Алексей Федорчук

bhv®

Доступный UNIX

Linux, FreeBSD, DragonFlyBSD, NetBSD, OpenBSD

Доступно для получения, изучения и использования!

- Идеология, история, практика
- Принципы настройки и использования
- Файлы, процессы и пользователи, терминалы, режимы и интерфейсы
- Универсальные приемы работы для решения повседневных задач

Наиболее полное руководство

Linux center
www.linuxcenter.ru

В ПОДЛИННИКЕ®

Алексей Федорчук

Доступный UNIX

Linux, FreeBSD, DragonFlyBSD, NetBSD, OpenBSD

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.06
ББК 32.973.26
Ф33

Федорчук А. В.

Ф33 Доступный UNIX: Linux, FreeBSD, DragonFlyBSD, NetBSD, OpenBSD. — СПб.: БХВ-Петербург, 2006. — 672 с.: ил.

ISBN 5-94157-876-8

Книга посвящена описанию свободных UNIX-подобных операционных систем, представителями которых являются Linux, FreeBSD и другие члены BSD-семейства, а также их использованию в качестве универсальной платформы общего (в том числе и домашнего) назначения. Материал книги не привязан к какой-либо конкретной ОС или дистрибутиву Linux, а содержит описание общих принципов установки, настройки и эксплуатации любого из представителей этого семейства. С позиции конечного пользователя рассмотрены вопросы идеологии и истории UNIX-подобных систем, изложены основные принципы, на которых они базируются, такие как понятия файлов, процессов, учетных записей пользователей, режимов и интерфейсов, даны приемы решения повседневных пользовательских задач.

Книга поможет начинающим пользователям приступить к работе в одной из описанных ОС, а опытным — освоить универсальные приемы работы в любой из них.

Для широкого круга пользователей

УДК 681.3.06
ББК 32.973.26

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Елена Кашлакова*
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 30.03.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 54,18.
Тираж 3000 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 5-94157-876-8

© Федорчук А. В., 2006
© Оформление, издательство "БХВ-Петербург", 2006

Оглавление

Преамбула **1**

Необходимое вступление 1
Зачем эта книга 1
Для кого эта книга 5
О чем эта книга 5
Почему она такая 8
Как она делалась 10
О терминологии 11
References 14
Реверансы 15

Глава 1. Открытость, свобода и халява **17**

Постановка вопроса 17
Степени свободы 18
Грани открытости 23
Кое-что о лицензиях 25
Истоки Free Software 28
Кто оплачивает банкет 30
Можно ли заработать на Open Source 33
Как же заработать на Open Source 36
О продолжении банкета 38

Глава 2. UNIX, Linux и BSD **41**

Что такое ОС 41
Что необходимо для ОС 43
Кое-что о стандартах 49
Кое-что о GNU, или не GNU ли Linux? 51
Немного о дистрибутивах Linux 53
О BSD сотоварищи 56

Глава 3. Вопросы истории POSIX'ивизма **61**

Bell-прелюдия 61
Берклиада UNIX-кода 63
Пусть расцветают все цветы 67
Упорядочивание стилей работы 69
Увертюра Линуса 70
Свободная берклиада: продолжение истории 73

Глава 4. Почему Linux не Windows ... 77

Linux — это не Windows ... 77
Почему компьютер — не видеомагнитофон ... 80
Рецепты против принципов ... 83

Глава 5. Как научиться плавать: установка системы ... 89

Подготовка к заплыву ... 89
Загрузка и запуск ... 91
Подготовка диска ... 95
Установка ... 98
Обеспечение загрузки ... 100
Обеспечение работы в графическом режиме ... 101
Особенности установки BSD-систем ... 103
Проблема выбора ... 105

Глава 6. Все для блага человека: пользовательские акаунты ... 109

Очередная преамбула ... 109
О себе любимом ... 110
Атрибуты учетной записи ... 111
Доступ к атрибутам ... 114
Создание и изменение акаунтов ... 121

Интермедия: средства управления акаунтами ... 123

Глава 7. Процесс пошел ... 133

Понятие процесса ... 133
Разновидности процессов ... 134
Атрибуты процесса ... 135
Жизнь и смерть процесса ... 138
Управление процессами ... 140

Глава 8. Файл как он есть ... 143

Что такое файл ... 143
Классификация файлов ... 146
Каталоги ... 147
Символические ссылки ... 149
Файлы устройств ... 151
Каналы и сокеты ... 153
Обычные файлы ... 153
Еще раз об именах ... 154
Право на файл ... 156
О времени и о файле ... 160

Интермедия: управление файлами 161

Создание 162
Атрибуция 167
Навигация по файловой системе 173
Получение информации о файлах 175
Манипулирование файлами 179
Архивация и компрессия 183
Резервное копирование 189
Венец универсализма: утилита *find* 196

Глава 9. Физика файловых систем 203

Дисковые накопители 203
Немного о геометрии 208
Собственно о разделах 212
Особенности BSD-разметки 214
RAID и LVM 217
Общие черты файловых систем POSIX-семейства 222
Основные типы файловых систем POSIX-мира 225
Виртуальные файловые системы 233

Глава 10. Файловая иерархия 237

Принципы построения иерархии каталогов 237
Типовой набор каталогов POSIX-системы 239
Корневая файловая система 240
Ветвь /usr 242
Ветвь /usr/local 244
Ветвь /opt 244
Ветвь /var 246
Каталог /mnt 246
Ветвь /home 247
Ветвь /tmp 247
Стратегия разделения файловых систем 248

Интермедия: инструменты дисковой разметки, форматирования и монтирования 251

Разметка диска 251
RAID и LVM 264
Создание файловых систем 276
Монтирование 280
Дополнительные утилиты 284

Глава 11. Терминалы, режимы, интерфейсы 287

Апология консоли 287
Что такое терминал 289

Понятие виртуального терминала 290
О режимах 295
Об интерфейсах 299

Глава 12. Истина — в командах 303

Введение в CLI 303
Командная строка 305
Опции 307
Аргументы 311
Кое-что об исключениях 314
Псевдонимы 315
Переменные 318
Навигация и редактирование 320
История команд 323
Регулярные выражения 325
Командные конструкции 330
Сценарии оболочки: первые представления 336
Понятие о функциях 339
Самая главная команда 340

Интермедия: команды обработки текстов 349

Вступление 349
Просмотр файлов 350
Сравнение, объединение и деление файлов 353
Поиск в файлах 360
Sed: средство потокового редактирования 362

Глава 13. Общесистемное конфигурирование 365

Введение 365
О загрузке и загрузчиках 367
Особенности загрузчиков Lilo и GRUB 369
Задачи инициализации 377
Стили инициализации 381

Глава 14. Принципы сборки и установки пакетов 389

Очень элементарное введение 389
Правила сборки 393
Три волшебных слова 398
Особенности сборки ядра 408
Вопросы оптимизации 412
Средства управления пакетами 417

Глава 15. О шеллах 421

О шеллах вообще 421
Какие бывают шеллы 423

Принципы конфигурирования 425
Проблема выбора 427
Sh-совместимые оболочки 428
Кое что о csh и tcsh....431

Интермедия: командная оболочка zsh....439

Обзор возможностей....439
Приступаем к установке....444
Начинаем настройку....445
Собственно конфигурирование....448
Личный рецепт приготовления....451

