любой системе, в которой есть среда времени выполнения Java, часто называемая JRE (Java Run-Time Environment). Многие системы программирования, в которых пытались обеспечить независимость от платформы, обладают существенным недостатком — потерей производительности. В языке Java, хотя и используется интерпретатор, тем не менее, байт-код легко переводится непосредственно в соответствующий код процессора данной платформы практически без потери производительности. Многие браузеры имеют встроенные компиляторы байт-кода, например, Netscape;
- высокой безопасностью и надежностью. Один из ключевых принципов разработки языка Java состоял в обеспечении защиты от несанкционированного доступа к данным. Программы, разработанные с использованием Java, не могут вызывать системные функции и получать доступ к произвольным системным ресурсам, что обеспечивает уровень безопасности, недоступный в других языках;
- большому набору модулей прикладного интерфейса программирования (API-модули), которые могут быть использованы при решении весьма широкого круга задач;

- надежностью — Java позволяет во многих случаях избегать и обнаруживать ошибки на ранних этапах разработки программы, одновременно исключая многие источники ошибок, свойственные другим языкам программирования (например, строгую типизацию, как в языке C). Характерными ситуациями для многих программ являются ошибки выделения памяти или исключительные ситуации, приводящие к сбою приложения. В языке Java эти проблемы решены, поскольку данная среда разработки использует так называемую "сборку мусора" (garbage collection) для освобождения незанятой памяти, а также встроенные средства обработки исключительных ситуаций;
- простотой изучения — язык Java, являясь более сложным, чем языки командных интерпретаторов, все же проще для изучения, чем другие языки программирования, например, тот же C++.

Что же происходит, когда на удаленной системе есть браузер, поддерживающий Java? Поскольку бóльшая часть программного кода Java-программы реализована с использованием классов API, то для запуска такой программы на удаленной системе требуется лишь поддержка API, что, как правило, имеет место.
В настоящее время разработано достаточно много компиляторов Java, причем не только для работы в среде UNIX, но и в других системах, например, Windows. Корпорация Microsoft включает компиляторы Java в линейку наиболее мощной среды разработки для данных систем — Visual Studio .NET.

Рассмотрим более подробно особенности работы с Java. Ввиду схожести языков Java и C++ проанализируем их основные отличия.

В Java используются практически идентичные с языком C++ соглашения для объявления переменных, передачи параметров, операторов и для управления потоком выполнения кода. В языке Java, в отличие от C++, объектно-ориентированный подход используется гораздо шире. С помощью компилятора C++ мож-

но оттранслировать и выполнить любую программу, написанную на C. В отличие от этого, все Java-приложения состоят исключительно из классов и методов, использующих эти классы.
В Java, в отличие от C++, не может быть функций, объявленных за пределами класса. Объектная модель, используемая в Java, проста и легко расширяема, хотя для повышения производительности числа и другие простые типы данных Java не являются объектами.

Еще один аспект касается использования глобальных переменных. При использовании глобальных переменных любая функция может изменить состояние программы, что может привести к краху. В Java единственным глобальным пространством имен является иерархия классов — здесь невозможно создать глобальную переменную, не принадлежащую ни одному из классов.

Очень часто источник ошибок заключается в неправильном использовании указателей. Сами по себе указатели являются очень мощным средством для написания эффективных программ на C++, но в то же время они выступают источником трудно предсказуемых ошибок. К типичным ошибкам относится, например, запись данных за пределы массива, что может привести к разрушению кода программы. В языке Java дескрипторы объектов также реализованы в виде указателей, тем не менее, здесь отсутствует возможность непосредственной манипуляции указателями — нельзя преобразовать целое число в указатель или обратиться к произвольному адресу памяти.

Существенные сложности в языке C++ связаны с распределением памяти. После выделения памяти одной из функций (`malloc()`, `HeapAlloc()` и т. д.) перед завершением программы обязательно следует освобождать выделенную область. Если программа довольно сложная, можно легко ошибиться, попытавшись, например, освободить ранее освобожденный блок памяти или забыть освободить память вообще. В языке Java каждая сложная структура данных представляет собой объект, память под который резервируется из кучи процесса с помощью оператора `new`. Освобождение памяти в Java берет на себя так назы-

ваемый "сборщик мусора", работающий в фоновом режиме. Сборщик мусора может запускаться немедленно, если Java не может удовлетворить запрос на выделение памяти.

Далее мы вкратце рассмотрим основные элементы языка Java и проанализируем несколько примеров программного кода.

8.2.1. Первая программа на Java

Как обычно, в начале анализа возможностей того или иного языка приводится исходный текст простейшей программы наподобие "Hello, world". В языке Java исходный текст такой программы может выглядеть так:

```
class HelloWorld {
   public static void main (String args []) {
    System.out.println("Hello, world");
   }
}
```

Как видно из примера, программный код должен размещаться внутри класса или классов (далее мы рассмотрим это более подробно). Исходный текст программы нужно сохранить в файле HelloWorld.java.

Обращаю ваше внимание на то, что имя класса (`HelloWorld`) и имя файла должны обязательно совпадать (регистр литер имеет значение!), а расширение файла должно быть java.

Для трансляции этого примера необходимо запустить транслятор Java с именем `javac`, указав в качестве единственного параметра имя файла с исходным текстом программы:

```
# javac HelloWorld.java
```

Если трансляция выполнена без ошибок, то в результате будет создан файл HelloWorld.class с машинно-независимым байт-кодом. Транслятор Java предполагает, что исходный текст находится в файле с расширением java, а полученный в результате

трансляции байт-код для каждого класса сохраняется в отдельном файле, имя которого совпадает с именем класса. Такому файлу автоматически присваивается расширение class.

Выполнить полученный байт-код можно только в том случае, если в системе установлена среда времени выполнения Java (в большинстве операционных систем UNIX такая среда устанавливается при инсталляции системы). Если среда выполнения по какой-либо причине отсутствует, то нужно предварительно ее установить. Для операционных систем UNIX наиболее часто используются программные пакеты фирмы Sun Microsystems, хотя можно применить и программные продукты других производителей.

Для выполнения оттранслированного файла нужно ввести следующую командную строку:

```
# java HelloWorld
Hello, world
```

Обратите внимание, что при вводе командной строки расширение файла указывать необязательно.

Проанализируем более подробно исходный текст этой программы.

В строке

```
class HelloWorld {
```

используется зарезервированное слово `class`, описывающее экземпляр `HelloWorld` базового класса `Object`. Само описание экземпляра класса находится между первой открывающей фигурной скобкой и последней закрывающей.

В строке

```
public static void main (String args []) {
```

описан метод (функция) `main`. Интерпретатор Java начинает работу с вызова метода `main`, причем транслятор Java (`javac`) может оттранслировать объект класса, в котором метод `main` отсутствует, а вот интерпретатор запускать объекты без метода `main`

не сможет. Фактически метод `main` является (по аналогии с языком C++) точкой входа в программу.

Ключевое слово `public` указывает модификатор доступа, позволяющий управлять видимостью любого метода и любой переменной. В данном примере `public` означает, что метод `main` виден и доступен любому классу (существуют еще два модификатора доступа — `private` и `protected`, напоминающие по действию аналогичные им в языке C++).

Ключевое слово `static` объявляет методы и переменные класса, используемые для работы с классом в целом. Методы, объявленные с ключевым словом `static`, могут работать только с локальными и статическими переменными.

Ключевое слово `void` указывает (опять-таки, по аналогии с C++), что метод `main` не возвращает значение.

Выражение

```
String args[]
```

внутри круглых скобок объявляет параметр с именем `args`, представляющий собой массив объектов строкового типа, принадлежащих классу `String`. Квадратные скобки после `args` указывают на массив строк, а не на отдельный элемент. Вообще, все параметры, которые нужно передать методу, следует указывать внутри пары круглых скобок в виде списка элементов, разделенных символами `;` — при этом каждый элемент списка параметров указывается в форме

тип идентификатор

В нашем примере *тип* — это `String`, а *идентификатор* — это `args[]`.

Имя метода, не имеющего параметров, следует завершать парой круглых скобок.

Строка

```
System.out.prlntln("Hello, world!");
```

позволяет вывести на экран консоли строку "Hello, world", для чего выполняется метод `println` объекта `out`. Объект `out`, в свою очередь, инициализируется в классе `System`.

8.2.2. Синтаксис языка

Как и любой язык программирования, Java включает в себя множество элементов, в число которых входят ключевые слова, идентификаторы, константы, операторы и разделители.

В языке Java допускается произвольное форматирование текста программ, при этом единственным требованием является наличие между отдельными лексемами, по крайней мере, одного пробела (символа табуляции или символа перевода строки).

Комментарии в Java определяются так же, как и в C++:

- если комментарий расположен в одной строке, то он начинается с символов `//`;
- если комментарий расположен в нескольких строках, то его следует начать символами `/*` и закончить символами `*/`.

Зарезервированные или *ключевые слова* (keywords) предназначены для идентификации встроенных типов языка, модификаторов и средств управления ходом выполнения программы. В языке Java имеется несколько десятков зарезервированных слов, описание которых приводится в документации. Ключевые слова нельзя применять в качестве идентификаторов переменных, классов или методов.

С ключевыми словами мы сталкивались при анализе первого примера на языке Java (`public`, `static`, `class`, `void`). Многие ключевые слова позаимствованы из языка C++ и имеют тот же смысл.

Классы, методы и переменные в языке Java именуются с помощью *идентификаторов*, представляющих собой любую последовательность строчных и прописных букв, цифр, а также символов подчеркивания и доллара, при этом идентификатор не должен начинаться с цифры.

Регистр букв различается, поэтому, например, `Var1` и `var1` — это разные идентификаторы.

Константы в Java задаются их литеральным представлением, при этом целые числа, числа с плавающей точкой, логические значения, символы и строки можно располагать в любом месте исходного кода.

К литералам относятся:

- целочисленные литералы, представленные десятичными значениями (например, `2`, `-89`), шестнадцатеричными значениями (например, `0xfd`, `0XA9`) и восьмеричными значениями (`025`, `017`);
- литералы с плавающей точкой, представляющие собой десятичные значения с дробной частью в стандартном или экспоненциальном форматах. Например, `2.01`, `3.14159` и `.17` — это допустимые значения чисел с плавающей точкой, записанных в стандартном формате, а числа `7.92e23`, `314159E-05`, `3e+10` представляют собой числа в экспоненциальном формате;
- логические литералы — они принимают лишь два значения — `true` (истина) и `false` (ложь), при этом логические значения `true` и `false` не равны 1 и 0 соответственно. В Java эти значения могут присваиваться только переменным типа `boolean` либо использоваться в выражениях с логическими операторами;
- символьные литералы — представляют собой 16-битовые значения, которые можно преобразовать в целые числа и к которым можно применять операторы целочисленной арифметики, например, операторы сложения и вычитания. Символьные литералы выделяются парой апострофов `''`. Все видимые символы таблицы ASCII можно прямо вставлять внутрь пары апострофов, например, `'a'`, `'z'`, `'@'`. Для символов, которые невозможно ввести непосредственно, предусмотрено несколько управляющих последовательностей;

- строчные литералы в Java представляют собой произвольный текст, заключенный в пару двойных кавычек "", причем такие литералы должны начинаться и заканчиваться в одной и той же строке исходного кода.

К элементам языка Java относятся и *операторы*. Большинство операторов языка имеет тот же смысл, что и в C++, например, `=`, `>`, `<`, `!=` и т. д., хотя есть и операторы, включенные только в Java.

Язык Java позволяет хранить данные в переменных нескольких типов: `byte`, `short`, `int`, `long`, `char`, `float`, `double`, `boolean`, *`имя класса`*. Переменная характеризуется идентификатором, типом и областью действия и объявляется, как и в большинстве других языков, следующим образом:

```
тип идентификатор = значение;
```

Вот примеры объявления переменных:

```
int a = 1, b = 2, c, d = 5;
```

Здесь объявляются четыре переменных целого типа `a`, `b`, `c`, `d` с присвоением им начальных значений.

8.2.3. Введение в классы

В этом разделе мы вкратце ознакомимся с основами объектно-ориентированного программирования (сокращенно ООП), на котором базируется язык Java. Анализ теоретических основ объектно-ориентированного программирования потребовал бы отдельной книги, поэтому проанализируем лишь важнейшие практические аспекты использования ООП. Более подробно теоретические основы ООП описаны в документации по языку Java и в многочисленной литературе по другим языкам программирования, в которых используется объектно-ориентированный подход.

Базовым понятием объектно-ориентированного программирования является *класс*. Класс можно определить как шаблон для создания объекта — он задает как данные, так и код, который этими данными оперирует. Язык Java использует спецификацию

класса для создания объекта. Объекты — это экземпляры класса. Таким образом, класс можно рассматривать и как множество намерений (планов), определяющих, как должен быть построен объект.

Здесь очень важно понять то, что класс — это логическая абстракция. О ее реализации нет смысла говорить до тех пор, пока не создан объект или экземпляр класса, и в памяти не появилось его физическое представление.

Определяя класс, вы определяете данные, которые он содержит, и код, манипулирующий этими данными. Данные содержатся в переменных экземпляров, определяемых классом, а код — в методах. Как переменные экземпляров, так и методы называются *членами класса*. Класс создается с помощью ключевого слова `class`. Общая форма определения класса, который содержит только переменные экземпляров и методы, имеет вид:

```
доступ class имя_класса {
      // Объявление переменных экземпляров
  доступ тип переменная1;
  доступ тип переменная;2
      // . . .
  доступ тип переменнаяN;
      // Объявление методов
  доступ тип_возврата метод1(параметры) {
      // тело метода
  }
    доступ тип_возврата метод2(параметры) {
      // тело метода
  }
      // . . .
  доступ тип_возврата методN(параметры) {
      // тело метода
  }
}
```

Обратите внимание на то, что объявление каждой переменной и каждого метода предваряется элементом *доступ*. Элемент *доступ* определяет спецификацию доступа (например, `public`), который задает, как к этому члену класса можно получить доступ.

Спецификатор доступа необязателен, и, если он не указан, подразумевается, что член класса закрыт (`private`). Члены класса с закрытым доступом могут использоваться только другими членами своего класса. Члены класса, указанные со спецификатором `public`, могут использоваться остальными частями программного кода, даже теми, которые определены вне класса.

Рассмотрим пример объявления класса:

```
public class AddInts {
  public int i1;
  public int i2;
  public int add (int x1, int x2)
   {
     return (x1+x2);
   }
 }
```

Здесь мы объявили класс `AddInts` (он будет использоваться в последующих примерах), включающий две переменные `i1` и `i2` целого типа, доступные из других частей программного кода (спецификатор `public`), а также метод `add`, доступный всей программе. Метод принимает два целочисленных параметра `x1` и `x2`, а возвращает сумму этих параметров.

Напомню, что объявление `class` не создает реальных объектов, это лишь описание типа (в данном случае `AddInts`). Чтобы реально создать объект класса `AddInts`, можно использовать такую, например, конструкцию:

```
AddInts a1 = new AddInts();
```

Обратите внимание, что для создания объекта типа `AddInts` используется оператор `new`, который создает экземпляр указанного класса, возвращая при этом ссылку на объект. После создания

экземпляра класса можно использовать его переменные и методы в программе. Для доступа к переменным и методам объекта `a1` типа `AddInts` нужно использовать оператор "точка" (.).

Например, присвоить переменной `i1` значение 7 можно, выполнив оператор

```
a1.i1 = 7;
```

Вычислить сумму чисел 34 и −67 и сохранить ее в переменной `sum` можно, например, так:

```
int sum = a1.add(34, -67);
```

Экземпляры класса необязательно могут содержать метод `main`. Метод `main` указывает интерпретатору Java, откуда следует начинать выполнение программы.

Исключительно ценным свойством класса является то, что на его базе можно создать другие классы, наследующие его переменные и методы. Созданные классы могут включать другие переменные и методы, кроме тех, что уже существуют в базовом классе, а также переопределять методы базового класса. Мы не будем обсуждать подробно данную тему ввиду ограниченности объема книги, а рассмотрим практический пример создания класса `SubInts` на основе `AddInts`. Для создания нового класса используется ключевое слово `extends`:

```
public class SubInts extends AddInts {
  public int add (int x1, int x2)
   {
     return (x1-x2);
   }
  public int mul (int x1, int x2)
   {
     return x1*x2;
   }
 }
```

Здесь объявлен класс `SubInts`, который включает новый метод `mul` и, кроме того, переопределяет (overrides) метод `add` базового класса `AddInts`. Теперь при вызове этого метода из экземпляра класса `SubInts` вместо суммы чисел будет вычисляться их разность.

Посмотрим, как все это работает в реальной программе. Для этого выполним следующие шаги:

1. Сохраним исходный текст объявления классов `AddInts` и `SubInts` в файлах AddInts.java и SubInts.java (не следует забывать о регистре букв!).
2. Создадим файл Demo.java, в который включим следующий исходный текст:

```
import java.io.*;

public class Demo {
  public static void main(String args[]) {
    AddInts a1 = new AddInts();

    int res = a1.add(34, -67);
    System.out.println("Sum = " + res);

    SubInts s1 = new SubInts();
    int sub = s1.add(34, -67);
    System.out.println("Substraction = " + sub);

    int mul = s1.mul(34, -67);
    System.out.println("Multiplication = " + mul);
  }
}
```

3. Откомпилируем файл Demo.java (файлы AddInts.java, SubInts.java и Demo.java должны находиться в одном каталоге!):

```
# javac Demo.java
```

После запуска на выполнение файла Demo.class получим следующий результат:

```
# java Demo
Sum = -33
Substraction = 101
Multiplication = -2278
```

В ряде случаев требуется создать метод или определить переменную, которые можно было бы использовать вне контекста какого-либо объекта его класса. Для создания членов класса с такими свойствами следует указать для них модификатор типа `static`. Переменные типа `static` в какой-то степени подобны глобальным переменным, поскольку доступны из любого места кода.

Далее приведен исходный текст класса `DemoStatic`, в котором объявлена статическая переменная `i1` и статический метод `mul`:

```
public class DemoStatic
{
 static int i1;
 static int mul(int a1, int a2)
  {
   return a1 * a2;
  }
}
```

А вот как можно использовать статические переменные и методы в программе:

```
public class Program
{
  public static void main(String[] args)
   {
    int res = DemoStatic.mul(11, 96);
    DemoStatic.i1 = -771;
```

```
    res += DemoStatic.i1;
    System.out.println("res = " + res);
  }
}
```

Обратите внимание, что для вызова статических методов и переменных не требуется создавать экземпляр класса `DemoStatic` — обращение к ним осуществляется посредством имени класса.

Результатом работы такой программы будет вывод:

```
res = 285
```

На этом закончим краткое рассмотрение классов и обсудим некоторые практические аспекты разработки программ в Java.

8.2.4. Обработка ошибок

Язык Java является основой для разработки большинства приложений, работающих с сетевыми устройствами и переносными (мобильными) системами. При разработке таких приложений чрезвычайно важно обеспечить их надежное функционирование и эффективную обработку при возникновении ошибок или, по-другому, исключительных ситуаций (exceptions).

Java имеет очень мощный механизм обработки исключительных ситуаций, позволяющий корректно обрабатывать возникающие ошибки с минимальными потерями функциональности приложений.

Механизм исключений позволяет отделить код обработки ошибок (error-handling code) от кода программы, что делает программный код приложения более читабельным и понятным. При возникновении исключительной ситуации управление передается специальному блоку перехвата исключений (catcher block) программы, который содержит информацию о характере ошибки и способы выхода из такой ситуации. Компилятор Java требует, чтобы метод объекта либо сам возбуждал исключение, либо, в крайнем случае, это исключение обрабатывал.