Глава 16. Икс — он и в Африке X 463

Кто вы, мистер Икс? 463
Иксы: принципы организации 466
Иксы: сборка из исходников....471
Варианты конфигурирования 473
Варианты запуска....487
Немного о раскладках 491
Шрифты: полиграфическое введение 494
Разборки со шрифтами....497

Интермедия: о менеджерах окон 505

О менеджерах окон 505
Об организации десктопа....507
FLWM, или предел минимализма....507
Blackbox как представитель семейства....509
WindowMaker: как делать окна 512
IceWM: гибкость и настраиваемость....519

Глава 17. KDE: интеграция десктопа 525

Проблема выбора 525
KDE — почему бы и нет 529
KDE как он есть....532
Установка 535
Собственные средства настройки....538
Конфигурационные файлы 554
Детали настройки клавиатуры 555

Интермедия: универсальный konqueror 565

Введение....565
О древовидниках и двухпанельниках....566
Konqueror в его величии....568
Лики konqueror 573

Konqueror в ипостаси браузера 575
Пустячок, но приятный: программа krename 578

Глава 18. Текстовые редакторы 585

Вводные замечания 585
Nano: входной билет к мир редакторов 587
Несколько слов о ее 588
Редакторы vi и Vim: введение в тему 590
Joe: гармония простоты и функциональности 597
NEdit — текстовый редактор для Иксов 612

Глава 19. Html-редактор Quanta Plus 641

О Web-инструментарии вообще 641
Представление героини 644
Главные элементы интерфейса 646
Дополнительные элементы интерфейса 649
Работа с проектами 650
Настройки редактора 652
Дополнительные возможности 657
Итоги 660

Вместо заключения 661

Предметный указатель 662

Преамбула

- Необходимое вступление.
- Зачем эта книга.
- Для кого эта книга.
- О чем эта книга.
- Почему она такая.
- Как она делалась.
- О терминологии.
- References.
- Реверансы.

В этой книге собрана большая часть того, что я знаю про Linux, FreeBSD и отчасти — про другие системы, задумчиво именуемые UNIX-подобными. Называют их также POSIX-совместимыми, и со временем я попытаюсь показать, почему второй термин является предпочтительным. Данное сочинение создается с целью популяризации и даже, не побоюсь этого слова, пропаганды POSIX-систем среди широких кругов компьютерной, околокомпьютерной и некомпьютерной общественности.

Необходимое вступление

В этой преамбуле, а также непосредственно следующих за ней главах, читатель, не сталкивавшийся ранее с UNIX-подобными системами, обнаружит некоторое количество незнакомых слов и терминов (например, POSIX-совместимые системы). Надеюсь, они его не испугают — поверьте, они не заключают в себе ничего сверхъестественного. А смысл их, надеюсь, разъяснится в ближайшее же время (частично — даже в этой преамбуле).

Зачем эта книга

Казалось бы, о UNIX, Linux и BSD за последние годы написано множество книг, статей, сетевых материалов. Нужно ли еще одно сочинение на заданную тему? Думается, что нужно, и по нескольким причинам.

Первая причина в том, что феномен движения Open Source (программ с открытыми исходными текстами) вообще и любые его частные проявления (а Linux и BSD-системы таковыми являются) столь многогранны, что каждый автор, обращающийся к этой тематике, привносит в нее что-то новое (надеюсь, что ваш покорный слуга в своих писаниях не был исключением).

Вторая причина — Linux, BSD и прочие родственные им системы живут и развиваются, и написанное о них даже год назад могло если не устареть (по причинам, которые станут ясными впоследствии, POSIX-системы мало подвержены старению), то в некоторой степени потерять актуальность. И в любом случае будет требовать уточнений, дополнений, корректив, отражающих реалии текущего момента.

Третья причина в том, что Linux-бум конца ушедшего тысячелетия в определенной мере оставил в тени других представителей семейства открытых POSIX-систем. В результате понятие Open Source прочно контаминировалось с ОС Linux, а последняя — с вещами, изначально к Linux никакого отношения не имевшими, такими как оконная система X, интегрированная рабочая среда KDE или офисный пакет OpenOffice.

Четвертая причина — часто встречающееся (и вполне объяснимое) стремление многих авторов объять необъятное. В итоге в толстых книгах о Linux и UNIX говорится о таких материях, как программирование, администрирование локальных сетей и интернет-технологии, да и многих других, не имеющих прямого отношения к операционным системам.

Пятая, с моей субъективной точки зрения, главная причина вытекает из третьей. И она такова: подавляющее большинство известных мне толстых книг, затрагивающих указанную тематику (не говоря уже о статьях и заметках, посвященных частным вопросам), опираются на опыт работы авторов с каким-либо конкретным дистрибутивом Linux или с одним из иных представителей UNIX-клана. Попыток рассмотрения свободных POSIX-систем вообще, без оглядки на конкретные реализации, весьма мало.

В настоящем сочинении я попытался максимально абстрагироваться от конкретных реализаций, воплощенных в том или ином дистрибутиве Linux или какой-либо BSD-системе. И потому в книге речь пойдет, в первую очередь, о том, что их всех объединяет. Особенности конкретных реализаций POSIX-систем отодвинуты на второй план.

Книга основана на уже более чем пятилетнем практическом использовании свободных ОС семейства UNIX, в первую очередь, FreeBSD и пары-тройки дистрибутивов Linux, а также ознакомлении (разной степени поверхностности) со всеми прочими представителями BSD-семейства и с полутора десятками представителей необъятного мира Linux-дистрибутивов.

Должен подчеркнуть, что по жизни я являюсь "чистым" пользователем, не имеющим опыта сетевого администрирования или разработки программ (и

не обнаруживаю ни малейшего желания такой опыт приобретать). А потому и говорить буду только о том, что, по моему скромному мнению, необходимо знать пользователю, а также о том, что ему может быть полезно или должно быть небезынтересно.

Большая часть обсуждаемых здесь вопросов в той или иной мере затрагивалась в моих прежних "бумажных" и онлайновых публикациях. Однако жизнь не стоит на месте, меняются системы, и мы меняемся с ними. И потому я сконцентрируюсь на тех аспектах, которые либо не были в силу различных причин мной затронуты раньше, либо представления о которых сильно изменились.

За последние годы Linux, исторически использовавшийся преимущественно в сфере разработки программного обеспечения и сетевых решениях, все более утверждается в роли операционной системы универсального, в том числе и так называемого домашнего, назначения. BSD-системы в этом аспекте почти никогда не рассматриваются. Однако именно с позиций пользователя разницы между ними почти нет, и они пригодны к настольному/домашнему применению ничуть не меньше, нежели любой из user-ориентированных дистрибутивов Linux. И это я тоже попытаюсь продемонстрировать в настоящем сочинении.

Мое сочинение ориентировано по большей части на пользователей, обладающих некоторым минимумом начальной подготовки (хотя и не обязательно имеющим опыт работы в UNIX-подобных системах). Или, по крайней мере, желанием ее приобрести, возможно, в процессе чтения именно этой книги.