Во многих случаях компилятор Java подсказывает, в каком месте программы может возникнуть исключение, и требует включения блока обработки исключения.

В практическом плане программа на Java должна содержать последовательность операторов `try...catch` примерно в такой форме:

```
try
   {
      обычный код программы
   }
catch
   {
      код обработки исключительной ситуации
   }
```

В блоке `try` заключен программный код выполняющейся программы, а блок `catch` должен содержать код обработки исключительной ситуации.

Исключения (exceptions) представляются объектами класса `java.lang.Exception` и его подклассами, при этом подклассы могут включать дополнительную информацию и, во многих случаях, алгоритм поведения при возникновении такой ситуации. Замечу, что обработка исключений — прерогатива программиста, и он сам несет ответственность за обработку ошибок. Тем не менее, многие компиляторы Java часто не выполняют компиляцию программного кода, требуя включения обработчика тех или иных возможных исключений.

Например, при работе с потоками ввода/вывода желательно включать в программу, как минимум, обработчик исключения `IOException`. Как правило, перед обработкой исключения желательно получить информацию о характере ошибки, что позволяет сделать метод `getMessage`. Вот пример программного кода с обработчиком исключения ввода/вывода:

```
try
   {
     код программы
```

```
   }
catch (IOException e)
   {
      System.out.println("IO Error: " + e.getMessage());

      <анализ ошибки и ее обработка>
   }
```

В программе могут присутствовать несколько блоков `catch`, которые обрабатываются последовательно:

```
try
    {
       . . .
     }
catch (. . .)
     {
       . . .
     }
catch (. . .)
     {
       . . .
     }
. . .
```

В последующих примерах программного кода вы увидите практическое использование блоков `try...catch`.

8.2.5. Работа со строками

Язык Java позволяет работать со строками символов, используя механизмы ООП. Для этого в пакете `java.lang` определен класс `String`, реализующий объектное представление символьного массива. В класс `String` включены методы, позволяющие выполнять различные манипуляции со строками: сравнение строк,

поиск символов и подстрок. Для модификаций строк удобно использовать класс `StringBuffer`, позволяющий объединять и форматировать строки, а также манипулировать отдельными символами строки.

Создать строку очень просто — для этого используется оператор `new`:

```
String s = new String();
```

В данном случае создается пустая строка `s`, не содержащая ни одного символа. Такой строке можно далее в программе присвоить какое-либо значение, например:

```
s = "Text";
```

Следующий оператор создает новую строку, присваивая ей значение строчного литерала:

```
String s1 = "Hello, world!";
```

Строки можно формировать из массивов символов, как в этом случае:

```
char[] bytes = { '1', 'q', 'w' };
String s2 = new String(bytes);
```

В данном случае будет сформирована строка `s2`, содержащая значение `"1qw"`.

Немного модифицированный фрагмент предыдущего примера помещает в строку `s3` значение `"qw"`:

```
char[] bytes = { '1', 'q', 'w' };
String s3 = new String(bytes, 1, 2);
```

Здесь при формировании строки второй параметр, равный 1, указывает индекс первого элемента массива, начиная с которого формируется строка, а третий параметр, равный 2, указывает количество элементов в строке. Поскольку индексация элементов массивов начинается с 0, то первым элементом строки `s3` будет символ `q`.

Обратная операция — преобразование строки в массив символов — обычно используется для манипуляций отдельными элементами,

а также добавления, вставки, перемещения и других операций с группами символов, что для массива выполнить намного проще.

Далее приводится исходный текст программного кода, выполняющего преобразование строки типа `String` в массив символов, после чего элементы массива выводятся на экран дисплея:

```
String s1 = "Test String ";
int len = s1.length();
char[] chars = new char[len];
s1.getChars(0, s1.length(), chars, 0);
System.out.println("Length of string = " + len);
for (int i1 = 0; i1 < s1.length(); i1++)
   System.out.print(chars[i1]);
```

Вначале определяется размер строки `s1` с помощью метода `length` и сохраняется в переменной `len`. Значение переменной `len` используется для инициализации массива символов `chars`, куда помещаются элементы.

Собственно преобразование выполняется посредством метода `getChars` класса `String`, после чего содержимое символьного массива выводится на экран. Метод `getChars` принимает четыре параметра:

- первый параметр — начальное смещение в строке (в данном случае принимаем его равным 0);
- второй параметр — конечное смещение (равно размеру строки);
- третий параметр — указатель на массив символов, куда будут записаны элементы строки;
- четвертый параметр — смещение в массиве (принимаем равным 0).

Рассмотрим еще несколько полезных методов класса `String`. Следующий фрагмент программного кода позволяет сохранить в переменной символьного типа элемент строки с индексом 5:

```
String s4 = "Test String 1";
char c1 = s4.charAt(5);
```

После выполнения этого кода переменная `c1` будет содержать литеру `S`.

Удалить пробелы в начале и в конце строки можно с помощью метода `trim` класса `String`. Если задана строка `s1`, то удаление пробелов осуществляется с помощью оператора `s1.trim()`.

Объединить две строки можно с помощью следующего фрагмента кода:

```
String s4 = "Test String 1";
String s5 = s4.concat (" + String 2");
```

После выполнения операции строка `s5` будет содержать значение

```
Test String 1 + String 2
```

Поскольку для строки `s4` был выделен фиксированный размер памяти, то ничего добавить в эту строку уже нельзя, поэтому и была сформирована строка `s5`.

Альтернативной записью для предыдущей операции объединения строк может быть такой программный код:

```
String s4 = "Test String 1";
String s5 = s4 + " + String 2";
```

Выделить подстроку из строки можно, например, так:

```
String s1 = "Test String 1 ";
String s2 = s1.substring(5, 12);
```

Здесь из строки `s1` выделяется подстрока `s2`, принимающая значение `"String 1"`.

Первым элементом подстроки `s2` является элемент строки `s1` с индексом 5 (`S`), а последним — элемент с индексом 12 (`1`).

Преобразовать все символы строки к строчным литерам можно следующим образом:

```
String s1 = "Test String 1 ";
String s2 = s1.ToLower();
```

В результате строка `s2` получит значение

```
test string 1
```

Обратную операцию — преобразование символов к верхнему регистру — можно выполнить с помощью метода `ToUpper` класса `String`.

Строку можно разделить на подстроки произвольным образом, задав в качестве разделителя один или несколько символов. Следующий фрагмент программного кода выделяет из строки `s1` подстроки, анализируя наличие одного из разделителей (символ пробела, символ табуляции или символ "плюс"):

```
String s1 = "Test+String    1+String 2 9";
char sep[] = { ' ', '', '+' };
String s2[] = s1.Split(sep);
int len = s2.length;
System.out.println("Number of substrings = " + len);
for (int i1 = 0; i1 < len; i1++)
  System.out.println("String " + i1 + ":  " + s2[i1]);
```

Здесь для разделения строки на подстроки используется метод `Split` класса `String`. В качестве параметра метод принимает указатель на массив символов `sep`, содержащий список разделителей. Выделенные подстроки сохраняются в массиве строк `s2`.

По завершению операции все строки массива `s2` выводятся на экран в цикле `for`. Переменная `len` содержит количество элементов в массиве `s2`.

В данном примере на экран будут выведены такие подстроки:

```
Number of substrings = 6
String 0:  Test
String 1:  String
String 2:  1
String 3:  String
String 4:  2
String 5:  9
```

Класс `String` позволяет сравнивать две строки, для чего используется метод `equals`. Следующий фрагмент программного кода сравнивает две строки, причем из одной из них удалены все пробелы:

```
String s1 = "  Test String   ";
if (s1.trim().equals("Test String"))
  System.out.println("Equals");
else
  System.out.println("Different");
```

В данном случае результат, выведенный на экран, будет `"Equals"`.

Класс `String` включает еще целый ряд методов для работы со строками, которые при желании читатели смогут без труда проанализировать самостоятельно.

Обработку строк можно выполнять и с помощью класса `StringBuffer`. В отличие от класса `String`, который позволяет манипулировать только строками фиксированной длины, объекты типа `StringBuffer` представляют собой последовательности символов, которые могут расширяться и модифицироваться.

Если создать объект `StringBuffer` без параметров, то для него выделяется область памяти для размещения 16 символов. Если передать конструктору строку, то она будет скопирована в объект, и, кроме этого, будет зарезервирована память еще для 16 символов.

Размер объекта `StringBuffer` можно определить с помощью метода `length`, а для определения объема памяти, зарезервированной для строки в объекте, следует воспользоваться методом `capacity`.

Далее приводится фрагмент программного кода, в котором выполняются манипуляции размером буфера:

```
StringBuffer sbuf = new StringBuffer("String 1");
System.out.println("sbuf length before appendage: " +
sbuf.length());
```

```
sbuf.append(" String 2");
System.out.println("sbuf length after appendage: " +
sbuf.length());


System.out.println("sbuf capacity before extending: " +
sbuf.capacity());
sbuf.setLength(64);


System.out.println("sbuf capacity after extending: " +
sbuf.capacity());
```

При указанных строках на экран будет выведен следующий результат:

```
sbuf length before appendage: 8
sbuf length after appendage: 17
sbuf capacity before extending: 24
sbuf capacity after extending: 64
```

Обратите внимание на то, как можно увеличить размер буфера строки — это делается с помощью метода `setLength` в операторе

```
sbuf.setLength(64);
```

В результате выполнения этого оператора размер памяти для буфера строки был увеличен до 64 байтов.

Добавить строку в объект `StringBuffer` можно при помощи метода `append`. В только что рассмотренном фрагменте программного кода в буфер была добавлена строка `"String 2"`.

Для извлечения одиночного символа из объекта типа `StringBuffer` служит метод `charAt`, что демонстрирует следующий пример:

```
StringBuffer sbuf = new StringBuffer("TEST String");
char c1 = sbuf.charAt(5);
```

После выполнения этих операторов в переменной `c1` будет находиться литера `S`.

Метод `setCharAt` позволяет записать в заданную позицию строки нужный символ. Например, записать на позицию 5 (6-й по порядку элемент) символ s можно с помощью оператора

```
sbuf.setCharAt(5, 's');
```

После выполнения этого оператора строка, представленная объектом `sbuf`, будет иметь вид `"TEST string"`.

Метод `insert` позволяет вставить символ или символы в определенное место в буфере, задаваемое первым его параметром, например:

```
sbuf.insert(5, "word ");
```

После выполнения этого оператора на 5-ю позицию в строке будет записана строка `word`, а размер строки увеличится на 1. Сама строка будет выглядеть как `"TEST word String"`.

8.2.6. Пакеты

В языке Java используется специальный механизм, который сообщает транслятору, в каком пакете должны определяться содержащиеся в данном файле классы. *Пакеты* задают набор раздельных пространств имен, в которых хранятся имена классов. Для этой цели предназначен оператор `package`.

Например, следующий оператор

```
package java.awt.image;
```

указывает транслятору на то, что исходный код этого класса должен храниться в каталоге java/awt/image.

В исходном тексте программы могут присутствовать операторы `import`, указывающие транслятору местонахождение пакетов. Они должны включаться в исходный текст до любого определения классов.

Оператор `import` можно представить в таком формате:

```
import пакет1[.пакет2].(имя_класса | *);
```

Параметры имеют следующий смысл:

- *пакет1* — имя пакета верхнего уровня;
- *пакет2* — необязательное имя вложенного пакета;
- *имя_класса* | * — имя класса или символ *, указывающий транслятору на то, что в процессе поиска класса следует просмотреть все содержимое пакета.

Вот примеры использования оператора `import`:

```
import java.security.Identity;
import java.io.*;
import java.net.*;
```

Все встроенные в Java стандартные классы находятся в пакете с именем `java`, а базовые функции языка определены в классах, находящихся во вложенном пакете `java.lang`, который автоматически импортируется во время трансляции программы. Эта особенность транслятора позволяет избегать применения оператора `import java.lang.*`.

8.2.7. Ввод/вывод в Java

Язык Java поддерживает целый ряд механизмов ввода/вывода данных, наиболее важным из которых является *механизм потоков*. Механизм потоков в Java реализован посредством нескольких базовых классов из пакета `java.io`, к которым относятся `InputStream`, `OutputStream`, `Reader` и `Writer`. В основе концепции потоков лежит следующее фундаментальное положение: независимо от физической природы передаваемых данных (терминальный ввод/вывод, коммуникационный канал, дисковый файл и т. д.), все они рассматриваются как непрерывные неупорядоченные потоки байтов, имеющие тот или иной тип.

В Java различаются потоки байтов и потоки символов, поэтому базовыми классами для обработки потоков байтов являются `InputStream` и `OutputStream`, в то время как базовыми классами для обработки неструктурированных потоков символов являются `Reader` и `Writer`. Все остальные классы ввода/вывода являются

подклассами этих 4-х основных классов. Вкратце рассмотрим множество подклассов потокового ввода/вывода:

- `InputStreamReader` и `OutputStreamWriter` — классы, предназначенные для взаимного преобразования байтов в символы и наоборот;
- `DataInputStream`, `DataOutputStream` — классы, выполняющие функции потоковых фильтров. Обеспечивают возможность передачи простых типов данных и объектов типа `String`;
- `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, `BufferedWriter` — обеспечивают дополнительную функциональность за счет буферизации ввода/вывода;
- `PrintWriter` — специализированный класс, предназначенный для вывода потока символов, например, текста;
- `PipedInputStream`, `PipedOutputStream`, `PipedReader`, `PipedWriter` — классы для обработки двунаправленных потоков данных, например, при обмене данными через программные каналы. Эти классы используются только в паре (чтение/запись). Например, данные, записанные в `PipedOutputStream` или `PipedWriter`, должны быть прочитаны из `PipedInputStream` или `PipedReader` соответственно;
- `FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter` — реализация классов `InputStream`, `OutputStream`, `Reader` и `Writer` для чтения/записи файлов в локальных файловых системах.

Все потоки данных в Java являются однонаправленными. Все потоковые классы, входящие в пакет `java.io`, работают либо с выходными, либо с входными потоками. Если программе необходимо осуществлять как ввод, так и вывод данных, то следует использовать соответствующие классы потокового ввода/вывода.

Далее мы рассмотрим некоторые практические примеры ввода/вывода данных в программах на языке Java, где будет показано использование потоковых классов.

В первом примере (листинг 8.1) демонстрируется ввод данных с консоли и вывод строки на экран дисплея.

Листинг 8.1. Ввод/вывод данных

```
import java.io.*;

public class readStdIn_2 {
  public static void main(String args[]) {
   try {
       int bytesReady = 0;
       do {
            bytesReady = System.in.available();
            if (bytesReady > 0) {
              byte [] bytes = new byte [bytesReady];
              System.in.read(bytes);
              String str = new String(bytes);
              System.out.println("You entered: " + str);
              System.out.println("Length: " + str.length());
             }
           } while (bytesReady == 0);
       }
   catch (IOException e)
       {
        System.out.println("Error: "+ e.getMessage());
       }
  }
}
```

В этом примере исполняемый код программы располагается в блоке `try`, а возможные ошибки обрабатываются в блоке `catch`. Рассмотрим ключевые моменты этой программы. Здесь класс

`java.lang.System`, представляющий системные ресурсы, ссылается на стандартный ввод посредством статической переменной `in`. Аналогично, ссылка на стандартный вывод осуществляется посредством статической переменной `out`.

Ввод данных с консоли осуществляется посредством метода `read` — при этом данные читаются в буфер `bytes`:

```
System.in.read(bytes);
```

Далее, данные в буфере `bytes` преобразуются к объекту типа `String`, после чего выполняется вывод строки на стандартный вывод:

```
System.out.println("You entered: " + str);
```

Следующий пример демонстрирует использование класса `DataInputStream` для ввода данных (листинг 8.2).

Листинг 8.2. Вывод данных с помощью класса `DataInputStream`

```
import java.io.*;


public class readStdIn {
  public static void main(String args[]) {
   try {
        DataInputStream dis = new DataInputStream(System.in);
        String str = dis.readLine();
        System.out.println("You entered: " + str);
       }
   catch (IOException e)
       {
        System.out.println("Error: "+ e.getMessage());
       }
  }
}
```

Приведу еще один пример, в котором показано, как записывать данные в файл (листинг 8.3).

Листинг 8.3. Запись данных в файл

```
import java.io.*;


public class Program
{
  public static void main(String[] args)
   {
    try
      {
        File myfile = new File("MyFile.txt");
        FileOutputStream fout = new FileOutputStream(myfile);
        String sf = "This string will be written to myfile";
        fout.write(sf.getBytes());
        fout.close();
       }
   catch (FileNotFoundException f)
       {
        System.out.println("File Error: " + f.get_Message());
       }
   catch (IOException e)
       {
        System.out.println("IO Error: " + e.get_Message());
       }
   }
}
```

Оператор

```
File myfile = new File("MyFile.txt");
```

создает новый объект `myfile` типа `File` с именем MyFile.txt. Далее создается экземпляр `fout` потока файлового вывода `FileOutputStream`, через который выполняется запись данных в файл.

Работа с сокетами

Одним из основных преимуществ языка Java по сравнению с другими является легкость создания сетевых приложений. Предпочтительным сетевым протоколом в Java является TCP, хотя можно работать и с UDP. Каналы потоковой передачи (TCP-соединения) наилучшим образом используются в Java, причем детали сетевого взаимодействия скрыты от программиста. Многие операции, связанные с работой сокетов, перенесены в классы ввода/вывода, что также облегчает программирование TCP. Для создания сетевых приложений в программе необходимо указывать пакет, где находятся соответствующие классы:

```
import java.net.*;
```

Рассмотрим простейшие примеры взаимодействия двух приложений по сети с использованием протокола TCP/IP, одно из которых является клиентом, а другое сервером. Начнем с клиента.

Клиентский сокет создается одной строкой, например:

```
Socket client = new Socket (hostname, port);
```

Здесь:

- *`hostname`* — имя хоста или его адрес (объект типа `String`);
- *`port`* — целочисленное значение из допустимого диапазона.

Вот два примера создания клиентских сокетов в Java:

```
Socket client = new Socket ("localhost", 7777);
Socket client = new Socket("127.0.0.1", 7777);
```

В первом случае первым параметром является имя хоста, а во втором — его IP-адрес в строковом представлении.

Для подключения к серверу собственно бóльшего и не требуется — все операции выполняются автоматически (сравните это с операциями создания сокета в C++ — кроме создания сокета, требуются инициализация структуры `sockaddr_in` и выполнение соединения явным образом при помощи функции `connect`).

После создания объекта класса `Socket` приложение не сможет принимать или передавать данные через него до тех пор, пока с сокетом не будут связаны входной (класс `InputStream`) и/или выходной (класс `OutputStream`) потоки. Следующие строки программного кода показывают, как выполнить такую привязку:

```
InputStream is = client.getInputStream();
OutputStream os = client.getOutputStream();
```

Предполагается, что объект `client` типа `Socket` был ранее создан одной из рассмотренных выше команд.

Далее можно осуществлять чтение/запись данных через сокет. Как выполняется это на практике, можно увидеть из примера, демонстрирующего передачу данных серверу TCP (листинг 8.4).

Листинг 8.4. Программа-клиент TCP

```
import java.io.*;
import java.net.*;

public class Client1
{
  public static void main(String[] args)
   {
     try
       {
        System.out.println("Enter string:");
        byte[] bytes = new byte[1024];
        Socket c1 = new Socket("127.0.0.1", 1777);
        OutputStream out = c1.getOutputStream();
```

```
        System.in.read(bytes);
        out.write(bytes);
        c1.close();
       }
   catch (SocketException s)
     {
      System.out.println("SOCKET error: " + s.getMessage());
     }
   catch (IOException e)
     {
      System.out.println("IO Error: " + e.getMessage());
     }
   }
}
```

Строки

```
Socket c1 = new Socket("127.0.0.1", 1777);
OutputStream out = c1.getOutputStream();
```

нам уже знакомы — так создается клиентский сокет `c1` на порту 1777, и выполняется его привязка к потоку вывода `out`.

Данные, которые должны передаваться по сети, читаются с консоли в массив байтов `bytes` с помощью оператора

```
System.in.read(bytes);
```

Далее, используя метод `write` потока `out`, данные передаются указанному адресату (в данном случае, как клиент, так и сервер работают на одной и той же машине, но это не меняет сути происходящего):

```
out.write(bytes);
```

По окончанию обмена данными сокет должен быть закрыт:

```
c1.close();
```

Вот, собственно, и все о клиенте. Как обычно, программа должна обрабатывать исключительные ситуации, если они возникают, для чего используются блоки `try...catch`. Обратите внимание на то, что можно обрабатывать сразу несколько исключений, разместив их обработчики последовательно. В данном случае обработка сводится к выводу сообщений о характере ошибки.

Разработка программы-сервера, принимающей данные от клиента, не намного сложнее. Для создания сокета в сервере TCP предусмотрен специальный класс `ServerSocket`:

```
ServerSocket srv = new ServerSocket (port);
```

Здесь создается объект `srv` класса `ServerSocket`, причем достаточно указать единственный параметр — номер порта.

Параметр порт, как и в клиентском сокете, является целочисленным значением из допустимого диапазона и должен совпадать с номером порта, указанным в клиентской программе.

Еще одной особенностью TCP-сервера является и то, что созданный серверный сокет только прослушивает соединение, но не может принимать или передавать данные через него. При поступлении от клиента запроса на соединение должен создаваться сокет для обмена данными именно по конкретному соединению, что можно сделать посредством метода `accept`. Сама программа-сервер должна работать в бесконечном цикле прослушивания запросов на соединение, обрабатывая поступившие запросы.

В листинге 8.5 представлен исходный текст программы-сервера.

Листинг 8.5. Программа-сервер TCP

```
import java.io.*;
import java.net.*;

public class Server1
{
```

```
 public static void main(String[] args)
  {
   try {
        ServerSocket s1 = new ServerSocket (1777);
        System.out.println("Waiting on port 1777...");
        while (true )
        {
         Socket new_con = s1.accept ();
         InputStream in = new_con.getInputStream();
         DataInputStream din = new DataInputStream (in);
         String st = din.readLine();

         System.out.println("The String Read: " + st);
         if (st.trim().equals("bye"))
         {
          new_con.close();
          break;
         }
         new_con.close();
       }
      s1.close();
    }
  catch (SocketException s)
  {
   System.out.println("Socket error: " + s.getMessage());
  }
  catch (IOException e)
  {
   System.out.println("IO error: " + e.getMessage());
  }
 }
}
```

Вначале программа создает прослушивающий сокет на порту 1777:

```
ServerSocket s1 = new ServerSocket (1777);
```

Далее в цикле `while` сервер ожидает подключений и, как только клиент устанавливает соединение, создает для него отдельный канал обмена данными:

```
Socket new_con = s1.accept ();
```

Обратите внимание на то, что созданный объект `new_con` является обычным (не серверным!) сокетом.

Далее для получения данных от клиента создается новый входной поток `in`, связанный с сокетом `new_con`:

```
InputStream in = new_con.getInputStream();
```

Для удобства обработки и вывода данных на консоль создается поток данных `din` типа `DataInputStream`:

```
DataInputStream din = new DataInputStream (in);
```

Наконец, принятые данные запоминаются в строке `st` типа `String`:

```
String st = din.readLine();
```

и отображаются на экране консоли:

```
System.out.println("The String Read: " + st);
```

Программа-сервер при приеме строки данных каждый раз сравнивает ее со значением `"bye"` (оператор `if`) и при совпадении закрывает соединение после выхода из цикла `while`.

Еще один момент: после обработки данных принимающий сокет следует закрыть:

```
new_con.close();
```

Рассмотренный нами сервер TCP работает в блокирующем режиме, что для многих приложений может оказаться неэффективным. Для работы в неблокирующем режиме можно модифицировать приложение, поместив обработчики принимаемых данных в отдельные потоки, что позволит принимать параллельно данные из нескольких источников.

8.2.8. Апплеты

С помощью языка Java, кроме обычных приложений, можно создавать и так называемые *апплеты*. Апплеты создаются на базе специального класса Java, используемого Web-браузерами, при этом и сами апплеты запускаются из браузера. Более того, апплеты являются программами с графическим интерфейсом.

Апплет запускается при обращении Web-браузера к HTML-странице, на которой он находится. Байт-код апплета передается браузеру, который далее интерпретирует его соответствующим образом. Для создания апплета метод `main` следует заменить методом `paint`, указав при этом, что программа является апплетом.

Кроме того, вместо консольного потока `System.out` необходимо использовать средства графического интерфейса.

В листинге 8.6 приводится пример программы HelloWorld.java, преобразованной в апплет.

Листинг 8.6. Апплет, отображающий "Hello, world!" в окне браузера

```
import java.applet.Applet;
import java.awt.Graphics;

public class Program extends Applet
{
  public void paint (Graphics g)
   {
     g.drawString("Hello, world!", 25, 25);
   }
}
```

Для работы апплета необходимо импортировать классы `Applet` и `Graphics`.

Запустить апплет можно из HTML-страницы, включив в нее, например, следующий текст:

```
<html>
  <applet code="HelloWorld.class" width=300 height=50>
  </applet>
</html>
```

Сохраним страницу под именем hello.html. Теперь при открытии страницы из браузера будет запускаться и апплет HelloWorld.class.

8.3. Perl

Прежде чем приступить к изучению языка Perl, хочу продемонстрировать его возможности. Многие программы, написанные на этом языке, занимают всего несколько строк, выполняя те же функции, что и сложные командные файлы, в которых задействованы команды `find`, `awk`, `sed` и т. д. Вот простой пример.

Предположим, имеются три файла — 1.tst, 2.tst и 3.tst со следующим содержимым:

```
$ cat 1.tst
1234567890
$ cat 2.tst
0123456789
$ cat 3.tst
0123006789
```

Пусть требуется заменить цифры `0` в этих файлах на символы `N`. Это легко выполняется при помощи команды

```
$ perl -e 's/0/N/gi' -p -i *.tst
```

После выполнения операции содержимое данных трех файлов будет следующим:

```
$ cat 1.tst
123456789N
```

```
$ cat 2.tst
N123456789
$ cat 3.tst
N123NN6789
```

Рассмотрим более сложный пример. Пусть файл 1.tst содержит следующие строки:

```
# cat 1.tst
111 2222 7777
VR 33LG 9037
9012 FT 34
WE JK355 133
```

Требуется заменить первые два элемента в строках, начинающихся двумя цифрами, на строку `NN`. Вот как можно это реализовать при помощи Perl:

```
# perl -e 's/^\d{2}/NN/gi' -p -i 1.tst
```

Результат преобразования содержимого файла выглядит так:

```
# cat 1.tst
NN1 2222 7777
VR 33LG 9037
NN12 FT 34
WE JK355 133
```

Как видно даже из этих примеров, далеко не самых сложных, язык Perl обладает весьма широкими возможностями по обработке данных.

Еще большего можно добиться, разрабатывая полноценные приложения в классическом стиле с использованием всех функций этого языка. Вот как можно модифицировать предыдущий пример, разработав приложение на языке Perl (исполняемый файл называется replace_demo.pl) — листинг 8.7.

Листинг 8.7. Замена двух лидирующих цифр в строках файла на NN

```
#!/usr/bin/perl -w

$orig='^\d{2}';
$replacement='NN';
$newfile = "new";
foreach $file (@ARGV)
  {
   if (! open(INPUT,"<$file") )
     {
       print "Can't open input file $file\n";
       next;
     }

open(OUTPUT, "+>>$newfile");
select(OUTPUT);
while (<INPUT>)
 {
   $data = $_;
   $data =~ s/$orig/$replacement/gi;
   print  OUTPUT $data;
 }
close (INPUT);
close(OUTPUT);

unlink("$file");
rename("new", $file) or die "Can't rename new to $file: $!";
}
exit(0);
```

Здесь я не буду подробно объяснять смысл операторов Perl, далее мы более подробно остановимся на этом. Замечу, что данная программа может принимать несколько аргументов командной строки (`$ARGV[0]`, `$ARGV[1]`, `$ARGV[2]`), в качестве которых должны выступать имена файлов.

Следует сказать, что программы на языке Perl легко пишутся, но не всегда легко читаются, учитывая, в особенности, то, что выполнить одну и ту же операцию можно несколькими способами.

Например, операторы, представленные далее, являются эквивалентными:

```
if ($x == 0) {$y = 10;}  else {$y = 20;}
$y = $x==0 ? 10 : 20;
$y = 20;  $y = 10 if $x==0;
```

Далее мы рассмотрим язык Perl более подробно и начнем с запуска программы.

8.3.1. Запуск программ

Запустить программу, написанную на Perl, не очень сложно. Рассмотрим простой пример запуска программы (назовем ее hello.pl), исходный текст которой представлен в листинге 8.8.

Листинг 8.8. Программа hello.pl

```
#!/usr/bin/perl -w
if ($#ARGV >= 0)
{
  $who = join(' ', @ARGV);
}
else
{
 $who = 'world';
}
print "Hello, $who!\n";
```

Здесь первая строка программы, которая начинается с символов `#!`, определяет расположение программы perl (обычно это каталог /usr/bin, хотя может быть и другой). В этой же строке задан параметр `-w`, указывающий на необходимость вывода предупреждающих сообщений.

Сам текст программы несложен. Если задан параметр командной строки, например, `root`, то программа выводит сообщение

```
Hello, root!
```

Если параметр не указан, то будет выведено сообщение

```
Hello, world!
```

После создания файла hello.pl нужно проверить атрибуты доступа — файл должен иметь доступ по чтению/записи и выполнению. Если какие-либо атрибуты отсутствуют, то, выполнив команду `chmod`, можно их установить. Например, чтобы владелец файла мог его прочитать, изменить и выполнить, нужно набрать командную строку:

```
# chmod 700 hello.pl
```

Для того чтобы файл можно было прочитать и выполнить всем остальным пользователям, наберите команду:

```
# chmod a+rx hello
```

Сама программа после этого запускается, как обычно, например:

```
# ./hello root
```

8.3.2. Скалярные переменные и массивы

Приступая к знакомству с любым языком программирования, следует, прежде всего, выяснить, а какие типы данных он позволяет обрабатывать. Данные определяются своим типом — множеством значений и набором допустимых операций. В языке Perl можно использовать все буквы латинского алфавита (прописные и строчные), арабские цифры и символ подчеркивания _. Perl относится к языкам, *чувствительным к регистру*. Это означает, что символы прописной и строчной буквы считаются

различными. Таким образом, два идентификатора `Str` и `str` задают две разные переменные.

Базовой единицей для работы с данными в Perl является скаляр (scalar) — отдельное значение, хранящееся в отдельной (скалярной) переменной. В скалярных переменных можно хранить строки, числа и ссылки. При этом строки могут иметь произвольную длину и содержать произвольные данные, включая двоичные последовательности с завершающим нулем. Числовые значения обычно хранятся в формате вещественных чисел с двойной точностью. Что же касается ссылок, то в них хранятся адреса переменных.

Скалярная величина может быть определенной или неопределенной. Определенная величина может содержать, как уже было сказано, число, строку или ссылку. При этом определенными величинами являются, например, 0 и пустая строка. Неопределенная величина может принимать единственное значение `undef`, для проверки которого можно воспользоваться функцией `defined`.

Все переменные должны начинаться с символа `$`, например:

```
$a=1;
$b="string";
```

Здесь переменной `$a` присваивается числовое значение, равное 1, а переменной `$b` — символьная строка `"string"`. Хочу отметить важный момент: интерпретатор Perl автоматически определяет тип переменной в момент присваивания ей значения, поэтому переменная `$a` в нашем примере будет считаться целочисленной, а переменная `$b` — строковой. Если требуется преобразовать строковое значение в число, то можно использовать функцию POSIX `strtod`.

В языке Perl существует несколько переменных, имеющих специальные назначения. К ним относится особый тип переменной, который обозначается, как `$_` и называется *переменной по умолчанию*. Эта переменная может принимать, например, строку со стандартного ввода или из файла, если явным образом не указана переменная-приемник.

Подобно другим языкам программирования, Perl позволяет выполнять над переменными различные арифметические и логические операции, что продемонстрировано в следующих двух примерах.

Программа из листинга 8.9 иллюстрирует принципы выполнения арифметических операций.

Листинг 8.9. Демонстрация арифметических операций

```
#!/usr/bin/perl -w
$a = 39;
$b = -99;
$sum = $a+$b;
$sub = $a-$b;
$mul = $a*$b;
$div = $a/$b;
$remainder = $b % $a;
#
print "Sum = $sum\n";
print "Sub = $sub\n";
print "Mul = $mul\n";
print "Div = $div\n";
print "Remainder = $remainder\n";
```

Результат работы программы (она называется arithmetic.pl) будет выведен на экран консоли:

```
# ./arithmetic.pl
Sum = -60
Sub = 138
Mul = -3861
Div = -0.393
Remainder = 18
```

А в листинге 8.10 показан пример выполнения операций сравнения.

Листинг 8.10. Демонстрация операций сравнения

```
#!/usr/bin/perl -w

print "Enter first number:";
$a = <STDIN>;
print "Enter second number:";
$b = <STDIN>;
if ($a > $b)
{
 print ("Max = $a\n");
}
if ($a < $b)
{
 print ("Max = $b\n");
}
 if ($a == $b)
{
 printf("Numbers are equal.\n");
}
```

Здесь показано выполнение операций сравнения чисел на "больше-меньше" или "равно", при этом на экране дисплея отображается соответствующий результат:

```
# ./logical.pl
Enter first number:-76
Enter second number:-762
Max = -76
```

Переменные `$a` и `$b` получают значения из устройства стандартного ввода, которое в языке Perl определено как `STDIN`:

```
$a = <STDIN>;
$b = <STDIN>;
```

Кроме того, в этом примере показано применение оператора условия `if`. Собственно, синтаксис условного оператора `if` языка Perl практически ничем не отличается от аналогичного оператора в C или других языках высокого уровня, поэтому останавливаться на нем подробно я не буду. То же самое касается и оператора цикла `while` и других логических конструкций — более подробно они будут рассмотрены далее.

Программе может понадобиться вычислять не только простейшие соотношения "равно — не равно" или "больше-меньше", но и более сложные выражения. В листинге 8.11 приводится пример программы, выводящей на экран дисплея число, введенное с консоли, только в том случае, если оно находится в диапазоне [−100; +100].

Листинг 8.11. Вывод числа из диапазона [–100; +100] на экран дисплея

```
#!/usr/bin/perl -w
print "Enter number between -100 and 100:";
$a = <STDIN>;
if (($a > -100) && ($a < 100))
{
 print ("Number = $a\n");
}
if (($a <= -100) || ($a >= 100))
{
 print "Number is out of range";
}
```

В этом примере в операторах `if` указано два условия:

```
($a > -100) && ($a < 100)
($a <= -100) || ($a >= 100)
```

Первое условие будет истинно только в том случае, если переменная `$a` будет больше −100 и меньше 100. Для истинности второго условия достаточно выполнения одного из условий: либо переменная `$a` меньше −100, либо больше 100.

Кроме простых скалярных переменных, Perl позволяет оперировать и группами элементов, представленными в виде массивов и хэшей. Здесь следует отметить, что строки не считаются массивом байтов, поэтому к отдельному символу нельзя обратиться по индексу, как к элементу массива — для этого следует использовать функцию `substr`.

Для обозначения массива используется символ амперсанда `@` перед именем массива, например,

```
@num_array= (0, 2, 4, 6)
```

Это выражение определяет массив `num_array` из четырех четных чисел. В следующем примере из листинга 8.12 (исполняемый файл 1.pl) на экране дисплея отображается массив из четырех вещественных чисел, разделенных пробелами.

Листинг 8.12. Вывод на дисплей элементов массива вещественных чисел

```
#!/usr/bin/perl -w
@array = (12.98, -56.02, 76.122, -4.031);
print "@array ";
```

Результат работы программы:

```
# ./1.pl
12.98 -56.02 76.122 -4.031
```

Тот же самый результат можно получить, используя программу с оператором `foreach` (листинг 8.13).

Листинг 8.13. Вывод на дисплей элементов массива из четырех вещественных чисел с помощью оператора `foreach`

```
#!/usr/bin/perl -w
@array = (12.98, -56.02, 76.122, -4.031);
print "Array: ";
print "$_ " foreach (@array);
```

Для обращения к отдельному элементу массива следует использовать индекс, а само обращение должно записываться в форме *$имя_массива[индекс]*. Например, для вывода на консоль 3-го элемента массива можно воспользоваться программным кодом, представленным в листинге 8.14.

Листинг 8.14. Вывод на консоль 3-го элемента массива вещественных чисел

```
#!/usr/bin/perl -w
@array = (12.98, -56.02, 76.122, -4.031);
print "$array[2]\n";
```

Вот как выглядит результат работы этой программы (она называется 2.pl):

```
# ./2.pl
76.122
```

Если необходимо вывести на экран дисплея несколько элементов массива, не обязательно идущих подряд, можно воспользоваться программным кодом из листинга 8.15.

Листинг 8.15. Вывод на консоль отдельных элементов массива

```
#!/usr/bin/perl -w
@array = (12.98, -56.02, 76.122, -4.031);
print "Array: ";
print "@array[1,3] ";
```

Здесь на экран выводятся 2-й и 4-й элементы массива `@array`.

С элементами массивов можно выполнять целый ряд операций. Например, для присвоения отдельным элементам массива определенных значений можно воспользоваться специальной формой оператора присваивания, как в примере из листинга 8.16.

Листинг 8.16. Инициализация отдельных элементов массива

```
#!/usr/bin/perl -w
@array = (0, 0, 0, 0, 0, 0, 0);
@array[1, 3, 5] = (-11, -19, -39);
print "@array ";
```

Результат работы программы будет таким:

```
0 -11 0 -19 0 -39 0
```

Если необходимо добавить элементы в конец массива, то можно сделать так, как показано в примере из листинга 8.17.

Листинг 8.17. Добавление элементов в конец массива

```
#!/usr/bin/perl -w
@array = (0, 0, 0, 0, 0, 0, 0);
@array[1, 3, 5] = (-11, -19, -39);
@array = (@array, 99, 77);
print "@array ";
```

Вывод программы показан далее:

```
0 -11 0 -19 0 -39 0 99 77
```

Для добавления элементов в конец массива часто бывает удобно воспользоваться функцией `push()`. Предыдущий пример, переписанный с использованием `push()`, будет выглядеть так, как показано в листинге 8.18.

Листинг 8.18. Добавление элементов в конец массива с помощью функции `push()`

```
#!/usr/bin/perl -w
@array = (0, 0, 0, 0, 0, 0, 0);
@array[1, 3, 5] = (-11, -19, -39);
push(@array, 99, 77);
print "@array ";
```

Чтобы удалить последний элемент массива, следует применить команду `pop()`. Следующий пример (он называется push_pop_demo.pl) демонстрирует это (листинг 8.19).

Листинг 8.19. Удаление последнего элемента массива

```
#!/usr/bin/perl -w
@array = (0, 0, 0, 0, 0, 0, 0);
@array[1, 3, 5] = (-11, -19, -39);
push(@array, 99, 77);
print "After push: @array \n";
$a1 = pop(@array);
$a2 = pop(@array);
print "After pop: @array \n";
print "\$a1 = $a1, \$a2 = $a2";
```

Обратите внимание на последнюю строку программы — здесь для вывода на экран специального символа `$` необходимо ликвидировать его специальное значение при помощи символа обратного слэша `\`.