Начинающий пользователь POSIX-систем, как правило, обращается к Linux. Причем именно к тем его разновидностям (дистрибутивам), которые обеспечивают ему наиболее комфортные условия миграции с системы, использовавшейся ранее (рискну предположить, что системой этой, в силу исторических причин, была та или иная версия Windows). Такие дистрибутивы, именуемые user-ориентированными (или дружественными к пользователю), как правило, обладают красивыми и удобными графическими инсталляторами, развитыми средствами универсального конфигурирования системы, богатыми наборами утилит для визуального управления программными пакетами и т. д. И потому их использование на первых порах помогает сломать психологический барьер между привычными объектными интерфейсами и аскетичным на вид исконно UNIX-инструментарием.

Однако довольно быстро к пользователю приходит понимание того, что UNIX — это не Windows, а Windows — это не UNIX. И эффективное использование любого представителя последнего семейства достигается совсем другими способами, на первый взгляд, непривычными, но чрезвычайно мощными и, главное, универсальными. К сожалению, user-ориентированные дистрибутивы, как правило, отнюдь не подталкивают пользователя к их

изучению. Ибо графические инсталляторы и конфигураторы, облегчая, казалось бы, ему жизнь, затеняют при этом внутреннюю сущность явлений.

К тому же все эти средства установки, настройки и пакетного менеджмента в подавляющем большинстве специфичны для конкретных дистрибутивов и подчас заимствуют из Windows дурную привычку существенно меняться от версии к версии. Навыки работы с таким инструментарием, приобретенные в одном дистрибутиве Linux, окажутся мало полезны в другом (и тем более в какой-либо BSD-системе). В результате изначальная универсальность UNIX-систем в значительной мере утрачивается.

Ибо один из факторов, определивших популярность UNIX, — это практически полная неизменность приемов работы во времени, вот уже на протяжении более чем тридцати лет (это не значит, однако, что сами средства работы не совершенствовались), и их независимость от конкретной реализации системы. В итоге пользователь, освоивший традиционный UNIX-инструментарий в рамках любой разновидности Linux, будет столь же свободно чувствовать себя не только в ином дистрибутиве этой ОС, но и в любой BSD-системе или каком-либо проприетарном представителе UNIX-семейства.

Древние греки считали наименее приспособленными к жизни людей, не умеющих читать и плавать. Нравится это или нет, но компьютеры вошли в нашу жизнь прочно и бесповоротно. Так что теперь к списку необходимых умений следует прибавить навыки работы с компьютерами. А потому очень остро встает проблема компьютерного образования. Причем от решения эта проблема далека, нельзя же в самом деле рассматривать в качестве основ компьютерной грамотности минимальное натаскивание для работы в Windows и конкретной версии Word. Причем именно натаскивание на уровне нескольких готовых рецептов: мне рассказывали страшную историю про то, как школьная учительница по информатике поставила ученику двойку за то, что он вышел из Windows через <Alt>+<F4> (следовало обязательно сделать это через меню **Пуск** и его пункт **Выход**).

И в сфере образования роль свободных UNIX-клонов трудно переоценить. Причем не только для тех, кто собирается выбрать специальность, хоть как-то связанную с компьютерами. Традиционные методы работы в UNIX могут использовать представители любых профессий. Причем подчас более эффективно, чем стандартные офисные приложения. Не нужно только впадать в другую крайность и объявлять Linux или, тем более, BSD, средством решения любых задач. UNIX создавался для работы с текстами (любого рода) и обеспечения коммуникаций, и именно в этом он проявляет свою силу. Разве не для работы с текстами и обеспечения коммуникаций использует компьютер подавляющее большинство людей в своей профессиональной деятельности?

Для кого эта книга

Я адресую эту книгу самым широким народным массам. Основную часть которых, конечно, составят достаточно опытные компьютерные пользователи, не имеющие навыков работы в UNIX-системах. Тем из них, кому стало тесно в "подоконном" мире, и кто в целях повышения эффективности своей работы (или просто движимый любопытством) хотел бы ознакомиться с миром POSIX-систем. А там, глядишь, и приобщиться к нему.

И потому вторая адресная категория читателей — уже приобщившиеся, но начинающие пользователи Linux или BSD, только приобретающие навыки работы в POSIX-системах.

Однако я не теряю надежды, что к этому сочинению обратятся и совсем начинающие пользователи компьютера вообще, не затронутые еще воздействием Windows. И мечта эта не столь уж утопична, как может показаться, ведь учиться новому "с нуля" всегда легче, нежели переучиваться. Достаточно иметь только толику любопытства и склонность к поиску и усвоению новой информации.

Наконец, я не исключаю вероятности, что эту книгу будут читать и, так сказать, действующие пользователи POSIX-систем, вне зависимости от сферы приложения их сил. Мне доводилось сталкиваться (и реально, и виртуально) с квалифицированными разработчиками или системными администраторами, имеющими, как ни странно, весьма неопределенное представление о пользовательских аспектах применяемых ими систем. Более того, многие из них просто искренне убеждены, что Linux или BSD народу (то бишь пользователям) — не нужны. Смею верить, что мое сочинение заронит у них сомнение в правоте такой позиции.

И еще: я не хотел бы, чтобы все сказанное на этих страницах воспринималось как некое поучение гуру. Заявляю определенно: гуру не являюсь и быть им не стремлюсь. Все написанное мной — написано для собственного удовольствия и на основании собственного опыта, полученного в ходе жизнедеятельности обычного пользователя.

В любом случае, я надеюсь, что эта книга покажется вам полезной по содержанию и, не в последнюю очередь, интересной по форме. Однако ни того, ни другого гарантировать не могу: вы тратите на нее время исключительно под свою ответственность.

О чем эта книга

Таким образом, эта книга, помимо чистой занимательности, претендует еще и на некоторую познавательность. И ее сюжет подчинен логике зна-

комства пользователя с новой ОС — так, как этот процесс должен был бы проходить в идеальных условиях, скорректированных принудительной силой реальности.

А знакомство пользователя с любой POSIX-системой начинается с ее установки. Конечно, по меткому замечанию Виктора Вагнера (**http://www.45.free.net/~vitus/handbook.html**), это примерно то же самое, что начинать обучение вождению автомобиля с регулировки клапанов или переборки коробки передач. С точки зрения затрат сил и времени, хорошо было бы учиться работать на системе, установленной и настроенной специалистом. Однако принудительная сила реальности такова, что, за редчайшим исключением, начинающий пользователь Linux или BSD вынужден сам установить и настроить систему. Что, конечно, требует знаний и умений. Да вот только приобрести их он может, предварительно получив систему в свое распоряжение, т. е. установив ее и хоть как-то настроив. Эта "уловка 22 от Linux", по выражению Владимира Попова (**http://www2.ldc.net/~popov/init...etc.html**), и составляет одно из главных препятствий для широкого распространения этой ОС (как и других UNIX-подобных систем).

Однако, с другой стороны, прорвавшись тем или иным образом (с помощью чтения руководств или посредством user-ориентированного дистрибутива, руководствуясь ли советами знакомого или задавая вопросы в многочисленных форумах), пользователь приобретает незаменимый ничем объем знаний и навыков. И вспомним другую аналогию: обучение верховой езде (по крайней мере, в школах, заслуживающих этого названия, да и в реальной жизни тоже) начинается именно с обучения чистке, седловке, взнуздыванию (хотя ковать лошадей новичка, пожалуй, заставлять сразу не будут).