Программа выводит на экран такой результат:

```
$ ./push_pop_demo.pl
After push: 0 -11 0 -19 0 -39 0 99 77
After pop: 0 -11 0 -19 0 -39 0
$a1 = 77, $a2 = 99
```

Весьма полезными являются и функции `shift()` и `unshift()`. Функция `shift()` удаляет первый элемент массива (с индексом 0), а `unshift()` — добавляет первый элемент в массив. Следующий пример (он называется shift_unshift_demo.pl) иллюстрирует это (листинг 8.20).

Листинг 8.20. Пример использования функций `shift()` и `unshift()`

```
#!/usr/bin/perl -w
@array = (0, 0, 0, 0, 0, 0, 0);
@array[1, 3, 5] = (-11, -19, -39);
print "@array \n";
unshift(@array, 999);
print "After unshift: @array \n";
shift(@array);
shift(@array);
print "After shift: @array \n";
```

Вот результат выполнения программы:

```
$ ./shift_unshift_demo.pl
0 -11 0 -19 0 -39 0
After unshift: 999 0 -11 0 -19 0 -39 0
After shift: -11 0 -19 0 -39 0
```

Количество элементов в массиве можно определить, используя выражение `$#`*`имя_массива`*`+1`. Если массив не определен, то вычисленное значение будет равно 0. В листинге 8.21 представлен простой пример (он называется sarray.pl).

Листинг 8.21. Определение количества элементов в массиве

```
#!/usr/bin/perl -w
@array = (5, -7, 25, 1, -9, 7, -3);
```

```
$sizeof_array = $#array+1;
print "Size of array = $sizeof_array\n";
```

Программа выводит на консоль такой результат:

```
$ ./sarray.pl
Size of array = 7
```

8.3.3. Хэши

Хэш представляет собой массив, состоящих из пар "ключ-значение", при этом доступ к каждому значению записи осуществляется по ассоциированному с ним ключу. Хэш можно представить как `%имя_хэша`, а доступ к его отдельным элементам выполняется в форме `$имя_хэша{выражение}`.

Вот примеры хэшей:

```
$hash{1}="key1";
$hash{'myset'}="file.txt";
```

Хэш должен содержать четное количество элементов.

Для удаления элементов из хэша нужно воспользоваться функцией `delete`:

```
delete($hash{1});
```

Чтобы выделить отдельные ключи и значения хэша, нужно воспользоваться функциями `keys()` и `values()` соответственно. Например, для хэша `%hash(1,5,2,6,3,7)` выполнение этих функций дает следующий результат:

```
@key = keys(%hash);
```

После выполнения функции `keys()` переменная `@key` будет содержать значения (1, 2, 3).

```
@value = values(%hash);
```

После выполнения функции `values()` переменная `@value` будет содержать значения (5, 6, 7).

8.3.4. Операции и выражения

Язык Perl обеспечивает обработку данных с помощью определенного набора *операций* — конкретных действий над операндами, результатом которых является новое значение. Все операции можно разделить на несколько основных типов, часть из которых перечислена далее.

Бинарные арифметические операции включают четыре арифметических действия: сложение (+), вычитание (-), умножение (*) и деление (/), а также остаток от деления двух целых чисел (%) и возведение в степень (**). Примененные к числовым данным или строковым, которые содержат правильные литералы десятичных чисел, они выполняют соответствующие арифметические действия.

Логические операции включают бинарные операции логического сложения ИЛИ (||), логического умножения И (&&) и унарную операцию логического отрицания !. Результат операции || (логическое ИЛИ) является истинным, если истинен хотя бы один из операндов, в остальных случаях он является ложным. Операция логического И (&&) возвращает в качестве результата истину, только если оба операнда истинны, в противном случае ее результат — ложь. Операция логического отрицания ! работает как переключатель: если ее операнд истинен, то она возвращает ложь, если операнд имеет значение ложь, то ее результатом будет истина.

Операции отношения используются для сравнения скалярных данных или значений скалярных переменных. Язык Perl предлагает набор бинарных операций, вычисляющих отношения равенства, больше, больше или равно и т. д. между операндами. Следует отметить, что для сравнения числовых данных и строковых данных Perl использует разные операции. Все они представлены в табл. 8.1.

Таблица 8.1. *Операции отношения*

Операция	Числовая	Строковая	Значение
Равенство	==	eq	Истина, если операнды равны, иначе ложь
Неравенство	!=	ne	Истина, если операнды не равны, иначе ложь
Меньше	<	lt	Истина, если левый операнд меньше правого, иначе ложь
Больше	>	gt	Истина, если левый операнд больше правого, иначе ложь
Меньше или равно	<=	le	Истина, если левый операнд больше правого или равен ему, иначе ложь
Больше или равно	>=	ge	Истина, если правый операнд больше левого или равен ему, иначе ложь
Сравнение	<=>	cmp	0, если операнды равны; 1, если левый операнд больше правого; –1, если правый операнд больше левого

Операция присваивания обозначается как = и является бинарной операцией, правый операнд которой может быть любым правильным выражением, тогда как левый операнд должен определять область памяти, куда помещается вычисленное значение правого операнда. В качестве левого операнда операции присваивания можно использовать переменную любого типа или элемент массива. В предыдущих примерах мы уже сталкивались с операцией присваивания.

В следующем примере

```
$a = $b + 10;
```

вычисляется значение правого операнда и присваивается переменной `$a`.

В языке Perl допускается осуществлять присваивание одного и того же значения нескольким переменным в одном выражении, например:

```
$varl = $var2 = $varl[0] = 1;
```

При реализации вычислительных алгоритмов приходится вычислять значение некоторой переменной, после чего результат присваивать этой же переменной. Например, если увеличить на 5 значение переменной `$a` и результат присвоить этой же переменной, то эту операцию можно реализовать так:

```
$a = $a+5;
```

Для таких и других подобных случаев в языке Perl предусмотрен более эффективный способ решения при помощи *оператора составного присваивания* `+=`, который прибавляет к значению левого операнда значение правого операнда и результат присваивает переменной, представленной левым операндом, например:

```
$a += 7;
```

Этот оператор эквивалентен оператору

```
$a = $a+7;
```

Операция составного присваивания более эффективна, чем обычное присваивание, поскольку в составном операторе присваивания переменная `$a` вычисляется один раз, тогда как в простом ее приходится вычислять дважды.

Операции над строками включают операции *конкатенации* (соединения) двух строковых операндов и *повторения*, которые объединяют два строковых операнда в одну строку.

Операция конкатенации обозначается символом "точка" (.). Пример конкатенации строк:

```
"first_string"."second_string";
```

В результате получится "`first_stringsecond_string`".

Результирующая строка содержит оба операнда, причем пробел между первым и вторым операндами отсутствует. Эта операция используется для присваивания переменной некоторого нового значения.

Для конкатенации двух и более строк через пробелы следует воспользоваться функцией `join`.

Вот еще один пример соединения строк:

```
$s1 = "First";
$s2 = "Second";
$s3 = "Third";
$s = $s1.$s2.$s3;
```

В результате строка `$s` будет содержать значение `"FirstSecondThird"`.

Операция повторения строки обозначается символом `x`. С помощью этой операции создается новая строка, повторяющая строку, заданную левым операндом, указанное число раз. Вот пример операции повторения:

```
"AB" x 2;
```

Результатом является строка `"ABAB"`.

Операцию повторения удобно использовать при повторении массивов для заполнения их одинаковыми значениями, например:

```
@array = ("Y") x 3;
```

В результате массив `@array` будет содержать элементы (`"Y"`, `"Y"`, `"Y"`).

Символ операции повторения `x` следует отделять пробелами от операндов, т. к. иначе он может быть воспринят интерпретатором, как относящийся к лексеме, а не представляющий операцию повторения. Например, интерпретатор Perl воспринимает строку `$xx$y` как состоящую из двух переменных `$xx` и `$y`, поэтому правильной записью будет `$x x $y`.

8.3.5. Логические структуры языка

Perl, как и другие языки высокого уровня, включает в себя много логических структур, позволяющих изменить линейное (последовательное) выполнение операторов программы и организовать вычислительные алгоритмы, зависящие от результатов выполнения операций. Рассмотрим наиболее важные из этих структур и начнем с оператора `if`. С ним мы уже сталкивались ранее, теперь рассмотрим его более подробно.

Оператор `if` можно представить в одной из двух форм:

```
if (условие) {
 операторы;
}
```

или

```
оператор if условие;
```

В листинге 8.22 приведен пример короткой программы с оператором `if`.

Листинг 8.22. Пример использования оператора `if`

```
#!/usr/bin/perl -w
$data = <STDIN>;
chomp($data);
print "You entered: $data" if ($data eq "OK");
```

Здесь переменной `$data` присваивается строка, полученная со стандартного ввода `STDIN`, затем функция `chomp()` убирает символ возврата каретки, после чего значение переменной `$data` выводится на экран, если оно равно `OK`.

Альтернативным к оператору `if` является `unless` (если не). Вот как будет выглядеть модифицированный вариант предыдущего примера, если вместо `if` используется `unless` (листинг 8.23).

Листинг 8.23. Пример использования оператора `unless`

```
#!/usr/bin/perl -w
$data = <STDIN>;
chomp($data);
print "You entered: $data" unless ($data ne "OK");
```

Следующая логическая структура, которую мы рассмотрим, является оператором цикла `while`. В общем виде синтаксис оператора можно представить так:

```
while (условие) {
  операторы;
}
```

Здесь условие представляет собой выражение, значение которого проверяется вначале каждой итерации. Операторы цикла выполняются до тех пор, пока условие является истинным. Если выражение становится ложным, происходит выход из цикла.

Рассмотрим следующий пример (листинг 8.24).

Листинг 8.24. Пример использования оператора `while`

```
#!/usr/bin/perl -w
@array = ("one", "two", "three", "four", "five", "six");
$index = 0;
while ($array[$index] ne "five")
 {
  print "$array[$index]\n";
  $index += 1;
 }
```

Здесь на экран дисплея выводятся все элементы массива `@array` вплоть до строки `"five"`. Условием выполнения цикла является неравенство текущего элемента значению `"five"`.

Вместо `while` можно использовать оператор `until`. В этом случае предыдущий пример можно модифицировать следующим образом (листинг 8.25).

Листинг 8.25. Пример использования оператора `until`

```
#!/usr/bin/perl -w
@array = ("one", "two", "three", "four", "five", "six");
$index = 0;
until ($array[$index] eq "five")
 {
  print "$array[$index]\n";
  $index += 1;
 }
```

Во многих случаях требуется выполнить определенное число повторяющихся операций. В языке Perl предусмотрена такая возможность, и реализована она посредством оператора `for`. Функционирование этого оператора лучше всего продемонстрировать на примере (листинг 8.26).

Листинг 8.26. Пример использования оператора `for`

```
#!/usr/bin/perl -w
@array = (5, -7, 25, 1, -9, 7, -3);
$sizeof_array = $#array+1;
print "Size of array = $sizeof_array\n";
$sum = 0;
for ($index = 0; $index < $sizeof_array; $index++)
 {
  $sum += $array[$index];
 }
print "Sum of elements = $sum\n";
```

Результат выполнения программы:

```
$ ./for_demo1.pl
Size of array = 7
Sum of elements = 19
```

Последняя логическая структура, которая будет рассмотрена, реализована в виде оператора `foreach`, специально предназначенного для обработки массивов. Оператор `foreach` можно представить следующим образом:

```
foreach $переменная (@массив) {
 операторы
}
```

Вот простейший пример применения оператора `foreach` (листинг 8.27).

Листинг 8.27. Пример использования оператора `foreach`

```
#!/usr/bin/perl -w
@array = (5, -7, 25, 1, -9, 7, -3);
foreach $entity (@array)
 {
  print "$entity ";
 }
```

Здесь на экран консоли выводится список элементов массива `@array`. Можно упростить исходный текст предыдущего примера, если воспользоваться переменной по умолчанию `$_` (листинг 8.28).

Листинг 8.28. Пример использования оператора `foreach` и переменной по умолчанию `$_`

```
#!/usr/bin/perl -w
@array = (5, -7, 25, 1, -9, 7, -3);
```

```
foreach (@array)
 {
  print "$_ ";
 }
```

8.3.6. Регулярные выражения

При обработке массивов данных и текстовых файлов часто возникает задача поиска элементов, удовлетворяющих заданным условиям, и выполнения над ними определенных операций. Условия поиска можно выразить посредством так называемых *регулярных выражений*, позволяющих описать образец или шаблон поиска при помощи специальных правил, а манипуляции с регулярными выражениями осуществляются при помощи соответствующих операций.

Регулярное выражение представляет собой набор правил для описания текстовых строк, а сами правила записываются в виде последовательности обычных символов и так называемых *метасимволов*. Метасимволы представляют собой символы, имеющие в регулярном выражении специальное значение:

```
\ . ^ $ | [ ( ) * + ? { }
```

Каждый метасимвол имеет свой смысл, например, он может использоваться для обозначения одиночного символа или их группы, обозначать привязку к определенному месту строки, число возможных повторений отдельных элементов, возможность выбора из нескольких вариантов и т. д. Регулярное выражение строится с соблюдением определенных правил, причем в нем можно выделить операнды и операции.

Проанализируем назначение отдельных метасимволов. Так, открывающая и закрывающая круглые скобки `()` выполняют группирование элементов, а метасимвол `*` означает, что находящийся перед ним элемент или группа элементов могут встречаться один раз, много раз или не встречаться вообще. Метасимвол `+` указывает на то, что стоящий перед ним элемент или группа элементов встретятся хотя бы один раз, а знак вопроса `?` означа-

ет, что элемент может отсутствовать или встретиться только один раз.

Далее приводятся примеры регулярных выражений с использованием перечисленных метасимволов:

- `/fr.*nd/` — этому выражению удовлетворяют строки "frnd", "friend", "front and back";
- `/fr.+nd/` — этому выражению удовлетворяют строки "frond", "friend", "front and back", но не "frnd";
- `/10*1/` — этому выражению удовлетворяют строки "11", "101", "1001", "100000001";
- `/b(an)*a/` — этому выражению удовлетворяют строки "ba", "bana", "banana", "bananananana";
- `/flo?at/` — этому выражению удовлетворяют строки "flat" and "float", но не "flooat".

Метасимвол "квадратные скобки" `[]` указывает на группу отдельных символов. Примеры регулярных выражений с квадратными скобками:

- `[0123456789]` — этому выражению удовлетворяет любая цифра из диапазона 0—9;
- `[0-9]` — этому выражению удовлетворяет любая цифра из диапазона 0—9;
- `[0-9]+` — этому выражению удовлетворяет любая последовательность цифр;
- `[a-z]+` — этому выражению удовлетворяет любое слово из символов нижнего регистра;
- `[A-Z]+` — этому выражению удовлетворяет любое слово из символов верхнего регистра;
- `[ab n]*` — этому выражению удовлетворяет строка нулевой длины "", "b", любое количество пробелов, а также строка "nab a banana";
- `[^...]` — этому выражению удовлетворяют символы, не равные "...";

- `[^0-9]` — этому выражению удовлетворяет любой символ, не являющийся цифрой.

Диапазон значений, определяемых регулярным выражением, можно сузить, если использовать метасимволы фигурных скобок `{}`. Например, выражению `[0-9]{6}` удовлетворяет любая последовательность из 6-ти цифр, а выражению `[0-9]{6,10}` — любая последовательность от 6-ти до 10-ти цифр.

Метасимволы `^` и `$` означают начало и конец строки соответственно (символ `^` должен находиться за квадратными скобками `[]`). Вот как можно их комбинировать в регулярных выражениях:

- `/at/` — этому выражению удовлетворяют "at", "attention", "flat" и "flatter";
- `/^at/` — этому выражению удовлетворяют "at" и "attention", но не "flat";
- `/at$/` — этому выражению удовлетворяют "at" и "flat", но не "attention";
- `/^at$/` — этому выражению удовлетворяет только "at";
- `/^at$/i` — этому выражению удовлетворяют "at", "At", "aT" и "AT";
- `/^[ \t]*$/` — этому выражению удовлетворяет пустая строка или любая комбинация пробелов и символов табуляции.

Следует упомянуть об одной важной особенности использования символов `+`, `?`, `.`, `*`, `^`, `$`, `()`, `[]`, `{}`, `|`, `\`. Если перед ними стоит символ `\`, специальное назначение для этих символов ликвидируется, и они воспринимаются как литералы (в общем случае нежелательно также располагать символ `\` перед `/`). Следующие примеры демонстрируют эту особенность:

- `/10.2/` — этому выражению удовлетворяют "10Q2", "1052" и "10.2";
- `/10\.2/` — этому выражению удовлетворяют "10.2", но не "10Q2" или "1052";
- `/\*+/` — этому выражению удовлетворяют один или более символов `*`;

- `/A:\\DIR/` — этому выражению удовлетворяет "A:\DIR";
- `/\/usr\/bin/` — этому выражению удовлетворяет "/usr/bin".

Если символ `\` предшествует алфавитно-цифровому символу, то такая комбинация имеет специальное назначение, представляя собой сокращенную форму выражения в квадратных скобках `[]`.

Например, выражение `\d` эквивалентно `[0-9]`, а `\s` обозначает пробел и эквивалентно выражению `[ \t\n\r\f]`. Символ в таком выражении может быть представлен и шестнадцатеричным значением, например, `\x1b` означает ESC.

Легче всего манипуляции с регулярными выражениями понять на примерах.

Пример 1:

```
#!/usr/bin/perl -w
$str = "Quck brown   fox jumped    over lazy dog";
if ($str =~ /fox/)
 {
  print "fox is found\n";
 }
```

Здесь в строковой переменной `$str` ищется слово `fox`, и если оно обнаружено, то на экране печатается соответствующее сообщение. Для поиска (замены) выражения используется специальный оператор `=~`.

Пример 2:

```
#!/usr/bin/perl -w
$str = "Quck brown   fox jumped    over lazy dog";
$str =~ s/ +/+/gi;
print "$str\n";
```

В этом примере используется строковая переменная `$str` с тем же значением, что и в предыдущем примере. Цель программы — заменить все пробелы между словами данной строки на единичный символ `+`. Обратите внимание на оператор

```
$str =~ s/ +/+/gi;
```

Здесь символом `s` в правой части выражения обозначен оператор замены (substitution), а выражение `" +"` указывает на один или несколько следующих подряд пробелов.

Вот результат работы такой программы (она называется regexpr_demo2.pl):

```
# ./regexpr_demo2.pl
Quck+brown+fox+jumped+over+lazy+dog
```

Пример 3. Это самый сложный пример по сравнению с предыдущими. Здесь на экран дисплея нужно вывести те элементы строкового массива `@str_array`, которые одновременно удовлетворяют двум условиям:

- строка не должна начинаться с цифры;
- строка не должна начинаться с литер "Fi".

Исходный текст примера представлен далее:

```
#!/usr/bin/perl -w
@str_array = ("First", "2-nd", "3-rd", "Fourth", "Fifth",
"Sixth");


foreach (@str_array)
 {
  print "$_\n" if (!/^\d/ && !/^Fi+/);
 }
```

Результат работы программы (она называется regexpr_demo3.pl) будет таким:

```
# ./regexpr_demo3.pl
Fourth
Sixth
```

8.3.7. Обработка файлов и каталогов

Поскольку файлы занимают центральное место в обработке данных, то в языке Perl предусмотрен весьма широкий набор

функций для манипуляций ими. При этом стандартные задачи (открытие файлов, чтение и запись данных) используют простые функции ввода/вывода, а для решения более сложных задач, таких, например, как асинхронная обработка данных и блокировка файла, разработаны и применяются более сложные функции. Здесь мы рассмотрим наиболее широко используемые функции для манипуляций файлами.

Доступ к файлам в Perl осуществляется посредством файловых манипуляторов — символических имен, представляющих данный файл в операциях чтения/записи. Эти символические имена на самом деле ссылаются на дескрипторы файлов, с которыми работает операционная система. Файловые дескрипторы используются во всех системных вызовах (`open()`, `read()`, `write()`, `close()`), но работать с ними сложнее, в то время как операции с файловыми манипуляторами не вызывают затруднений. Открыть файл можно с помощью функции `open`, имеющей синтаксис:

`open(`*`файловый_манипулятор`*`, "[`*`тип_операции`*`]`*`путевое_имя_файла`*`")`

Здесь *`файловый_манипулятор`* указывает на файловый манипулятор, *`тип_операции`* определяет, какая операция (чтение, запись, чтение/запись) будет выполняться над файлом, а *`путевое_имя_файла`* указывает путь к файлу.

Если обозначить файловый манипулятор как `FH`, а путевое имя файла как `$path`, то функцию `open` можно представить в более доступной для понимания форме:

- `open(FH,"$path")` — файл открывается для чтения;
- `open(FH,">$path")` — файл открывается для записи;
- `open(FH,">+$path")` — файл открывается для чтения и записи;
- `open(FH,">>$path")` — файл открывается для дозаписи.

Содержимое файла можно прочитать одним из двух способов. В первом случае можно использовать следующий оператор:

```
@data = <FH>
```

Здесь `FH` — файловый манипулятор.

Во втором случае (который часто используется для чтения текстовых файлов с разделителями строк) можно читать строку за строкой в цикле `while`. Далее показаны оба способа чтения файлов.

Пример 1:

```
#!/usr/bin/perl -w
$file = shift or die "Usage: $0 path";
open(FH, "$file") or die "open: $!";
@data = <FH>;
close(FH);
print @data;
```

Программа выводит на экран содержимое файла, путевое имя которого задается в качестве единственного параметра командной строки, который помещается в переменную `$file`. Если параметр командной строки отсутствует, программа завершается (функция `die`). Здесь `$0` — специальная переменная Perl, которая указывает на имя исполняемого файла программы.

По завершению файловых операций обязательно следует закрыть файловый манипулятор при помощи функции `close`, параметром которой является файловый манипулятор.

Пример 2:

```
#!/usr/bin/perl -w
$file = shift or die "Usage: $0 path";
open(FH, "$file") or die "open: $!";
while (<FH>)
 {
  print ":$_";
 }
close(FH);
```

В этом примере чтение содержимого файла и вывод на экран выполняется строка за строкой в цикле `while`, причем для этого

используется переменная по умолчанию `$_`, в которую помещаются данные из файла.

Следующие два примера более сложные и показывают, как можно скопировать или переместить файлы.

Пример 3:

Программа копирования файлов представлена исходным текстом, показанным далее:

```
#!/usr/bin/perl -w
if ($#ARGV+1 != 2)
 {
   die "Usage: $0 src dst\n";
  }
$src = $ARGV[0];
$dst = $ARGV[1];
open(FSRC, "$src") or die "open: $!";
@data = <FSRC>;
close(FSRC);
open(FDST, "+>$dst") or die "open: $!";
print FDST @data;
close(FDST);
```

Программа принимает два параметра командной строки. Первый параметр соответствует зарезервированной переменной `$ARGV[0]` языка Perl и представляет собой путевое имя файла, который следует скопировать. Второй параметр является путевым именем файла, в который будет выполнено копирование (переменная `$ARGV[1]`). Файловый манипулятор файла-источника обозначен как `FSRC`, а файловый манипулятор файла-приемника — `FDST`. После чтения содержимого исходного файла в переменную `@data` выполняется запись данных в манипулятор `FDST`, для чего используется обычная функция `print`:

```
print FDST @data;
```

Смысл остальных операторов программы, думаю, понятен, и объяснять его не нужно.

Программа перемещения файлов отличается от только что рассмотренной программы копирования только тем, что после завершения операции исходный файл удаляется при помощи функции `unlink`, принимающей в качестве параметра путевое имя файла-источника.

Пример 4:

```
#!/usr/bin/perl -w
if ($#ARGV+1 != 2)
 {
   die "Usage: $0 src dst\n";
  }
$src = $ARGV[0];
$dst = $ARGV[1];
open(FSRC, "$src") or die "open: $!";
@data = <FSRC>;
close(FSRC);
open(FDST, "+>>$dst") or die "open: $!";
print FDST @data;
close(FDST);
unlink($src);
```

Мы рассмотрели основные методы работы с файлами в языке Perl. Хочу упомянуть еще одну функцию, позволяющую переименовать файл — `rename`, имеющую синтаксис:

`rename(`*текущее_имя_файла*`, `*новое_имя_файла*`)`

Смысл параметров функции, думаю, понятен из их названия.

Следующие операторы позволяют выполнить проверки объектов файловой системы:

- `-e` *файл/каталог* — проверка существования файла/каталога;
- `-z` *файл* — проверка файла на предмет нулевого размера;

- `-d` *файл* — проверка того, является ли объект файловой системы каталогом;
- `-s` *файл* — проверка размера файла (в байтах).

Рассмотрим особенности работы с каталогами в языке Perl. Напомню, что каталог представляет собой файл специального формата, помеченный в индексном узле как каталог. Каталог содержит информацию о других объектах файловой системы, находящихся в нем, и этим отличается от файла, содержащего только двоичные данные. Для работы с содержимым каталога в языке Perl имеется несколько функций:

- `opendir(DIR, $`*имя_каталога*`)` — открывает каталог с именем *имя_каталога* и присваивает ему манипулятор каталога `DIR`;
- `readdir(DIR)` — читает содержимое каталога с манипулятором `DIR`;
- `closedir(DIR)` — закрывает каталог с манипулятором `DIR`.

Манипуляции содержимым каталога рассмотрим на примере, исходный текст которого представлен в листинге 8.29. В этом примере на экран выводится содержимое рабочего каталога приложения, за исключением файлов "." и "..".

Листинг 8.29. Вывод содержимого каталога на экран дисплея

```
#!/usr/bin/perl -w
use Cwd;
$workdir = &cwd;
opendir(DIR, $workdir) or die "opendir: $!";
while (defined($file = readdir(DIR)))
 {
   next if ($file =~ /^\./);
   print "$file\n";
 }
closedir(DIR);
```

Чтение каталога и вывод информации осуществляется в цикле `while`, который выполняется до тех пор, пока не будет обработан последний элемент каталога. Оператор

```
next if ($file =~ /^\./);
```

пропускает вывод имени файла, если оно начинается с точки (.) или двоеточия (..).

По окончанию обработки каталога он закрывается посредством функции `closedir`.

Заканчивая обзор файловых операций, хочу заметить, что ввиду ограниченности объема книги мы рассмотрели только часть возможностей языка Perl по работе с файлами и каталогами. Намного больше информации можно найти в документации, расположенной на различных интернет-сайтах, в частности, на **www.perl.org**.

Последняя тема, которой мы уделим внимание, — сетевое программирование на языке Perl.

8.3.8. Сетевое программирование в Perl

Одной из замечательных особенностей языка Perl является возможность легкой разработки сетевых программ. Программный код таких приложений, в отличие, например, от языка C, намного более читабельный и скрывает многие детали программирования, что значительно облегчает создание программ. Здесь будут вкратце рассмотрены общие аспекты сетевого программирования — более подробная информация находится в документации по языку Perl.

При разработке сетевой программы в нее следует включить модуль `Socket` (оператор `use Socket`), содержащий все необходимые функции.

Рассмотрим две программы. Одна из них является сервером TCP и работает с портом 5151, читая данные, передаваемые программой-клиентом, и выводя их на экран консоли. Обращаю внимание читателей на то, что как программа-сервер, так и програм-

ма-клиент работают на одной машине, используя интерфейс обратной связи с адресом 127.0.0.1, поэтому их легко проверить и отладить. Исходный текст программы-сервера представлен в листинге 8.30.

Листинг 8.30. Сервер TCP

```
#!/usr/bin/perl -w
use Socket;
$host = inet_aton("127.0.0.1");
$port = 5151;
$addr = sockaddr_in($port, $host);
socket(SERVER, AF_INET, SOCK_STREAM, getprotobyname("tcp"))
      or die "socket: $!";
bind(SERVER, $addr) or die "bind: $!";
listen(SERVER, 10);
print "Waiting for requests...\n";
while (1)
{
 if (accept(NEW_CON, SERVER))
  {
   while (<NEW_CON>)
    {
      print "$_\n";
    }
   close(NEW_CON);
  }
}
close(SERVER);
```

В этой программе используются уже знакомые из предыдущих глав сетевые функции `socket`, `bind`, `accept` и `listen`. Эти функции имеют тот же смысл, что и соответствующие библиотечные

функции языка C (в System V) или системные вызовы (BSD-совместимые системы). Хочу обратить внимание на то, что чтение данных из сокета выполняется так же, как и для обычных файловых манипуляторов.

Программа-клиент намного проще (листинг 8.31).

Листинг 8.31. Программа-клиент TCP

```
#!/usr/bin/perl -w
use Socket;
$host = inet_aton("127.0.0.1");
$port = 5151;
$addr = sockaddr_in($port, $host);
socket(CLIENT, AF_INET, SOCK_STREAM, getprotobyname("tcp"))
      or die "socket: $!";
connect(CLIENT, $addr) or die "connect: $!";
print CLIENT "Test String for TCP server\n";
close(CLIENT);
```

В программе-клиенте используются сетевые функции `socket` и `connect` с соответствующими параметрами. Передача данных программе-серверу выполняется посредством функции `print`.

Для проверки работы приложений следует открыть два окна консоли, после чего в одном из них запустить программу-сервер, затем в другом — программу-клиент.

Глава 9

Графический интерфейс пользователя

Операционные системы UNIX в течение многих лет развивались как системы, взаимодействующие с пользователем посредством текстовой консоли. Все изменилось с появлением графических интерфейсов, в частности, графической системы X Window, являющейся базисом для всех современных графических оболочек.

Система X Window более известна под названием X11 или X и была разработана как оконный интерфейс для графических (GUI) приложений UNIX, использующих растровые дисплеи.
X включает набор стандартных библиотек и работает в соответствии со стандартным протоколом, специально разработанным для этой системы. X включается в стандартный комплект поставки и других UNIX-подобных операционных систем. Хочу отметить тот факт, что все современные операционные системы UNIX поддерживают стандарты X Window.

X обеспечивает базовый набор функций для графического интерфейса пользователя, позволяющих рисовать и перемещать окна на экране дисплея, а также взаимодействовать с клавиатурой и мышью. При этом система X Window не является графическим интерфейсом пользователя, она включает в себя функции и протоколы, позволяющие создавать графический интерфейс клиентским программам. Клиентские программы могут использовать возможности, предоставляемые X, обеспечивая какие угодно интерфейсы. С точки зрения программной архитектуры

X не интегрирована в операционную систему UNIX, а работает поверх нее, так же как и другие сервисы.

Система X Window является, по сути, сетевой системой, прозрачной как для локальных, так и для сетевых клиентов. При этом "сервер" и "клиент" отличаются от привычных для пользователей представлений: обычно программа-клиент выполняется на локальном компьютере и запрашивает услуги удаленного сервера в сети. Для X все наоборот — сервер находится на локальной машине, а клиент — на удаленной.

Система X Window была разработана более 20 лет назад и в настоящее время использует стандарт X11. Сам проект X Window в настоящее время сопровождает организация XOrg Foundation, выпустившая систему как свободно распространяемое программное обеспечение. Поскольку X является открытым стандартом, то она может функционировать на различных платформах, не только UNIX.

9.1. Архитектура системы X Window

Архитектура X базируется на принципе модульности, который обеспечивает высокую степень функциональности. Каждый модуль или, по-другому, компонент системы выполняет строго определенную функцию. Одним из важнейших компонентов системы X Window является *оконный менеджер* (window manager), определяющий внешний вид и возможности интерфейса системы. Интерфейс пользователя при таком подходе изменить очень легко — достаточно поменять один оконный менеджер на другой. В настоящее время существует несколько десятков различных оконных менеджеров, начиная от самых простых (twm) до очень сложных, в которых реализованы целые интегрированные системы, такие, например, как KDE и GNOME.

Система X создавалась для работы в сети и использует модель "клиент-сервер": пользователь работает с сервером (он называется X-сервером), а программы, с которыми взаимодействует X-сервер, называются клиентами (X-клиенты).

X-сервер является отдельным UNIX-процессом, взаимодействующим с программами-клиентами посредством пакетов данных. Если сервер и клиент находятся на разных машинах, то обмен данными осуществляется по сети, а если на одной, то используется механизм межпроцессных коммуникаций (IPC).
X-сервер предоставляет свои ресурсы программам-клиентам, обеспечивающим всю остальную функциональность. Функции сервера включают управление дисплеем, обработку ввода с клавиатуры и мыши и т. д. Это означает, что существует принципиальная возможность использования ресурсов разных вычислительных машин без специальных методов при разработке программного обеспечения.

X-сервер принимает запросы от оконных интерфейсов, возвращая параметры ввода от таких устройств, как клавиатура и мышь. Он может одновременно обслуживать многие компьютеры, обладая при этом собственной вычислительной мощностью, что позволяет ему выполняться даже на рабочей станции. При использовании X Window сервер работает на компьютере пользователя, в то время как клиентские программы могут функционировать на других машинах. Такая конфигурация является несколько необычной, поскольку в большинстве реализаций схемы "клиент-сервер" клиент выполняется на машине пользователя, в то время как сервер работает на удаленном хосте.

В терминологии X Window удаленные программы вызывают X-сервер на локальной машине и считаются клиентами, а локальный X-дисплей принимает входящие запросы, поэтому работает как сервер. На локальном компьютере сервер и клиенты X работают одновременно. Следует четко уяснить себе, что клиентами являются все программы, создающие окна.

Схема взаимодействия клиентов и сервера в системе X Window показана на рис. 9.1.

В данном примере X-сервер принимает запросы двух клиентов (Web-браузера и эмулятора терминала), работающих на той же машине, а также запросы двух удаленных X-клиентов.

Взаимодействие клиентов с X-сервером осуществляется посредством протокола обмена, известного под названием X-прото-

кола. Он обеспечивает стандартный метод обмена данными, не зависящий от характера установленного между клиентом и сервером соединения (локальное или сетевое), а это означает, что передача-прием данных является "прозрачным" как для клиента, так и сервера.

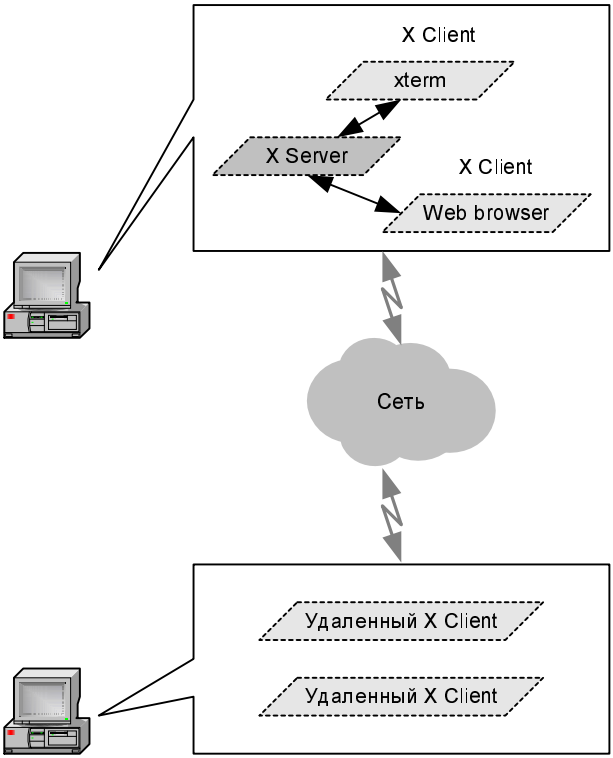


Рис. 9.1. Реализация X Window в архитектуре "клиент-сервер"

Взаимодействие между X-сервером и клиентом осуществляется посредством обмена пакетами по программному каналу или по сети. Клиент устанавливает соединение с сервером, затем посылает ему первый пакет. Сервер отвечает клиенту пакетом, указы-

вающим либо на отказ в установлении соединения, либо разрешением на создание соединения и дополнительным запросом для дальнейшей аутентификации. Если соединение разрешено, то очередные пакеты содержат данные для последующего взаимодействия с сервером.

После установки соединения между клиентом и сервером для обмена информацией используются три типа пакетов:

- запрос — клиент запрашивает информацию с сервера или требует выполнения определенного действия;
- ответ — сервер отвечает на запрос клиента; следует отметить, что не на все запросы даются ответы;
- извещение о событии — сервер информирует клиента о наступлении определенного события, например, о вводе информации с клавиатуры, о нажатии кнопки мыши или о перемещении окна и изменении его размера. Во многих случаях клиент может потребовать от X-сервера сообщить о наступлении какого-либо события другому клиенту — подобным образом осуществляется взаимодействие между клиентами. Например, когда программа-клиент запрашивает обработку выделенного фрагмента текста, информация об этом событии отправляется другой программе-клиенту, которая в данный момент управляет окном с находящимся там текстом. Информация о некоторых типах событий всегда отправляется клиенту, но в большинстве случаев такая информация отправляется только в том случае, если клиент предварительно объявил, что заинтересован в ней. Например, клиента могут интересовать только события, связанные с нажатием клавиш, и не интересовать события, связанные с мышью.

X-протокол является асинхронным — как X-клиент, так и X-сервер начинают выполнять следующую операцию, не ожидая завершения предыдущей. Если по какой-либо причине соединение между клиентом и сервером обрывается, то в соответствии с X-протоколом выполняется сброс установок с потерей состояния сеанса в момент обрыва. Подобные ситуации ограничивают применение базового варианта X Window в беспроводных системах из-за частых замираний и пропаданий сигнала.

Все, что вы наблюдаете на экране монитора, является результатом взаимодействия различных компонентов: операционной системы, X Window, оконного менеджера и, возможно, какой-нибудь графической оболочки, наподобие GNOME или KDE. При этом отдельные компоненты (оконный менеджер, графическая оболочка и сама система X) могут настраиваться автономно от остальных, что обеспечивает высокую гибкость в настройке интерфейса пользователя, но одновременно усложняет задачу настройки всей системы.

Спецификации протокола X Window не обязывают машины клиента и сервера работать под управлением одной и той же операционной системы или даже быть одним и тем же типом компьютера. Существует возможность запускать X-сервер в Microsoft Windows или Mac OS, и есть множество свободно распространяемых и коммерческих приложений, которые это реализуют.

Программирование приложений, работающих в X Window, является достаточно простым, поскольку система предоставляет для этих целей стандартную библиотеку процедур. Например, для вывода на экран точки достаточно вызвать соответствующую стандартную библиотечную процедуру, передав ей требуемые параметры: процедура выполнит всю работу по формированию пакетов данных и передаче их X-серверу.

Программный интерфейс X-сервера с соответствующим аппаратно-программным обеспечением образует в терминах X Window "дисплей", в качестве которого может выступать, например, обычный компьютер. Дисплей выполняет программный код X-сервера, обслуживая клавиатуру и позиционирующее устройство, например, мышь. Дисплей поддерживает множества физических "экранов" — аппаратно-программных комбинаций графических подсистем (видеокарта, графический акселератор) и мониторов, создающих растровое изображение. Количество обслуживаемых экранов в текущих реализациях X Window ограничено значением 231, что является максимальным значением для 32-разрядных машин.