Так вот, эта книга и призвана облегчить пользователю понимание процесса установки и настройки произвольной POSIX-системы, а также ее последующего использования.

Большая ее часть представляет собой своего рода общее введение в мир свободных POSIX-систем. Поэтому сначала, в *главе 1*, я не могу не остановиться на таком явлении, как программы с открытыми исходными текстами вообще (Open Source) и свободные программы (Free Software, не путать с Freeware), поскольку речь здесь будет идти исключительно об открытых и свободных представителях POSIX-семейства.

Затем *(в главе 2)* вас ожидает рассмотрение вопроса о том, что такое операционная система вообще, POSIX-совместимые (или UNIX-подобные), в частности, и Linux и BSD-системы — в особенности.

Понимание POSIX-систем невозможно без представления их истории — хотя эта тема интересна и сама по себе. Так что она будет рассмотрена в *главе 3*. На чем вводную часть книги можно считать законченной.

В последующих главах следует обзор "вечных истин" POSIX-систем. В частности, *глава 4* посвящается специфике POSIX-совместимых систем и принципиальному отличию методов их использования от ОС семейства Windows.

Дальнейшее изучение любого свободного UNIX-клона, как уже было сказано, вынужденно происходит одновременно с установкой и настройкой этой системы. И в *главе 5* рассмотрены принципы установки POSIX-совместимой ОС, вне зависимости от конкретной реализации. Поэтому здесь не будет описаний работы инсталляторов и конфигураторов того или иного конкретного дистрибутива Linux или BSD. Вместо этого я постараюсь рассказать о внутренней сущности действий при установке любой POSIX-системы, о том, что обычно в user-ориентированных дистрибутивах Linux (а ведь именно с них, скорее всего, и начнет свое знакомство начинающий пользователь) остается за кадром графического интерфейса.

Главы с 6 по 10 посвящаются "трем китам", на которых зиждется POSIX-мир: понятиям пользователя *(глава 6)*, процесса *(глава 7)* и файла *(глава 8)*, а также физической *(глава 9)* и логической *(глава 10)* организации файлов. Интермедии, вклинивающиеся в основной сюжет, посвящены управлению пользовательскими акаунтами, файлами и файловыми системами.

При этом у читателя не предполагается никаких предварительных знаний на сей предмет. Я постарался описать эти материи в форме, понятной начинающему пользователю, — насколько это у меня получилось, судить вам.

В *главе 11* описываются связывающие звенья между пользователем системы, протекающими в ней процессами и составляющими ее файлами, охватываемые понятиями терминалов, режимов их работы и их интерфейсов. А в *главе 12* подробно описываются принципы основного пользовательского интерфейса POSIX-мира — интерфейса командной строки. Следующая за ней интермедия содержит описание одной из важнейших групп пользовательских команд — средств для работы с текстами.

Глава 13 посвящена принципам настройки системы. Она не подразумевает использования какого-либо определенного дистрибутива Linux или конкретной BSD-системы, будучи применима ко всем ОС POSIX-семейства. Моменты, специфичные для какой-либо системы или дистрибутива, выделены особо.

Любая операционная система устанавливается, изучается и настраивается, в большинстве случаев, ради ее практического применения. А это требует понимания принципов управления пакетами — дополнительным программным обеспечением, не входящим в состав собственно операционной системы. Что и составит предмет *главы 14*.

Глава 15 развивает тему командного интерфейса и посвящена она первой и одной из важнейших пользовательских программ — командной оболочке (shell), точнее их многочисленным разновидностям. А в следующей за ней

интермедии описывается наиболее развитый и функционально богатый представитель этого семейства — zsh.

Операционные системы POSIX-семейства изначально создавались для работы с текстами (в самом широком смысле слова) и потому родной их режим текстовый. Однако никаких причин отказываться от использования графики в них также не имеется. И поэтому *глава 16* будет посвящена универсальной графической метасистеме POSIX-мира — оконной системе X (или, в просторечии, Иксам). И соответствующая интермедия рассказывает о программах, делающих возможным ее использование, — менеджерах окон.

В следующей *главе 17* речь пойдет о среде, обеспечивающей пользователю взаимодействие с этой метасистемой, — интегрированном десктопе KDE. Последующая же интермедия описывает главную программу этой среды — konqueror, универсальное средство манипулирования файлами.

18 глава посвящается классу программ, лежащих на грани между общесистемными и пользовательскими — текстовым редакторам. А последняя *19 глава* — HTML-редактору Quanta Plus, т. к. обработка HTML-документов есть частный случай работы с текстами вообще.

Почему она такая

Форма книги связана с одним из фундаментальных понятий UNIX — рекурсией. В собственном смысле слова это специальный программистский термин, и означает он определение функции через себя (что такое функция, надеюсь, станет понятным из *главы 12*).

Однако в UNIX-сообществе понятие рекурсии широко применяется в бытовом, так сказать, смысле. Известным чему примером является акроним проекта GNU (о котором будет говориться в *главе 2*).

Так вот, весь процесс изучения POSIX-систем пронизан рекурсией. Возьмем для примера три фундаментальных понятия мира UNIX: понятие файла, процесса и пользователя. Отношения между ними неизбежно определяются друг через друга: каждый файл принадлежит какому-либо пользователю в силу того, что он (файл) порожден процессом, запущенным этим пользователем (на самом деле все еще сложнее, но это пока неважно).

Чтобы понять, что такое файл и его атрибуты, нужно предварительно разобраться в том, что такое процесс и атрибуты оного. Что, в свою очередь, невозможно без представления об атрибутах исполняемого файла этого процесса и атрибутах пользователя, запускающего этот процесс на исполнение. А если вспомнить, что для получения информации о файлах, процессах и пользователях необходимо использовать команды, каковые и сами представ-

ляют собой файлы с определенными атрибутами и при этом используют в качестве своих аргументов файлы же, атрибуты которых также существенны, становится ясным: пользователь, начинающий изучение POSIX-систем, сталкивается не с чем иным, как с рекурсией. Ведь для того, чтобы осознать базовые понятия UNIX, необходимо обладать навыками работы с командами, а чтобы понять, как работают команды, и использовать их осмысленно, а не механически, требуется знание базовых понятий.

Именно в такой рекурсивной связи и кроется, на мой взгляд, основная сложность изучения POSIX-систем — для начинающих пользователей, и трудность написания книг, этому пользователю адресованных, — для авторов. И в рамках традиционной книжной структуры противоречие это неразрешимо. Ведь книга по определению линейна и читается, как правило, с начала и до конца (по крайней мере, авторам книг обычно хочется, чтобы их произведения читали именно так). И как, при линейной-то структуре, прикажете обращаться с рекурсивным определением понятий? Отсылать (так и хочется сказать — посылать) читателя к еще непрочитанным главам, возможно, через многие сотни страниц? Эдак он быстро мозоли на пальцах натрет — от листания...

Поэтому я после долгих размышлений и отказался от линейного стиля изложения в этой книге, придав ей иерархическую структуру, подобную древовидной файловой системе UNIX (о которой будет говориться в *главе 10*). Каркас этой структуры (т. е. подобие корня файловой системы) образуют главы, содержание которых было вкратце охарактеризовано в предыдущем параграфе. Они представляют собой своего рода введение в POSIX'ивизм, рассчитанное, в том числе, и на начинающего пользователя. Главы чередуются с интермедиями, каждая из которых посвящена детализации понятий, рассмотренных в соответствующей главе.

Предполагается, что пользователь сначала ознакомится с элементами каркаса — главами, после чего перейдет к изучению материала, представленного в интермедиях. Пользователь же, обладающий определенным опытом работы в UNIX-подобных системах, возможно, обнаружит в "корневом разделе" мало нового для себя и обратится сразу к интермедиям.

Важно также, что материал из "корневого раздела" книги, за редкими, специально оговоренными, исключениями имеет силу для любой POSIX-совместимой системы. Аспекты же, специфичные для Linux, отдельных его дистрибутивов или какого-либо представителя BSD, по возможности сосредоточены в интермедиях. Кроме того, в них же мне показалось уместным собрать все конкретные примеры и рекомендации, в том числе на уровне "делай, как приказано" (если, конечно, сам не знаешь, как делать).

Как она делалась

Излишне говорить о том, что все это сочинение, от первой до последней страницы, осуществлялось на машине, оснащенной исключительно свободными программами. В качестве операционной системы в разное время написания выступали дистрибутивы Linux: Gentoo (**http://www.gentoo.org**), CRUX (**http://www.crux.nu**) и Archlinux (**http://www.archlinux.org**). Кроме того, изрядная часть книги сочинялась под управлением FreeBSD (**http://www.FreeBSD.org**). Подготовка электронной версии книги (**http://linuxcenter.ru/lib/books/posixbook**) была проведена на машине с ОС DragonFlyBSD (**http://www.dragonflybsd.org/main/**). А завершающая доводка и компоновка книги для печатной публикации выполнена в Debian GNU/Linux (**http://www.debian.org**) и Kubuntu Linux (**http://www.kubuntu.org**). Этим я хочу в очередной раз подчеркнуть, что все POSIX-системы (и все дистрибутивы Linux) практически равноценны с точки зрения настольного применения.

Основным инструментом собственно сочинительства в разное время были текстовые редакторы joe (в консоли), nedit (в системе X с оконным менеджером fluxbox) и kate (в интегрированной среде KDE, **http://www.kde.org**). Текст набирался непосредственно в HTML-формате с помощью макросов собственного изготовления. Поскольку в этой книге в актуализированном и модернизированном виде использованы кое-какие тексты из ранее мной написанного (как опубликованного, так и неопубликованного), проблема поиска необходимых фрагментов, их модификации и помещения, куда следует, решалась с помощью командных утилит, пришедших из мира классического UNIX: (find, grep, cat, split, sed). Итоговый вариант книги компоновался и доводился в HTML-редакторе Quanta Plus. Разбиение единого первоначально текста на главы выполнено утилитой split. И, наконец, для передачи книги в издательство пришлось "прогнуться под испорченный мир" — т. е. конвертировать HTML-документы в формат MS Word. Эта нелегкая задача была решена с помощью пакета Openoffice.org.

По всему тексту я старался придерживаться следующих условных обозначений. Команды в командной строке, примеры их экранного вывода, фрагменты конфигурационных файлов и сценариев передаются моноширинными гарнитурами:

```
Так даются примеры команд,

примеры их экранного вывода,

конфигурационных файлов,

листинги сценариев
```

При этом команды предваряются символом приглашения командной строки, в качестве которого я принял `$`. Символ же `#` закреплен за коммента-

риями в скриптах и конфигурационных файлах. Разумеется, если синтаксис данного скрипта или конфига требует иного обозначения комментария (например, `!`), — будет использоваться именно он (с соответствующей оговоркой).

Команды, их опции и аргументы, упоминаемые в теле абзацев, также передаются моноширинным шрифтом: `имя_команды`.

Если же название программы с запускающей ее командой не совпадает — оно дается обычным начертанием пропорциональной гарнитуры.

Все примеры команд, конфигурационных файлов и сценариев в книге взяты из реальных систем, использовавшихся автором, где они были работоспособны. Конечно, гарантировать их функционирование в произвольной ОС или дистрибутиве я не могу, однако они могут быть использованы как основа для собственных модификаций.

Важную роль в структуре книги играют ссылки на источники информации. В первую очередь это официальная документация к описываемым системам и программам, представленная преимущественно в форме так называемых man-страниц (подробно описанных в *главе 12*). Ссылки на такие документы выглядят следующим образом: `man (#) name`, где `name` — имя объекта документирования (например, команды), а `#` — номер тематического раздела, если он необходим.

Не менее важны источники информации в Интернете. Они оформлены традиционно, **http://www.posix.ru**. Кстати, именно на этом сайте будет дан полный список использованных источников.

Я старался не злоупотреблять традиционными смайликами. Надеюсь, что читатель догадается, когда они подразумеваются по умолчанию (а подразумеваются они во многих местах).

О терминологии

Информационные технологии и Computer Sciences имеют международный характер. Однако в силу ряда обстоятельств развивались они преимущественно в англоязычной среде. Можно сказать, что американский английский — родной язык IT-пространства. Особенно ярко это проявляется в мире Open Source, для разработчиков которого, рассеянных по странам и континентам, говорящих на самых разных языках и формально никак друг с другом не связанных, английский выступает в качестве lingua franca — его можно сравнить с латынью средневековой науки.

В связи с этим перед автором любого сочинения на компьютерную тему вообще, и на тему Open Source в особенности, пишущего на своем родном языке (отличном от американского английского), встает проблема адекват-

ной передачи специфической терминологии, имеющей, за редким исключением в виде латинских и древнегреческих корней, англоязычное происхождение. В отечественной компьютерной литературе (как традиционной, бумажной, так и сетевой) сложились две устойчивые терминологические традиции: пуристическая и жаргонная.

Пуристическая традиция характерна для авторов, ориентирующихся на книжные издательства и "толстые" журналы, позиционирующие себя в качестве профессиональных. Суть ее в подборе для исходно английских терминов максимально "русских" эквивалентов. Слово "русских" я не случайно взял в кавычки, причина станет ясна через пару абзацев.

Жаргонная (сленговая) традиция преобладает в онлайновых публикациях, роль которых в области Open Source, как минимум, сравнима с ролью традиционных "бумажных" изданий. Она выражена в простой транслитерации (часто с подчеркнутым нарушением ее правил) соответствующих английских слов и аббревиатур.

Пуристическая терминология всегда ведет к утяжелению текста. Например, вместо короткого английского слова software или даже просто soft употребляется неуклюжее описательное определение "программное обеспечение". И это еще не самый тяжеловесный оборот — читатель любого компьютерного издания легко может умножить количество примеров.

Кроме того, борьба за чистоту русского компьютерного языка иногда влечет за собой просто комический эффект, когда один "чуждый по происхождению" термин определяется через слово столь же "заграничное", только, в отличие от первого, вошедшее в русский язык раньше (на века или годы, в данном случае не важно). Тут достаточно вспомнить просто анекдотичную борьбу за замену чуждого иностранного слова "менеджер" исконно русским "диспетчер". Что вызывает в памяти известного адмирала Шишкова, боровшегося за чистоту русского языка еще в XVIII веке (хотя подозреваю, что для адмирала это было формой своеобразного юмора).