Система X Window предназначена для работы на растровых дисплеях, изображение в которых представляется матрицей светящихся точек — пикселов, каждый из которых кодируется определенным числом битов (как правило, 2, 4, 8, 16 или 24). Количество битов на пиксел называют *толщиной* или *глубиной дисплея*. Биты с одинаковыми номерами во всех пикселах образуют как бы плоскость, параллельную экрану, которую называют *цветовой плоскостью*. X позволяет рисовать в любой цветовой плоскости (или плоскостях), не затрагивая остальные.

Значение пиксела не задает непосредственно цвет точки на экране — он определяется с помощью специального массива данных, называемого *палитрой*. Цвет представлен содержимым ячейки палитры, номер которой равен значению пиксела.

X имеет большой набор процедур, позволяющих рисовать графические примитивы — точки, линии, дуги, выводить текст и работать с областями произвольной формы.

Каждый экран дисплея может содержать множество перекрывающихся окон — базовых элементов системы, представляющих собой прямоугольные или произвольной формы (с учетом расширений системы X Window) отображаемые области растровой памяти. Система X Window использует окно для вывода графической информации, предоставляя его программе-клиенту.

Любое созданное окно должно иметь родительское окно — таким образом создается иерархия или дерево окон. В вершине этой иерархии находится так называемое *корневое окно* (root window), которое автоматически создается X-сервером. Визуально корневое окно занимает всю площадь экрана и находится позади всех остальных окон, а сами окна могут располагаться на экране произвольным образом, перекрывая друг друга.

Один из примеров иерархии окон показан на рис. 9.2.

На этой схеме (1) — корневое окно, занимающее экран целиком, (2) и (3) — окна переднего плана, а (4) и (5) являются подокнами окна (2). При этом части некоторых окон (3 и 5) выходят за пределы родительского окна, поэтому их не видно (пунктирная линия).

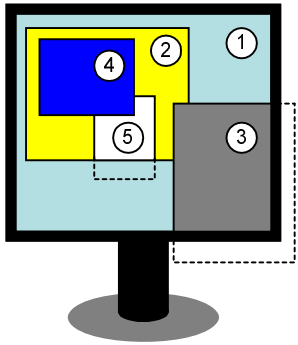


Рис. 9.2. Пример иерархии окон в X Window

С каждым окном связана система координат, начало которой находится в левом верхнем углу окна — при этом ось *x* направлена вправо, ось *y* вниз, а единицей измерения по обеим осям является пиксел. Окно имеет внутреннюю область и рамку, а его основными атрибутами являются ширина и высота внутренней области, а также ширина (толщина) рамки. Все перечисленные параметры называют *геометрией окна*. Концепция окна положена в основу представления большинства графических элементов управления, таких как кнопки, меню, значки и др.

Здесь хотелось бы отметить одну важную особенность X Window: если не используются дополнительные программы-клиенты (оконные менеджеры или графические оболочки), то X-сервер при запуске создает корневое окно без рамок, значков и всплывающих меню. Пример корневого окна вместе с окнами других X-клиентов в оболочке Cygwin (эмулятор UNIX) представлен на рис. 9.3.

Здесь для создания корневого окна (серый цвет) была использована команда

```
$ xinit -g 60x20+150+150 -fn 10x20 -bg black -fg white -
display :0
```

При этом был запущен X-сервер на дисплее 0, и, поскольку `xinit` автоматически вызвала программу эмулятора терминала

`xterm` с указанной геометрией и параметрами шрифта и цветов, было создано окно приложения `xterm` (черный цвет) с указанной геометрией. После этого из окна эмулятора `xterm` было запущено приложение редактора текста `emacs` с указанной геометрией:

```
$ emacs -g 50x30+250+250
```

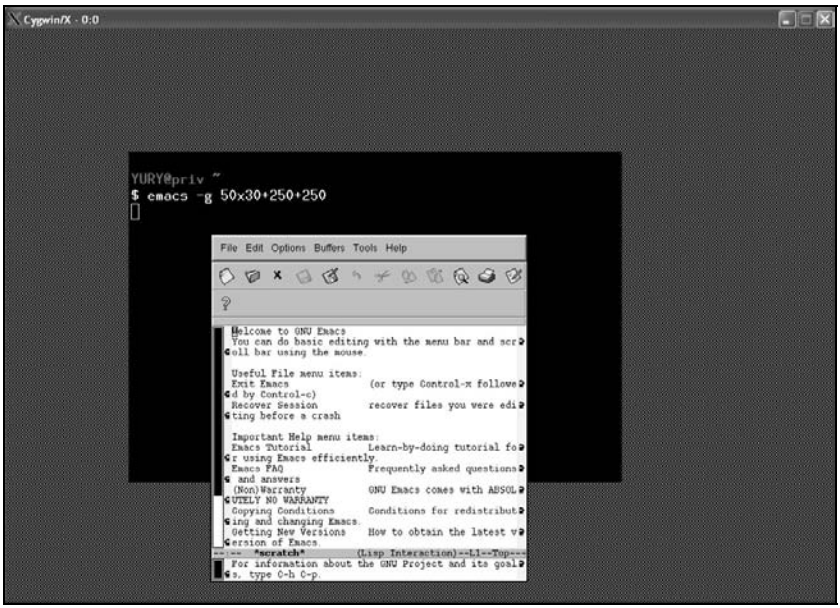


Рис. 9.3. Пример иерархии окон, созданных при помощи X-сервера

Программный интерфейс X Window позволяет программе-клиенту выполнять целый ряд операций как с окнами, так и с их содержимым — окна можно создавать, перерисовывать на экране, удалять, в них можно выводить текст и графические примитивы. Особенностью системы является то, что в ней не предусмотрен встроенный механизм для управления окнами с помощью мыши или клавиатуры — эти функции выполняет

оконный менеджер. Завершить работу в X Window можно несколькими способами. Можно закончить сеанс, нажав кнопку завершения на панели инструментов виртуального рабочего стола. Быстрый выход из X Window — нажатие комбинации клавиш <Ctrl>+<Alt>+<Backspace>.

Для установки соединения с X-сервером и инициализации начальных установок для клиентских программ можно применить один из трех методов.

- Использовать менеджер дисплея `xdm`, управляющий бесперебойным функционированием X-сервера. Кроме того, `xdm` позволяет пользователю войти в систему — если он запущен, то на экране будет отображаться приглашение к вводу имени и пароля. После успешного входа в систему `xdm` создает окружение для работы графической подсистемы. Если в исходном (домашнем) каталоге пользователя есть файл .xsession, то менеджер `xdm` интерпретирует данный файл как командный и использует его для запуска X-клиентов (эмуляторов терминала, часов, оконного менеджера, пользовательских настроек, например, фона рабочего стола и т. д.).
- Использовать программу `xinit` (если таковая включена в систему). Этот вариант запуска графической системы применяется в том случае, когда в операционной системе имеется несколько оконных систем. Программа вызывается из командного интерпретатора и, в свою очередь, запускает дополнительные утилиты, выполняющие специфичную для данной системы инициализацию (загрузку необходимых ресурсов, запуск оконного менеджера, отображение часов, запуск эмуляторов терминала оптимальным образом).

  Настройки X-клиентов содержатся в соответствующих файлах конфигурации. При запуске система X Window считывает файлы конфигурации (как пользовательские, так и общесистемные) и запускает указанные пользователем X-клиенты. Даже если X Window по каким-либо причинам не запускается автоматически при входе в систему, можно запустить ее из командной строки в окне терминала, работающего в текстовом режиме, выполнив команду `startx`. Если появляется со-

общение, что такая команда отсутствует, нужно воспользоваться командой `xinit`.

Если же ни одна из этих команд не работает, возможно, в операционной системе не установлена X Window. В этом случае придется подробно изучить документацию на используемый дистрибутив операционной системы.

- Использовать команду `startx`. Эта программа имеет следующий синтаксис:

  `startx [ [client] options ... ] [ -- [server] options ... ]`

  Команда `startx`, помимо других операций по инициализации, запускает на исполнение двоичный файл `xinit`, который, как мы знаем, запускает X-сервер. Поскольку `startx`, кроме вызова `xinit`, выполняет и другие необходимые действия, то запуск системы X Window с помощью этой команды является предпочтительным. Если данная команда не запускается, то это может быть вызвано одной из трех причин:

  - система X Window вообще не установлена;
  - имеются технические неполадки в X, препятствующие запуску X-сервера;
  - ошибка в конфигурации (например, установлены такие параметры, которые не соответствуют возможностям графической подсистемы).

Если дополнительные программы отсутствуют, то вы можете создать командный файл, использующий утилиту `xinit`, самостоятельно. Естественно, что разработать командный файл для запуска X-сервера и всех клиентов может только опытный пользователь.

В зависимости от конфигурации системы X Window может стартовать сразу же после загрузки операционной системы или запускаться самим пользователем. Сеанс работы пользователя в системе X Window начинается после запуска X-сервера или ввода пользовательского идентификатора и пароля в специальную форму менеджера экрана системы X Window (`xdm`, `kdm`, `gdm` или другой) и заканчивается после завершения работы X-сервера.

Сеанс работы в самой операционной системе UNIX при этом может закончиться (при наличии менеджера экрана) или продолжиться в режиме консоли.

Все программы, запускающие систему X Window, используют те или иные файлы конфигурации. Эти файлы используются командами `startx` и `xinit`, при этом наиболее важным из них является файл .xinitrc.

Файл .xinitrc является командным, т. е. исполняемым, и передается программе `xinit` командой `startx`. В этом файле содержатся настройки некоторых общих (глобальных) ресурсов, таких как хранитель экрана, клавиатура и др., поэтому он является основным для всего процесса запуска.

Для запуска X может понадобиться и файл Xclients. Он находится в домашнем каталоге пользователя и содержит командные строки для запуска клиентов, специфичных для данного пользователя, в отличие от .xinitrc, содержащего общие настройки.

Файл Xresources содержит программные настройки, позволяющие изменить заданные по умолчанию значения. Он обычно применяется для определения размеров шрифтов, цветов и общего внешнего вида. Кроме этого, его можно использовать для получения графических эффектов.

Проанализируем процесс запуска X Window более подробно, предположив, что запуск осуществляется посредством команды `startx`. Программа `startx` передает управление `xinit`, выполняющей основную часть процесса запуска X.

Программа `xinit` просматривает файл /etc/X11/xinit/.xinitrc (глобальный) либо файл ~/.xinitrc (локальный), если он существует. Если ни один из этих файлов не обнаружен, то `xinit` выбирает настройки самостоятельно.

Если командный файл .xinitrc существует, то ему передается управление, после чего .xinitrc запускает X-клиентов. Последняя строка этого командного файла содержит исполняемый файл оконного менеджера, который и завершает процесс запуска X — обычно здесь присутствует команда `exec`, запускающая оконный

менеджер. Следует помнить, что приоритет локального файла .xinitrc выше, чем у глобального.

В простейшем варианте файл .xinitrc может содержать всего три строки:

```
# /etc/X11/xinit/.xinitrc
# !/bin/sh
exec /usr/X11R6/bin/fvwm2
```

Здесь выполняется только запуск оконного менеджера `fvwm2` (Feeble Virtual Window Manager 2).

При редактировании командного файла .xinitrc любая команда, кроме той, что запускает оконный менеджер, должна завершаться символом амперсанда `&`. Это гарантирует, что все программы будут выполняться в фоновом режиме, не блокируя запуск последующих команд. Если этого не сделать, следующая команда может не выполниться вообще. Это не относится к оконному менеджеру, который, как мы знаем, запускается последним — он должен выполняться в основном режиме, что позволяет завершить все процессы, запущенные через .xinitrc, при выходе из X.

Последняя программа, включенная в командный файл, обычно является или эмулятором терминала, или оконным менеджером и должна запускаться в основном режиме — в этом случае командный файл не закончит работу, так что пользователь будет продолжать работать в графическом режиме, пока не захочет из него выйти.

Пользователь может установить индивидуальные настройки интерфейса X Window, указав, например, определенные параметры экрана и клавиатуры. Эти настройки будут выполняться автоматически всякий раз при запуске X Window.

Вот пример исходного текста такого модифицированного командного файла .xinitrc:

```
#/etc/X11/xinit/.xinitrc
#!/bin/sh
```

```
# Выполним установку клавиши <Backspace>
xmodmap -e "keycode 22=BackSpace"

# Установим фон
xsetroot -solid LightSlateGrey

# Хранитель экрана должен отработать через 10 минут
xset s 600

# Запуск X-клиентов
xterm -g 80x20+150+8 &        # запуск окна терминала
xterm -g 80x20+150+325 &      # запуск второго окна под первым
xload -g +4+0 &               # индикатор загрузки процессора
xclock -g +815+0 -digital &   # запуск цифровых часов

# Запуск оконного менеджера
exec fvwm2
```

С помощью этого файла устраняется проблема замены функции клавиши <Backspace> (в некоторых системах ее функции отличаются от стандартных), удаляются муаровые "обои" и запускаются две терминальные программы.

Следует отметить, что локальный файл .xinitrc из домашнего каталога пользователя будет использоваться всегда, при этом общесистемные установки не берутся во внимание — напомню, что установки в локальном файле имеют высший приоритет по сравнению с глобальными.

Как видно из листинга, для многих X-клиентов используется параметр `-g`, позволяющий определять геометрию окна.

Одним из существенных преимуществ системы X Window является то, что она никаким образом не ограничивает приложения ни в плане размещения окон, ни их размерами на экране, что не всегда получается, если использовать аппаратные терминальные соединения. Несмотря на то, что расположением окон на дис-

плее управляет оконный менеджер, большинство приложений, работающих в X Window, могут устанавливать параметры окон из командной строки в формате

```
-g Ширина × Высота + Координата X + Координата Y
```

Параметры *Ширина*, *Высота*, *Координата X* и *Координата Y* задаются в виде чисел и определяют размер и размещение главного окна приложения.

Параметры *Ширина* и *Высота* обычно определяются или в пикселах, или в символах, в зависимости от особенностей работы приложения. Параметры *Координата X* и *Координата Y* измеряются в пикселах и используются для позиционирования окна от левой/правой или верхней/нижней границ экрана соответственно.

Смещение *Координата X* можно задавать следующим образом:

- `+XOF` — левая кромка окна размещается на расстоянии `XOF` пикселов от левого края экрана. Иными словами, это значение представляет собой координату *x* данного окна. Следует сказать, что `XOF` может быть отрицательным числом — в этом случае левая кромка окна будет находиться вне экрана;
- `-XOF` — правая кромка окна размещается на расстоянии `XOF` пикселов от правого края экрана. `XOF` может быть отрицательным числом — в этом случае правая кромка окна будет находиться вне экрана.

Смещение по вертикали определяется следующими значениями:

- `+YOF` — верхняя кромка окна размещается на расстоянии `YOF` пикселов ниже от верхнего края экрана, причем данное значение представляет собой координату *y* окна. `YOF` может быть отрицательным числом — в этом случае верхняя кромка окна будет находиться вне экрана;
- `-YOF` — нижняя кромка окна размещается на расстоянии `YOF` пикселов выше от нижнего края экрана. `YOF` может быть отрицательным числом — в этом случае нижняя кромка окна будет находиться вне экрана.

Смещения могут быть представлены в виде пары значений, например:

- `+0+0` — верхний левый угол;
- `-0+0` — правый верхний угол;
- `-0-0` — правый нижний угол;
- `+0-0` — левый нижний угол;

В следующем примере окно эмулятора терминала размещается приблизительно в центре экрана, а окна монитора производительности, почты и часов находятся в правом верхнем углу экрана:

```
xterm -fn 6x10 -geometry 80x24+30+200 &
xclock -geometry 48x48-0+0 &
xload -geometry 48x48-96+0 &
xbif -geometry 48x48-48+0 &
```

Нужно быть внимательным при использовании команды `xterm`. Для некоторых терминалов значение ширины и высоты указывается в символах, в то время как большинство программ использует для этой цели пикселы, из-за чего окна могут получиться очень маленькими.

Как видно из предыдущих рассуждений, отправной точкой для инициализации и запуска системы X Window является исполняемый файл `xinit`. Более того, и в системах, которые не могут запускать X-сервер непосредственно через /etc/init, и в операционных средах с возможностью запуска разных оконных систем он является единственной программой, которая запускает X. Рассмотрим опции программы `xinit` более детально. Командная строка при запуске `xinit` может включать такие параметры:

```
xinit [ [ client ] options ] [ -- [ server ]
[ display ] options ]
```

Если имя `client` не указано, то `xinit` пытается найти в домашнем каталоге пользователя и запустить командный файл .xinitrc.

Если .xinitrc не обнаружен, то `xinit` по умолчанию выполнит команду

```
xterm -geometry +1+1 -n login -display :0
```

Если в списке параметров отсутствует сервер `server`, `xinit` пытается обнаружить в домашнем каталоге пользователя и выполнить командный файл .xserverrc для запуска сервера. В случае отсутствия файла .xserverrc программа `xinit` выполнит команду

```
X :0
```

Запускаемая программа называется `X`, хотя обычным названием является `Xdisplaytype`, где `displaytype` представляет собой тип графического дисплея, который управляется данным сервером.

Далее приводятся примеры использования программы `xinit` с аргументами командной строки.

```
# xinit
```

Команда позволяет запустить X-сервер и выполнить пользовательский командный файл ./xinitrc (если он существует), в противном случае запускается эмулятор терминала `xterm`.

```
# xinit -- /usr/X11R6/bin/Xqdss :1
```

В этом случае запускается специфичный тип сервера на альтернативном дисплее 1.

```
# xinit -geometry =80x65+10+10 -fn 8x13 -j -fg white -bg navy
```

Здесь запускается сервер X, после чего выполняется команда `xterm` с указанными в командной строке параметрами. Командный файл .xinitrc в этом случае игнорируется.

```
# xinit -e widgets -- ./Xsun -l -c
```

В данном примере командой `./Xsun -l -c` запускается сервер, и, кроме того, эмулятору терминала `xterm` передаются параметры `-e widgets`.

```
# xinit /usr/ucb/rsh fasthost cpupig -display ws:1 -- :1 -a 2
-t 5
```

Здесь на дисплее 1 стартует сервер X с аргументами `-a 2 -t 5`, а также запускается команда `cpupig` на удаленной машине `fasthost`.

В следующем примере показано содержимое файла .xinitrc. Здесь запускаются часы и несколько терминалов, после чего выполняется программа оконного менеджера, являющаяся последней в командном файле. Вспомним, что последняя команда этого командного файла должна работать, в отличие от остальных, в обычном режиме.

```
xrdb -load $HOME/.Xresources
xsetroot -solid gray &
xclock -g 50x50-0+0 -bw 0 &
xload -g 50x50-50+0 -bw 0 &
xterm -g 80x24+0+0 &
xterm -g 80x24+0-0 &
twm
```

Если оконный менеджер сконфигурирован правильно, выйти из графического режима можно по команде `exit`, завершающей работу X-сервера.

9.2. Команды X Window

В системе X Window имеется множество команд, позволяющих настроить графический интерфейс пользователя. Как и любые другие команды операционной системы UNIX, они могут быть использованы в командных сценариях.

Необходимо учитывать, что эти команды работают в любых графических оболочках и с любыми оконными менеджерами, поскольку являются встроенными в X Window. Вне зависимости от того, какой оконный менеджер или графическая оболочка применяется, команды X Window можно использовать для настройки и оптимизации как пользовательского интерфейса, так и графической системы в целом.

Описание наиболее часто используемых команд приведено далее:

- `xterm` — эмулятор терминала;
- `xdm` — менеджер дисплея;
- `xconsole` — команда перенаправления вывода консоли;
- `xmh` — интерфейс почтовых служб;
- `bitmap` — редактор ресурсов;
- `edires` — редактор ресурсов;
- `xauth`, `xhost` — команды управления доступом к программам;
- `xset` — вывод пользовательских настроек X Window;
- `xrdb`, `xcmsdb`, `xset`, `xsetrot`, `xstdcmap` и `xmodmap` — программы для настройки интерфейса пользователя.

Кроме того, имеется целый ряд очень полезных утилит пользователя, позволяющих выполнить настройки графического интерфейса и устройств (клавиатуры, мыши, экрана):

- `xmodmap` — настройка таблицы кодировки и клавиатуры, а также переназначение клавиш;
- `xfd`, `xlsfonts` — вывод информации о шрифтах, доступных пользователю;
- `xwd`, `xwud` — утилиты управления экраном;
- `xmag` — монитор производительности;
- `xfs` — редактор шрифтов;
- `xkbcomp` — редактор таблиц раскладки клавиатуры. Может использоваться совместно с другими утилитами для работы с клавиатурой, такими как `xkbcomp`, `xkbprint`, `xkbbell`, `xkbevd`, `xkbvleds` и `xkbwatch`;
- `xkill` — утилита завершения сеанса работы пользователя;
- `rstart`, `xon` — программы удаленного доступа и выполнения программ;
- `twm` — оконный менеджер;
- `clocks`, `xclock` и `oclock` — утилиты настройки даты/времени;

- `xfd` — выводит информацию об используемых шрифтах;
- `xlsfonts`, `xwininfo`, `xlsclients`, `xdpyinfo`, `xlsatoms`, `xprop` — утилиты для вывода информации о шрифтах, окнах и дисплеях;
- `xwd`, `xwud` и `xmag` — утилиты настройки изображения (screen image manipulation utilities);
- `x11perf` — утилита для измерения производительности;
- `bdftopcf` — компилятор шрифтов;
- `Xserver`, `rgb`, `mkfontdir` — сервер дисплея и вспомогательные утилиты;
- `rstart` и `xon` — утилиты для запуска удаленных программ;
- `xclipboard` — менеджер буфера обмена (clipboard manager);
- `xkbcomp`, `xkbprint`, `xkbel`, `xkbevd`, `xkbvleds` и `xkbwatch` — компилятор раскладки клавиатуры (keyboard description compiler) и вспомогательные утилиты.

Существует и масса других утилит, оконных менеджеров и иных вспомогательных программ, которые можно загрузить из Интернета как свободно распространяемое программное обеспечение.

Команды X Window можно выполнять как из командной строки, так и в командных сценариях. Вот несколько примеров командных строк с утилитами X Window:

```
$ xrdb $HOME/.Xresources
$ xmodmap -e "keysym BackSpace = Delete"
$ mkfontdir /usr/local/lib/X11/otherfonts
$ xset fp+ /usr/local/lib/X11/otherfonts
$ xmodmap $HOME/.keymap.km
$ xsetroot -solid 'rgbi:.8/.8/.8'
$ xset b 100 400 c 50 s 1800 r on
$ xset q
$ twm
$ xmag
$ xclock -geometry 48x48-0+0 -bg blue -fg white
```

```
$  xeyes -geometry 48x48-48+0
$  xbiff -update 20
$  xlsfonts '*helvetica*'
$  xwininfo -root
$  xdpyinfo -display myworkstation:0
$  xhost -myworkstation
$  xrefresh
$  xwd | xwud
$  bitmap companylogo.bm 32x32
$  xcalc -bg blue -fg magenta
$  xterm -geometry 80x66-0-0 -name myxterm $*
$  xon filesysmachine xload
```

Более детально ознакомиться со всеми возможностями команд X Window читатели смогут, обратившись к соответствующим man-страницам и другой документации. Среди множества утилит для настройки X Window выделим несколько, очень часто используемых в командных файлах.

Команда `xset` — предназначена для настройки параметров дисплея. Вот смысл некоторых часто используемых опций:

- `-display` *дисплей* — определяет используемый X-сервер;
- `b` — устанавливает параметры для динамика (звук, тон, частоту);
- `-dpms` — запрещает использование DPMS (Energy Star);
- `+dpms` — разрешает использование DPMS;
- `dpms` *флаги* — устанавливает параметры для режима DPMS. Опция может принимать до трех числовых параметров. Например, параметр `force` требует немедленного переключения в требуемое или установленное состояние. Для режима возможно одно из состояний: `standby`, `suspend`, `off` или `on`;
- `fp=`*путь*`,...` — определяет путь к файлам шрифтов. Этот параметр интерпретируется только сервером и представляет собой имя каталога;

- `fp default` — восстанавливает умолчания для пути поиска файлов шрифтов;
- `fp rehash` — требует от сервера повторной инициализации базы данных шрифтов. Опция используется при добавлении нового шрифта;
- `-fp` — удаляет указанные путевые имена к файлам шрифтов;
- `fp+` — добавляет путевые имена к файлам шрифтов;
- `led` — управляет светодиодами (LEDs) на клавиатуре. С помощью этой опции можно включать или отключать индикацию клавиатуры;
- `m` — устанавливает параметры мыши. Используется для настройки "ускорения" и "чувствительности" мыши. Обычно установкой этих параметров пытаются добиться более точного позиционирования указателя мыши;
- `r` — устанавливает опции автоповтора символов клавиатуры. Установка в `-r` или `r off` запрещает автоповтор, в то время как установка `r` или `r on` — разрешает. Если после опции `-r` или `r` следует число в диапазоне от 0 до 255, то это запрещает или разрешает автоповтор для клавиши с соответствующим кодом. Коды клавиш со значением меньше 8 обычно не используются. Например, команда

  ```
  xset -r 10
  ```

  запрещает автоповтор для клавиши <1> на обычной IBM-клавиатуре;
- `s` — устанавливает параметры для хранителя экрана (screen saver). Опция может принимать до двух числовых параметров. Этими параметрами могут быть: `blank` или `noblank`, `expose` или `noexpose`, `on` или `off`, `activate`, `reset` или `default`. Если параметры не заданы, по умолчанию принимается `default`;
- `q` — выводит на экран информацию о текущих настройках.

Важно отметить, что все настройки этой команды устанавливаются в значения по умолчанию после завершения сеанса работы

в X Window. Кроме того, не все реализации X Window поддерживают те или иные опции.

Рассмотрим несколько примеров применения команды `xset`. Вывод информации о текущих настройках сеанса X Window можно выполнить при помощи такой команды:

```
# xset q
Keyboard Control:
  auto repeat:  on    key click percent:  0    LED mask:
00000000
  auto repeat delay:  500    repeat rate:  30
  auto repeating keys:  00ffffffdffffbbf
                        fadfffffffdfe5ff
                        fffffffffffffff
                        fffffffffffffff
  bell percent:  50    bell pitch:  400    bell duration:
100
Pointer Control:
  acceleration:  2/1    threshold:  4
Screen Saver:
  prefer blanking:  yes    allow exposures:  yes
  timeout:  600    cycle:  600
Colors:
  default colormap:  0x20    BlackPixel:  0    WhitePixel:
16777215
Font Path:
  /root/.gnome2/share/cursor-
fonts,unix/:7100,/root/.gnome2/share/fonts
Bug Mode: compatibility mode is disabled
DPMS (Energy Star):
  Standby: 1200    Suspend: 1800    Off: 2400
  DPMS is Enabled
  Monitor is On
```

```
Font cache:
  hi-mark (KB): 5120  low-mark (KB): 3840  balance (%): 70
File paths:
  Config file:  /etc/X11/XF86Config
  Modules path: /usr/X11R6/lib/modules
  Log file:     /var/log/XFree86.0.log
```

Следующая команда устанавливает время активизации хранителя экрана равным 10 минутам:

```
# xset s 600
```

К одной из очень полезных команд X Window относится команда `xmodmap` — с ее помощью можно выполнить модификацию раскладки клавиатуры и переназначение клавиш. Утилита часто применяется для персонализации настроек интерфейса и помещается обычно в стартовом командном файле пользователя.

Синтаксис команды таков:

```
xmodmap [-опции ...] [имя_файла]
```

Вот смысл некоторых опций:

- `-display` *дисплей* — определяет хост и дисплей, для которых выполняются установки;
- `help` — выводит краткое описание команды и ее опций;
- `-e` *выражение* — определяет выражение, которое должно быть выполнено, при этом само выражение может задаваться в командной строке;
- `-pk` — указывает на необходимость вывода таблицы текущей раскладки клавиатуры на устройство стандартного вывода.

В команде `xmodmap` могут использоваться выражения для переназначения функций отдельных клавиш или присваивания функций одних клавиш другим. Например, такое выражение как

```
keycode NUMBER = KEYSYMNAME ...
```

определяет перечень символических кодов клавиш (`keysyms`), которые могут быть присвоены указанному коду клавиши (`keycode`).

Команда `xmodmap`, заданная без параметров, выводит на дисплей таблицу раскладки управляющих клавиш для данного сеанса X Window:

```
# xmodmap
xmodmap:  up to 2 keys per modifier, (keycodes in parentheses):

shift       Shift_L (0x32),  Shift_R (0x3e)
lock        Caps_Lock (0x42)
control     Control_L (0x25),  Control_R (0x6d)
mod1        Alt_L (0x40),  Alt_R (0x71)
mod2        Num_Lock (0x4d)
mod3
mod4        Super_L (0x73),  Super_R (0x74)
mod5
```

Следующая команда

```
xmodmap -pk
```

выводит таблицу с раскладкой клавиатуры с символическими именами, присвоенными клавишам. Вот часть такой таблицы:

```
# xmodmap -pk
There are 4 KeySyms per KeyCode; KeyCodes range from 8 to 255.

    KeyCode     Keysym (Keysym) ...
    Value       Value   (Name)  ...

      8
      9         0xff1b (Escape)
     10         0x0031 (1)      0x0021 (exclam)
     11         0x0032 (2)      0x0040 (at)
     12         0x0033 (3)      0x0023 (numbersign)
     13         0x0034 (4)      0x0024 (dollar)
```

```
     14          0x0035 (5)       0x0025 (percent)
     15          0x0036 (6)       0x005e (asciicircum)
     16          0x0037 (7)       0x0026 (ampersand)
     17          0x0038 (8)       0x002a (asterisk)
     18          0x0039 (9)       0x0028 (parenleft)
     19          0x0030 (0)       0x0029 (parenright)
     20          0x002d (minus)   0x005f (underscore)
     21          0x003d (equal)   0x002b (plus)
     22          0xff08 (BackSpace)      0xfed5 (Termi-
nate_Server)
     23          0xff09 (Tab)     0xfe20 (ISO_Left_Tab)
     24          0x0071 (q)       0x0051 (Q)
     25          0x0077 (w)       0x0057 (W)
```

Команду `xmodmap` можно применить в командных файлах при настройке рабочего окружения пользователя или при разработке служебных утилит. Например, если требуется определить ASCII-код определенной клавиши, можно выполнить командную строку

```
# xmodmap -pk|grep '(a)'|cut -f2|cut -c1-6
0x0061
```

В этом примере на экран выводится код клавиши <A>.

Часто бывает очень удобно функции одной клавиши передать другой. Команда `xmodmap` позволяет это сделать довольно просто:

```
# xmodmap -e "keysym BackSpace = Delete"
```

После выполнения этих команд клавиша <Backspace> будет работать как <Delete>, т. е. удалять символы, расположенные за курсором.

Для того чтобы эти изменения приняли постоянный характер, с помощью менеджера ресурсов `xrdb` нужно откорректировать базы данных ресурсов X Window, что и выполняется второй командой:

```
# echo "XTerm*ttyModes:  erase ^?" | xrdb -merge
```

Выражение, используемое в команде `xmodmap`, может быть записано в файл и в дальнейшем использовано в конвейере команд.

Сохраним, например, выражение

```
keysym BackSpace = Delete
```

в файле xmodmap.ch:

```
echo keysym BackSpace = Delete > xmodmap.ch
```

Тогда для переприсвоения функций клавиш можно использовать командную строку:

```
cat xmodmap.ch | xmodmap -
```

Система X Window устанавливается с обширной базой шрифтов. Команда `xlsfonts` дает пользователю возможность получать информацию о шрифтах, изменять настройки и добавлять в систему новые шрифты. Чтобы отобразить, например, все шрифты размером 6×13, установленные в системе, необходимо выполнить команду:

```
# xlsfonts|grep 6x13
6x13
6x13
6x13bold
6x13bold
heb6x13
heb6x13
```

Можно установить для приложения те шрифты, которые более удобны. Например, при создании X-терминала можно установить для него шрифт 12×24:

```
# xterm -g 80x30+250+280 -fn `xlsfonts|grep 12x24 -m 1`
```

Приведенная здесь командная строка работает так, как описано далее. Терминальная сессия открывается с помощью команды `xterm`. В качестве параметров этой команды используются геометрические характеристики терминального окна (параметр `-g`). Для задания шрифта применяется опция `-fn`, при этом коман-

дой `xlsfonts` выводятся все шрифты 12×24. Команда `grep` выбирает первую строку с указанным шрифтом и передает ее в `xterm`.

9.3. Оконные менеджеры и графические оболочки

Система X Window, как известно, сама по себе не предоставляет клиенту готовый графический интерфейс для работы. При запуске X-сервер создает корневое окно, в котором могут запускаться простейшие программы-клиенты (`xedit`, `xterm` и т. д.). X не создает для пользователя графический интерфейс с меню, окнами, кнопками управления и т. д., предоставляя пользователю лишь инструменты, а уж он сам принимает решение, как их использовать.

Такими инструментами являются *оконные менеджеры* и *графические оболочки*. Оконные менеджеры и графические оболочки обеспечивают различный уровень функциональности для работы с клиентскими приложениями и имеют много общего, как в плане программной архитектуры, так и в способах взаимодействия с базовой системой X Window.

9.3.1. Оконные менеджеры

Взаимосвязь X-сервера, X-клиентов и оконных менеджеров демонстрирует рис. 9.4.

Самый первый уровень, наиболее близко находящийся к операционной системе, представлен X-сервером, принципы функционирования которого мы достаточно подробно проанализировали. На практике используется несколько наиболее известных дистрибутивов X-серверов: XFree86, Metro-X и Accelerated-X. X-сервер обычно устанавливается во время инсталляции операционной системы, хотя можно проинсталлировать его и из отдельного дистрибутива.

На втором уровне располагается оконный менеджер, который отвечает за управление окнами. Он представляет собой специальную

клиентскую программу, которая указывает серверу, каким образом располагать окна, и обеспечивает пользователю способ перемещения окон. Границы окон, заголовок, размеры и т. д. — все эти параметры регулируются оконными менеджерами.

Рис. 9.4. Модель взаимодействия оконных менеджеров с системой X Window

Рисунок 9.4 демонстрирует очень важный факт: все графические приложения (X-клиенты) могут взаимодействовать с X-сервером как напрямую, так и посредством оконных менеджеров. Оконные менеджеры позволяют управлять характеристиками окон: размером, положением, с их помощью можно перемещать окна и обновлять их содержимое, а также закрывать окна приложений, что, в большинстве случаев, равнозначно завершению программ.

Без оконного менеджера окно нельзя переместить в определенную позицию, изменить его размер или, например, свернуть его. Характеристики окна приложения задаются при его создании, например, в файле .xinitrc, и изменить их можно только про-

граммным путем, используя функции X-сервера. Например, при инициализации X-сервера (см. рис. 9.3) мы задавали команду

```
$ xinit -g 60x20+150+150 -fn 10x20 -bg black -fg white
-display :0
```

с помощью которой создавалось окно терминала с указанными характеристиками и фиксированным положением на корневом окне. Далее было создано окно графического редактора `emacs`, также с определенными фиксированными параметрами:

```
$ emacs -g 50x30+250+250
```

Оконные менеджеры значительно "облегчают жизнь" пользователю, поэтому их, как правило, вызывают на исполнение в последней командной строке файла .xinirc.

Посмотрим, как будет выглядеть интерфейс пользователя, если для управления X-клиентами использовать оконный менеджер. В данном примере применяется оконный менеджер `mwm` (Motif Window Manager), который будет работать в эмуляторе Cygwin.

Создадим (если еще не создан) или модифицируем командный файл .xinitrc, записав в него такие команды:

```
xsetroot -solid magenta&

xterm -g 60x30+250+250 -fg white -bg black&

xclock -digital&

exec mwm
```

Здесь первая команда устанавливает пурпурный цвет фона для корневого окна, вторая создает окно терминала, третья запускает цифровые часы, и, наконец, последняя команда вызывает на исполнение оконный менеджер `mwm`.

Запустим X-сервер:

```
$ xinit
```

Если параметры дисплея установлены корректно, то получим примерно такое графическое изображение, которое показано на рис. 9.5.

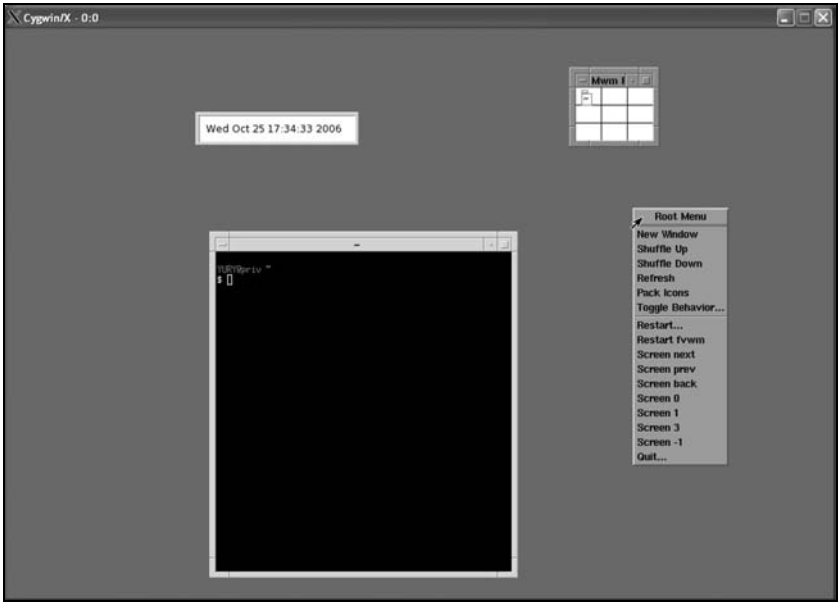


Рис. 9.5. Система X Window
с работающим оконным менеджером `mwm`

Теперь можно управлять поведением графических приложений. Щелкнув правой кнопкой мыши на свободном поле, можно вызвать общее меню менеджера `mwm` (**Root Menu** на рисунке). Если то же самое сделать на заголовке окна какого-либо приложения, то выпадет меню, позволяющее переместить, свернуть, закрыть окно или изменить его размер.

Оконный менеджер не может корректно управлять окнами, не имея о них информации. Окна обладают различными свойствами, управление которыми должен обеспечивать именно менеджер окон. Например, во многих случаях необходимо отображать заголовки окон, в других желательно, чтобы окно имело фиксированный размер. Окно может быть свернуто в пиктограмму (значок) — в этом случае оконный менеджер должен знать, какую пиктограмму использовать и как она называется.

Менеджер получает от клиентов информацию о параметрах окна одним из способов:

- при создании X-окна ему могут быть переданы "подсказки" (hints) о начальных координатах окна, его геометрии, минимальных и максимальных размерах и т. д.;
- при использовании встроенного в X способа взаимодействия между программами — механизм "свойств".

Еще одной важной функцией, выполнение которой берет на себя оконный менеджер, является "политика фокусирования" мыши. Каждая оконная система должна иметь некоторый способ активизации окна при его выборе через нажатие клавиш, а также визуальную индикацию активности окна. Наиболее широко используется метод фокусировки (click-to-focus). Этот метод применяется в Microsoft Windows, и суть его в том, что окно становится активным после щелчка на нем мышью.