Второй основной недостаток пуристической терминологии — ее неоднозначность, вплоть до полного искажения смысла. Так, перевод английского термина Window Manager как "диспетчер окон" способен только ввести в заблуждение человека, недостаточно знакомого с предметом.

Поэтому в настоящей книге (как, впрочем, и во всех своих сочинениях) я категорически порываю с пуристической традицией. Хотя это не значит, что я призываю к полному переходу на терминологию жаргонную, также не свободную от недостатков: она не вполне изящна и не всегда понятна начинающим. Однако имеет неоспоримое достоинство — однозначность, унаследованную от англоязычного прототипа; по крайней мере, в контексте данной тематики.

В качестве примера можно рассмотреть попытки подобрать родной эквивалент термину Distributions, что по-русски транскрибируется обычно как "дистрибутив" (или даже жаргонное "дистр"). Очевидно, что попытки эти будут тщетны, ведь это слово имеет совершенно различный смысл в разных сочетаниях. Например, значения FreeBSD Distributions, Linux Distributions или Windows Distributions — абсолютно разные: в первом случае имеется в виду базовая комплектация операционной системы FreeBSD, во втором — один из (многочисленных) способов распространения операционной системы Linux, в третьих же — диск с надписью и голограммой либо его более-менее точное пиратское воспроизведение. По-английски смысл любого подобного выражения ясен из контекста, русский же их перевод потребует различных (и очень сложных) описательных конструкций.

Кроме того, компьютерные жаргонизмы настолько вросли в окружающий нас мир (особенно — в мир онлайновых источников информации), что полный отказ от них возможен только ценой потери ясности изложения. В сущности, жаргонные термины давно уже превратились в своего рода идеограммы, достоинство которых — в однозначности понимания людьми, говорящими на самых разных языках. Тысячелетняя история китайской письменности доказала жизненность идеографического подхода, по крайней мере — в области специальной терминологии.

Примером может служить поиск адекватной русской передачи названия для графического метаинтерфейса UNIX — X Window System, по-английски называемой также X System или просто X. Обычно в качестве таковой предлагается достаточно неуклюжее словосочетание "система X Window" (а подчас, даже система X Windows). Дословный же перевод здесь будет "оконная система X", где компонент X и в английском, и в русском языке играет роль идеограммы, а не языковой конструкции (почему — будет рассказано в *главе 17*). И потому жаргонное выражение "Иксы", как абсолютно адекватное и однозначное, вполне имеет право на существование.

Конечно, использование жаргона требует взвешенного подхода. Нелепо переносить английские кальки в области, где существует развитая, устоявшаяся для русского языка терминология, например, в область полиграфии и шрифтовых технологий. Однако во многих случаях именно жаргонизмы (подобно упомянутым Иксам) позволяют адекватно выразить суть явления.

Поэтому в настоящем сочинении я довольно широко употребляю общепринятые (и общепонятные в компьютерных кругах) жаргонизмы. Хотя, каюсь, в ряде случаев использую их и просто для достижения большей выразительности, подобно "местным идиоматическим выражениям" в художественной литературе.

Впрочем, при первом вхождении термина я стараюсь дать все известные синонимы — как пуристические, так и жаргонные, а при необходимости привожу и оригинальное англоязычное написание. Это вызывается тем,

что русскоязычная терминология в околокомпьютерной области далека от унификации, а достижение последней — задача, для одного человека непосильная. Да, возможно, и ненужная, отказ от синонимичности в языке (особенно в русском, синонимичном по сути своей) часто приводит к потере его выразительности.

References

Основу этого сочинения составили заметки для серии моих прежних сайтов, перечислять их было бы долго, на текущий момент функционирует **http://www.posix.ru**, на форуме которого (**http://forum.posix.ru**) можно высказать свои соображения о данной книге. Здесь я должен оговориться — в случаях расхождений с ранее мною написанным (а они будут довольно часты), в качестве "истины текущего момента" должен рассматриваться именно настоящий текст.

В своих сочинениях я опирался на множество источников. В первую очередь это официальная документация ко всем упоминаемым по ходу дела программам в формате man и, реже, info, а также HTML-, XML- и PDF-документация на сайтах различных проектов Open Source.

Далее, я использовал бесчисленные сетевые материалы с многочисленных русско- и англоязычных сайтов, посвященных тематике UNIX, Linux и Open Source, ссылки на которые даются по ходу изложения.

На протяжении последних 10 лет тема Open Source, особенно Linux, занимает все большее место на страницах традиционной "бумажной" компьютерной прессы. Я, по мере сил, слежу за такими публикациями и при необходимости использую их в своей работе, так что не избежать и ссылок на них (и, по возможности, их онлайновые версии).

Разумеется, не обошелся я и без чтения "толстых" книг по UNIX и Linux. Из них наибольшее влияние на меня оказали:

- Б. В. Керниган, Р. Пайк. UNIX — универсальная среда программирования. — М.: Финансы и статистика, 1992; ей, купленной много лет назад совершенно случайно, суждено было стать моей настольной книгой на протяжении всего последующего времени (новый перевод с того же английского издания ныне продается под названием "UNIX: программное окружение");
- Д. Тейнсли. Linux и UNIX: программирование в shell. — К.: Издательская группа BHV, 2001; книга, содержащая детальное описание работы в shell, укрепила меня в мысли, что это самый эффективный инструмент пользователя всех времен и народов;

- Дж. Армстронг. Секреты UNIX. — М.: Издательский дом "Вильямс", 2001; прочитав эту книгу не так давно, я очень пожалел, что она не попала мне в руки раньше — ее чтение позволило бы сэкономить немало сил и времени при освоении POSIX-систем; эта книга — та самая, которую непременно нужно взять с собой на необитаемый остров с UNIX-машиной; именно она служила мне эталоном, к коему я стремился приблизиться в своем сочинении.

Наконец, множество сведений я почерпнул в ходе переписки с моими многочисленными корреспондентами и в общении на различных форумах.

В заключение раздела стоит привести некоторые ссылки:

- все упомянутые в книге ОС и дистрибутивы могут быть приобретены в онлайновом магазине Линуксцентра на **http://linuxcenter.ru**;
- их же, плюс многочисленные приложения для них "россыпью", можно скачать с FTP-архива: **http://www.citkit.ru**, выбрав их посредством Web-интерфейса;
- дополнительную информацию по большинству затронутых в книге вопросов можно найти в материалах сайта Citforum: **http://www.citforum.ru**;
- обратиться со своими вопросами и проблемами по установке, настройке и использованию POSIX-систем можно на Линуксфорум: **http://www.linuxforum.ru**;
- и, наконец, на **http://www.posix.ru** можно не только обсудить эту книгу (**http://forum.posix.ru/viewtopic.php?id=8**) и другие материалы родственной тематики, но и поместить собственные сочинения.