X не поддерживает какой-либо конкретный метод фокусирования — этим занимается оконный менеджер, который управляет передачей фокуса окну в определенные моменты времени. Различные оконные менеджеры поддерживают разные методы фокусирования. Все они поддерживают метод щелчка для фокусирования, и большинство из них — некоторые другие методы.

Самыми популярными методами фокусирования являются:

- focus-follows-mouse (фокус следует за мышью). Фокусом владеет окно, находящееся под указателем мыши. Оно не обязательно должно быть расположено поверх всех остальных окон. Фокус меняется при перемещении указателя на другое окно, при этом щелчок не требуется;
- sloppy-focus (нечеткий фокус). При использовании данного метода фокус меняется только тогда, когда курсор перемещается на другое окно, но не в момент ухода с текущего окна;
- click-to-focus (щелчок для выбора фокуса). Активное окно выбирается щелчком мыши, после чего окно может быть "поднято" и появиться поверх остальных окон. Все нажатия клавиш теперь будут направляться этому окну, даже если курсор мыши переместится к другому.

Многие оконные менеджеры поддерживают и другие методы, а также различные модификации вышеперечисленных. Современные оконные менеджеры предоставляют пользователю различные графические интерфейсы: одни поддерживают "виртуальные рабочие столы", другие позволяют изменять установки по умолчанию клавиш управления и навигации, третьи поддерживают несколько визуальных форм, позволяя тем самым изменять внешний вид.

9.3.2. Графические оболочки

В большинстве современных операционных систем UNIX для управления графическими приложениями используются еще более сложные и совершенные программы, которые обычно называют *графическими оболочками*. Графические оболочки, помимо того, что выполняют функции менеджеров окон, обеспечивают также и многие другие функции, например, синхронизацию запуска и выполнения нескольких приложений, высокоуровневый интерфейс настройки параметров графической системы и т. д. Графические оболочки в большинстве случаев интегрируют в себя функции оконных менеджеров, хотя могут использовать и автономные программы оконных менеджеров.

В таких случаях оконный менеджер и графическая оболочка работают вместе, причем каждая графическая оболочка рассчитана на совместную работу с вполне определенными оконными менеджерами, хотя это не обязательно. Например, ранние версии графической оболочки GNOME работали с оконным менеджером Enlightenment, хотя более поздние версии этой же оболочки рассчитаны на работу с оконным менеджером Sawfish. Наиболее популярными оконными менеджерами являются WindowMaker, BlackBox, Ice WM, Sawfish, Enlightenment, AfterStep, Fvwm2 и Fvwm1.

В большинстве современных операционных систем предусмотрена возможность работы с различными графическими оболочками. Например, такая распространенная операционная система

как Linux допускает работу с популярными графическими оболочками GNOME и KDE. Более того, поскольку все операционные системы UNIX являются многопользовательскими, то на одном компьютере может работать несколько графических оболочек, причем разные пользователи, работающие с одной и той же программой, могут использовать разные оболочки.

Наиболее распространенными графическими оболочками в настоящее время являются KDE и GNOME. Они имеют интегрированные оконные менеджеры, причем каждый из них, как правило, обладает собственным механизмом настройки, а некоторые из них используют созданный вручную файл конфигурации.

В GNOME имеется панель для запуска приложений и отображения их состояния, а также рабочий стол с набором стандартных инструментов, на котором можно разместить данные и приложения. Кроме того, в графической оболочке предусмотрены механизмы взаимодействия приложений и пользователей, значительно облегчающие им работу.

Особенностью оболочки KDE является ее простота использования. Вот некоторые из большого числа возможностей, которые предоставляются пользователю данной оболочкой:

- современный рабочий стол;
- удобная справочная система;
- единый подход к управлению всеми приложениями оболочки;
- стандартизированные элементы управления (меню и панели инструментов), раскладки клавиатуры, цветовые схемы и т. д.;
- локализация более чем для 40 языков;
- диалоговый режим конфигурирования рабочего стола;
- возможность выбора большого количества полезных приложений для KDE.

Для KDE разработан пакет офисных приложений, включая программы для работы с электронными таблицами и создания презентаций, органайзер, клиент телеконференций и другие программы. В комплект поставки KDE входит также Web-браузер

Konqueror, который является серьезной альтернативой другим обозревателям для UNIX-систем.

Вот как выглядят интерфейсы графических оболочек KDE (рис. 9.6) и GNOME (рис. 9.7).

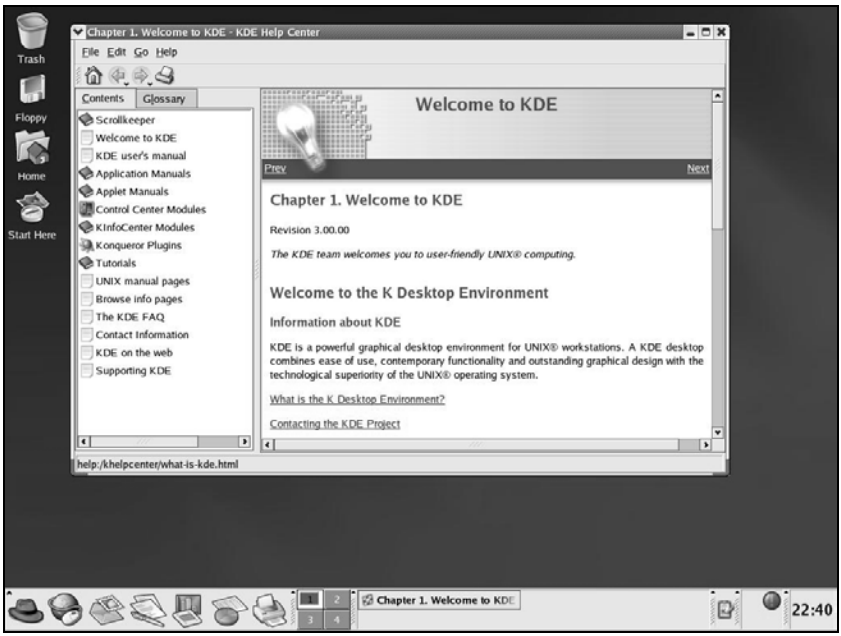


Рис. 9.6. Интерфейс графической оболочки KDE

Рассмотрим способы инсталляции и настройки графических оболочек в операционных системах UNIX на примере FreeBSD.

Начнем с GNOME. Наиболее легкий вариант установки этой оболочки — использование меню **Desktop Configuration**, которое появляется в процессе установки FreeBSD. Другим способом установки является инсталляция оболочки из пакета или коллекции портов, а также из исходных текстов (в этом случае используется дерево портов).

Последовательность команд для установки из исходных текстов такова:

```
# cd /usr/ports/x11/gnome2
# make install clean
```

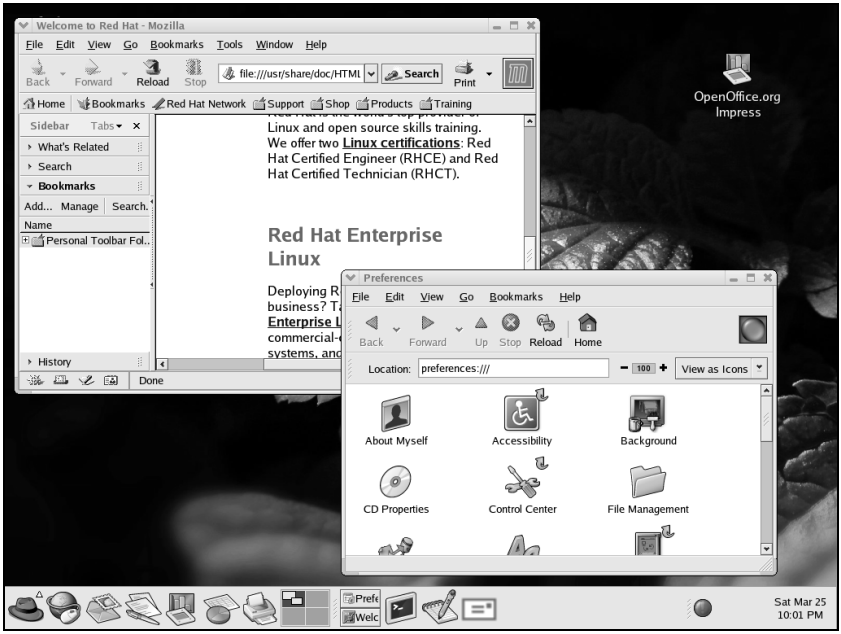


Рис. 9.7. Интерфейс графической оболочки GNOME

После установки GNOME необходимо указать X-серверу, что следует запускать оболочку GNOME вместо стандартного оконного менеджера. Для этого в файле .xinitrc нужно заменить строку (обычно это последняя строка), в которой указан оконный менеджер, той, что вызывает /usr/X11R6/bin/gnome-session. Вот как может выглядеть откорректированный файл .xinitrc:

```
#/etc/X11/xinit/.xinitrc
#!/bin/sh
```

```
xmodmap -e "keycode 22=BackSpace"
xsetroot -solid LightSlateGrey
xset s 600
xterm -g 80x20+150+8 &
xterm -g 80x20+150+325 &
xload -g +4+0 &
xclock -g +815+0 -digital &

# Запуск графической оболочки GNOME
# exec fvwm2    Эта строка должна быть закомментирована или
удалена
/usr/X11R6/bin/gnome-session
```

Саму строку можно добавить в файл конфигурации с помощью команды:

```
# echo "/usr/X11R6/bin/gnome-session" > ~/.xinitrc
```

Если теперь выполнить команду `startx`, то будет запущена графическая оболочка GNOME. Здесь, однако, есть один нюанс. Если используется менеджер дисплея `xdm`, то GNOME не запустится. В этом случае нужно создать исполняемый файл .xsession, в котором будет присутствовать строка /usr/X11R6/bin/gnome-session.

Для этого следует отредактировать файл, заменив существующую команду запуска оконного менеджера на /usr/X11R6/bin/gnome-session:

```
# echo "#!/bin/sh" > ~/.xsession
# echo "/usr/X11R6/bin/gnome-session" >> ~/.xsession
# chmod +x ~/.xsession
```

Можно использовать еще один вариант — так настроить менеджер дисплея, чтобы он позволял выбирать оконный менеджер во время входа в систему.

Оболочку KDE, как и GNOME, проще всего установить через меню **Desktop Configuration** в процессе инсталляции FreeBSD.

Хочу напомнить, что программное обеспечение легко установить из пакета или из коллекции портов. Для сборки KDE из исходных текстов можно воспользоваться деревом портов:

```
# cd /usr/ports/x11/kde3
# make install clean
```

Для запуска KDE по умолчанию нужно отредактировать файл .xinitrc так:

```
# echo "exec startkde" > ~/.xinitrc
```

Теперь при вызове X Window по команде `startx` в качестве оболочки будет использоваться KDE. При работе с менеджером дисплеев типа XDM настройка несколько отличается — отредактировать нужно файл .xsession. Дополнительную информацию по работе с оболочкой KDE можно получить из справочной системы. Неплохим справочным материалом по KDE является online-документация.

Заключение

Система UNIX, разработанная еще в 70-е годы прошлого столетия, в настоящее время переживает новый этап развития. Архитектура системы является оптимальной, обеспечивая устойчивую и надежную работу, поэтому многие решения, используемые в UNIX, в той или иной степени позаимствованы другими операционными системами, например, той же Windows.

Основной проблемой, препятствующей широкому распространению UNIX до начала 90-х годов, было отсутствие ее реализации для массового пользователя, работающего на персональном компьютере. Появление в начале 90-х операционной системы Linux повлекло действительно революционные изменения — системы UNIX приобрели массового пользователя, что обусловило быстрое расширение сфер их применения. Массовое использование системы с открытым кодом, какой является Linux, обусловило лавинообразный рост разработок для этой операционной системы, начиная с середины 90-х годов XX века. В процессе своего развития операционная система Linux приобрела массу дополнительных возможностей, включая возможность работы с графическим интерфейсом пользователя, обработки мультимедийных данных и т. д. Развитие этой операционной системы, в свою очередь, стимулировало развитие и других реализаций UNIX, как коммерческих, так и систем с открытым кодом. В этом контексте следует упомянуть о системе FreeBSD, ставшей довольно популярной и составляющей серьезную конкуренцию Linux.

В настоящее время сфера применения операционных систем UNIX постоянно расширяется. Кроме областей, в которых операционные системы UNIX используются чаще остальных, например, серверы Интернета, коммуникационные системы, военные и специальные разработки, появились и новые сферы применения. Благодаря развитию офисных приложений для UNIX, таких, например, как StarOffice, пакеты для обработки

графики и видео и т. д., рабочие станции под управлением этой операционной системы стали понемногу вытеснять Windows-совместимые системы с установленным программным обеспечением Microsoft Office.

Что же касается перспектив развития операционных систем UNIX, то сделать какие-либо прогнозы на отдаленное будущее трудно. Тем не менее, можно смело предположить, что архитектура UNIX-систем, в отличие от Windows, далеко не исчерпала своих возможностей и имеет значительный ресурс для развития. Также очевидно, что основной проблемой быстро растущего рынка операционных систем UNIX является стандартизация.
В краткосрочной перспективе операционные системы UNIX будут развиваться в сторону поддержки новых технологий обмена данными, оставаясь наиболее популярной платформой для серверов Интернета и систем телекоммуникаций. Не следует исключать и постепенное расширение парка офисных систем на базе свободно распространяемых операционных систем Linux и FreeBSD, учитывая тенденции в области развития пакетов программ для обработки текстовых и графических документов.

Возможность создания дистрибутивов операционных систем UNIX, занимающих небольшой объем дискового пространства, делает перспективным применение UNIX в мобильных устройствах и в сфере автоматизации производственных процессов, учитывая значительный прогресс в области твердотельной электроники, позволяющей создавать запоминающие устройства небольших размеров и большой емкости.

Предметный указатель

A

Access permission 69
Account 93

D

Daemon 18
Device hint 197
DNS 281

F

Free Software Foundation (FSF) 13

G

GNOME 455

H

Handle 54

J

Java 347
  апплет 383
  ввод/вывод 372
  класс 356
  обработка ошибок 362
  оператор:
    "точка" 359
    new 358
  пакет 371
  поток 372
  синтаксис языка 354
  строки 364

K

KDE 457
Kernel threads 19, 21

L

Lightweight process 22
Loopback interface 47

M

Mail Delivery Agent 300
Mail Transfer Agent 299
Mail User Agent 299

N

Non-preemptive multitasking 22

P

PartitionMagic 140
Perl 384
  запуск программы 387
  каталоги 416
  логические структуры 403
  операции 390, 399
  регулярное выражение 407
  сетевое программирование 417
  строки 401
  типы данных 388
  файлы 412
  хэш 398
POSIX 9
Process ID (PID) 26

R

Root window 427
Router 287
Routing tables 289
Run-time library 19

S

Security profile 173
Shell 19, 40
Socket 277
System calls 19

T

Threads 31

U

Uid aging 115

V

Volume Table Of Contents (VTOC) 139

W

Web-сайт 308
Window manager 422

X

X Window 421
  запуск 432
  команды 438
X-сервер 175

А

Агент:
доставки 300
пользовательский 299
транспортный 299
Адрес:
IP 282
URL 310
аппаратный 267
сети 284
Адресация, групповая 285

Б

Библиотека времени выполнения 19
Браузер 309

В

Виртуальная консоль 94

Г

Геометрия окна 428
Графическая оболочка 448, 453

Д

Демон 18
Дескриптор файла 54
Дефектная область диска 139
Диск, разметка 140
Дорожка 139
Драйвер 47

Ж

Жесткий диск 138

З

Загрузочный блок 142

И

Идентификатор процесса 26
Индексный дескриптор 58, 143
Интерпретатор команд 218
синтаксис 221
Интерфейс обратной связи 47

К

Канал:
именованный 48
неименованный 33
программный 232
Каталог:
корневой 49
поиск 87
создание 86
удаление 85
Коллизия 262
Команда:
arp 293
boot 201
chgrp 76
chmod 71
chown 76
chpass 121
cp 80
env 102

(окончание рубрики см. на с. 464)

Команда *(окончание)*:
export 246
expr 244
find 87
halt 200
kenv 198
make 344
mkdir 86
mkfs 149
mount 150
mv 83
netstat 295, 315
passwd 118
ping 319
ps 95
reboot 200
rm 86
rmdir 85
rmuser 122
route 293, 294
shutdown 200
traceroute 322
umount 150
unmask 74
userdel 115
usermod 116
Командная оболочка 40
Командный интерпретатор 19
Конвейер команд 232
Контроллер 139

M

Маршрутизатор 287
Маршрутизация 287
динамическая 293, 298
статическая 294
Маска сети 282
Метка диска 139
Многозадачность,
невытесняющая 22
Модель OSI 267

O

Объект файловой системы 17
Окно, корневое 427
Оконный менеджер 422, 448
Оператор:
case 254
условия if 250
цикла:
for 248
while 253
Операционная система:
FreeBSD 11, 161, 191
Linux 12, 183, 209
Solaris 11, 180, 201
запуск и останов 190
многопользовательская 17
установка 131

П

Пакет 260
сетевой 267
Переназначение:
ввода 227
вывода 225
Подкачка 204
Подсказка устройства 197
Позиционный параметр 242
Пользователь,
привилегированный 107
Поток 19, 21, 31
Права доступа к файлам 69
Программа 24
fdisk 145
fsck 143, 160

ifconfig 311
mail 301
sendmail 306
sync 143
Пространство ядра 26
Протокол:
DHCP 168, 286
FTP 280
HTTP 308
RIP 298
SMTP 280
SNMP 281
TCP 274
TCP/IP 272
telnet 278
UDP 275
маршрутизации 294
Профиль безопасности 173
Процесс 17, 24
дочерний 27
порожденный 27
прикладной 41
режим выполнения 26
родительский 27
сервер 18
системный 18, 39
элементарный 22
Путевое имя 49
абсолютное 49
относительное 49

P

Раздел диска 140
главный 140
логический 140
расширенный 140
Регулярное выражение 407

C

Сектор 139
Сетевой интерфейс 311
Сетевой протокол передачи данных 266
Сеть 259
глобальная 260
локальная 260
Системный вызов 19, 53
chmod() 54
chown() 54
close() 54, 57
fork() 53
fstat() 54
link() 54
lseek() 54
open() 53, 54
pipe() 33
read() 53, 56
socket() 48
stat() 54, 65
umask() 54
unlink() 54
utime() 54
write() 53, 57
Служба разрешения имен 281
Сокет 39, 47, 277
Ссылка:
жесткая 48, 52
символическая 48, 52
Суперблок 142

T

Таблица:
дескрипторов 27
индексных дескрипторов 61
маршрутизации 289

Терминальная линия 93
Терминальное соединение 93
Топология сети 261
звезда 263
кольцо 264
общая шина 262

У

Установка операционной системы 131
Устаревание идентификатора 115
Учетная запись 93
изменение параметров 116
создание 113
удаление 115

Ф

Файл:
атрибуты 63
бинарный 46
командный 217, 240
копирование 80
паролей passwd 109
перемещение 83
поиск 87
создание 61
удаление 85
устройства 44, 46
Файловая система 43, 48
UFS 141
демонтирование 150
диагностика 160
монтирование 150
сетевая 172
создание 138
структура каталогов 50
Фильтр 231

Х

Хост 260

Ц

Цилиндр 139

Ш

Шлюз 287

Э

Электронная почта 299

Я

Ядро 16, 18, 20, 201
многопоточность 21
модульное 21
монолитное 21