Реверансы

Таким образом, для создания этой книги привлекались плоды работы множества людей. И потому, пользуясь случаем, я хотел бы выразить свою искреннюю признательность им всем — знакомым, очно или заочно, и совсем незнакомым:

- основоположникам идеи Open Source и всему Open Source Community;
- разработчикам всех использованных операционных систем и сборщикам упомянутых в тексте дистрибутивов Linux;
- разработчикам свободных программ, использованных мной при подготовке книги (и всех используемых мной "по жизни");
- авторам официальной документации к программам и всех прочих использованных мною материалов, печатных и электронных;
- авторам и переводчикам материалов проекта **http://unix.ginras.ru** и **http://www.posix.ru**; командам фирмы Линуксцентр (**http://linuxcenter.ru**)

и интернет-портала Citforum (**http://www.citforum.ru**, с которыми я связан сотрудничеством на протяжении последнего времени);

- всем своим корреспондентам по электронной переписке;
- постоянным участникам Линуксфорума (**http://linuxforum.ru**), которым я обязан многими описанными здесь соображениями, и общение с которыми для меня не только полезно, но и приятно;
- родным и близким многих и многих пользователей, которые выступали в качестве подопытных кроликов, оценивая удобопонятность первоначальных версий ряда глав книги.

Прошу прощения, но полное поименное перечисление в этом реверансе заняло бы не один десяток страниц. Тем не менее, отдельные благодарности (не ищите скрытого смысла в порядке перечисления — он абсолютно случаен):

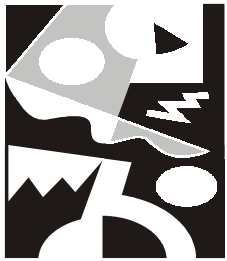
- постоянному товарищу по онлайновым проектам и виртуальному (а подчас и реальному) собеседнику Владимиру Попову (**http://www2.ldc.net/~popov**);
- моему товарищу, коллеге и воинскому начальнику Сергею Соколову, зав-лабораторией Геологического института РАН (**http://north-east.ginras.ru**), и всем сотрудникам нашей лаборатории;
- руководителю информационной службы нашего института Дмитрию Кудрявцеву (**http://www.ginras.ru**);
- бессменному администратору наших серверов Игорю Борейко;
- многолетнему соратнику по компьютеризации геологии Кириллу Крылову, покинувшему отчизну не корысти для, но ради счастья в личной жизни;
- моему самому старому товарищу по ремеслу компьютерщика Владимиру Родионову (**http://www.rwpbb.ru**);
- команде издательства "БХВ-Петербург" и лично Евгению Рыбакову и Елене Кашлаковой, приложившим немало сил для того, чтобы эта книга увидела свет.

Особая благодарность моей жене Лене и моим детям Оле и Вите — за сам факт их существования...

Все перечисленные (и не перечисленные) лица оказали в той или иной мере положительное влияние на данное сочинение. Тем не менее, написано оно мною, и я принимаю на себя всю полноту ответственности за его (несомненные) достоинства и (вполне возможные) недостатки.

Окончательный вариант книги был подготовлен по заказу Линуксцентра **http://www.linuxcenter.ru** и при его поддержке.

Глава 1



Открытость, свобода и халява

- Постановка вопроса.
- Степени свободы.
- Грани открытости.
- Кое-что о лицензиях.
- Истоки Free Software.
- Кто оплачивает банкет.
- Можно ли заработать на Open Sources.
- Как же заработать на Open Sources.
- О продолжении банкета.

Предметом всего дальнейшего изложения будут почти исключительно операционные системы, их утилиты и приложения, относимые к категориям программ открытых (Open Source буквально — программы с открытыми исходными текстами) и свободных (Free Software, что в данном контексте имеет значение "свободно распространяемые", но не обязательно "бесплатные", о чем будет подробно сказано далее).

Постановка вопроса

Понятия программ открытых (Open Source) и свободных программ (Free Software) близки как по букве, так и по духу, но не идентичны, ибо сосуществуют как бы в разных, хотя и пересекающихся, плоскостях. Кроме того, и то, и другое явление тесно соприкасаются с понятием открытых систем (Open Systems).

И потому для начала следует прояснить вопрос о том, что такое открытость и свобода, каково их соотношение друг с другом. Кроме того, большинство упоминаемых далее программ в определенном смысле бесплатны, так что

стоит остановиться и на понятии открытости, свободы и той бесплатности, которая выражается этим сладким словом "халява", столь милым сердцу россиянина (да и вообще выходца с бывших рублевых пространств).

Определиться с понятиями открытости и свободы программ тем более важно, что они часто служат источником недоразумений, основанных на незнании существа вопроса и его истории, буквальности перевода соответствующих английских терминов и прочих причинах информационного характера. На чем, в свою очередь, подчас базируется прямое или косвенное передергивание, выражающееся в типичной подмене понятий, когда теплое путается (осознанно или неосознанно) с мягким.

Сразу оговорюсь, что все описанное далее в этом разделе основано на документах движений Open Source (**http://www.opensource.org**), Free Software Foundation (далее FSF — **http://www.fsf.org/**) и проекта GNU (**http://www.gnu.org**), а также публичных высказываниях их лидеров. Однако я излагаю свое понимание предмета, которое отнюдь не обязано совпадать ни с одним из первоисточников (каковые, не будучи марксизмом, являются не догмой — руководством, и не столько к действию, сколько к размышлению).

Степени свободы

Начнем с понятия Open Source Software, как наиболее, с моей точки зрения, общего. Это термин чисто технический и применяется к программам, чьи исходные тексты принципиально общедоступны без дизассемблирования и прочих "хакерских" приемов.

Обращаю внимание на кавычки, в которые заключено слово "хакерские". Потому что расхожее представление о хакерах как компьютерных взломщиках, злоумышленниках и прочих хулиганах неверно — перечисленные явления объединяются понятием "крэкерство". Хакеры же — это как раз, наоборот, мирные и добронамеренные люди, занятые созидательным компьютерным трудом на благо общества. В частности, движение Open Source и Free Software — в существенной мере их рук дело.

Подчеркну, для определения Open Source важна именно принципиальная возможность доступа к исходникам, ибо конкретная форма такового может быть самой разной: платной или бесплатной, безусловной или с соблюдением определенных условий (например, запретом дальнейшего распространения или коммерческого использования). Смысл же общедоступности в том, что любой человек, придя с улицы и, возможно, выполнив указанные условия (заплатив деньги или согласившись с ограничениями на распространение), получает исходные тексты данной программы.

В этом контексте Open Source противопоставляется программам закрытым. То есть тем, исходные тексты которых не распространяются вовсе или распространяются среди определенного круга — коммерческих партнеров, разработчиков или, скажем, правительственных организаций. И потому, например, декларированное не так давно открытие кодов Windows — именно для правительственных организаций не делает эту систему открытой: некий произвольный индивидуум (или предприятие) все равно не может получить доступ к исходникам этой ОС по своему желанию.

Тем, кто помнит времена развитого (и всякого прочего) социализма, будет ясна аналогия: Open Source можно уподобить материалам, опубликованным в открытой, как тогда говорили, печати (обратите внимание на совпадение терминов), закрытый же софт — материалам "Для служебного пользования" (ну и прочим "Секретным" и "Совершенно секретным"). То есть для ознакомления с первыми было достаточно желания потребителя (от цены отвлечемся — даже газету "Правда" и при социализме бесплатно не раздавали), со вторыми требовалось разрешение тех же компетентных органов. И не важно, что в качестве последних мог выступать один орган — директор предприятия, да и давалось это разрешение практически кому угодно (в случае материалов ДСП, конечно, доступ к сведениям секретным был строго регламентирован). Главное, что об этом нужно было просить по установленной форме. А в компетенции разрешающего органа было наложить резолюцию: "Разрешить. Иван Иваныч". Или — "Не разрешить. Иван Иваныч"...

Таким образом, понятием Open Source определяется первая степень свободы — свободы изучения исходников. Рискуя повториться, подчеркну: открытые исходники сами по себе не подразумевают ни свободы распространения, ни бесплатности оного. Хотя с первой связаны почти неразрывно. Свидетельством тому судьба всех проектов, исходники которых открывались: либо они становились истинно свободными, либо закрывались обратно.

Вторая степень свободы определяется понятием свободного софта — Free Software, что не следует путать ни с Freeware, ни с Free Software Foundation, о котором я скажу попозже. Это понятие — более узкое, нежели Open Source Software, и охватывает аспекты использования и распространения программ. Оно базируется на знаменитых принципах свободного софта, сформулированных Ричардом Столлменом (Richard M. Stallman, известный в миру как RMS), основоположником движения за свободный софт вообще и FSF (Free Software Foundation — организация движения за открытые исходники) в частности: свободе использования, свободе изучения и модификации, свободе распространения.

Свобода использования подразумевает, что копия (конкретный экземпляр) любой программы из категории Free Software, приобретенная пользователем (как — пока не важно, об этом еще пойдет речь), становится, подобно лю-

бому другому товару, в соответствии с законами людскими (которые не следует путать с государственными законами) и просто здравым смыслом, полной его собственностью. И может быть использована по его усмотрению: установлена на один компьютер или на энное их количество, на локальную машину или на сетевой сервер, продана за углом, подарена и т. д. Повторяю, речь в данном тезисе идет именно о конкретном экземпляре на некоем носителе — пользователю вольно этот самый носитель засунуть себе в... ящик стола, прибить на стенку или использовать в качестве подставки под пивную кружку.

Свобода изучения и модификации подразумевает следующий шаг: пользователь вправе изучать устройство программы, выявлять в ней ошибки (а кто видел безошибочную программу?) и исправлять оные, вносить любые изменения, какие ему покажутся нужными, адаптировать под свои задачи и так далее. Очевидно, что реализовать это право он может только при наличии непременного условия — доступа к исходным текстам. И именно поэтому Free Software — лишь частный случай понятия Open Source.

Наконец, свобода распространения относится уже не к правам на экземпляр программы, а к условиям ее тиражирования. То есть пользователь имеет право копировать программу Free Software и распространять ее любым угодным ему способом: раздавать в переходе метро или продавать, причем — за любые деньги, в которые он оценит свои усилия по тиражированию, амортизацию своего оборудования и прочие накладные расходы. Разумеется, в рамках действующего по сему поводу законодательства данной страны, но к программному обеспечению, как таковому, это отношения уже не имеет.

В сущности, на свободу распространения Free Software накладывается только два ограничения, также напрямую с природой предмета распространения не связанные и вытекающие из других общепринятых в цивилизованном мире норм. Первое ограничение есть следствие авторского права в его исконном, неимущественном, понимании (как возникающего вследствие факта создания автором произведения). И оно запрещает распространителю приписать себе авторство, в данном случае — объявить себя разработчиком продаваемой (или раздаваемой) программы.

Второе же ограничение базируется на понимании свободы как таковой (не по Марксу, опять же, а по здравому смыслу), т. е. на запрете ограничивать свободу других. И потому распространитель не вправе запретить другому воспользоваться теми же правами, что и он сам — правом свободного использования, изучения, модификации и распространения, в том числе и за деньги.

А где же здесь бесплатность, резонно спросите вы меня? А нигде, столь же резонно отвечу я вам. Потому что ни понятие Open Source, ни понятие Free Software к понятию халявы никакого отношения не имеют. Каким бы способом вы ни приобретали свободную программу, вам придется заплатить: за

трафик при получении по Интернету, за носитель, его оформление и доставку при покупке на компакт-диске, за полиграфию упаковки и печатной документации при покупке коробочной версии и т. д. Включая оплату накладных расходов распространителя (вплоть до, возможно, даже его затрат на рекламу). Короче говоря, за все то же, за что вы платите при покупке бутылки пива или автомобиля.

Другое дело, что при покупке Free Software вы платите только за это, и ни копейкой больше. Ни о какой оплате интеллектуальной собственности речи быть не может. Даже в том случае, если распространитель продает собственный (или собственноручно доработанный) продукт.

При покупке Windows вы приобретаете не товар, а право на его использование, ограниченное теми условиями, какие заблагорассудится поставить разработчику. Конечно, ваше право — принять их или не принять (так называемый договор присоединения). Но во втором случае право на использование вами теряется.

В сущности, даже право собственности на носитель и его содержимое вами не приобретается: в большинстве случаев условия лицензионного соглашения запрещают не только продажу, аренду и т. д., но и дарение или передачу "на попользоваться". Раньше в этих соглашениях бывали оговорки типа запрещения только одновременного использования приобретенного экземпляра программы. Но в последнее время это признано, видимо, вредным либерализмом, и из обихода изъято.

Маленькое отступление

Вообще мне все эти лицензии на проприетарный софт очень напоминают не "договор присоединения" (как это часто квалифицируется), а скорее, перечень прав и обязанностей пассажира советского (да и российского) общественного транспорта. Где вы найдете множество фраз типа "пассажир обязан", "пассажир должен", "пассажир не имеет права" и ни слова о том, что пассажир может рассчитывать хотя бы на то, что его довезут до конечной остановки ("Поезд дальше не пойдет, просьба освободить вагоны!"). Хотя тоже своего рода договор присоединения: не нравится — ходи пешком...

Понятие Free Software не имеет также непосредственной связи с таким явлением, как Freeware — этим термином обозначаются программы, которые их разработчик по каким-либо соображениям (из врожденного альтруизма или в целях саморекламы) предоставляет всем желающим безвозмездно (т. е. даром — но за трафик, доставку и носитель заплатить все равно придется). В отличие от Free Software, никакой свободы, за исключением свободы использования, да и то не всегда (встречаются такие явления, как: хочешь на второй компьютер — качай заново), Freeware не подразумевает: ни тебе свободы изучения, ни свободы модификации, ни, тем более, свободы распространения. Да и изучать или модифицировать

особенно нечего: подавляющее большинство Freeware-программ распространяются без исходников.

Еще одно отступление: по моим наблюдениям, статус Freeware в большинстве случаев — сугубо временный, чем-то эти программы сродни товарам, приобретенным на рекламных распродажах и тому подобных акциях. Очень мало разработчиков принципиально декларируют Freeware-статус своих произведений. Это и есть случай врожденного альтруизма; на память приходит, пожалуй, только Пауль Лютус с Arachnophilia (**http://www.arachnoid.com**). В большинстве случаев Freeware-проект либо превращается в коммерческий, либо, если коммерческий успех не светит, тихо и незаметно умирает.

Вернемся, однако, к явлению Free Software. Для его понимания очень важно уяснить себе, что принципы Ричарда Столлмена, на которых оно основывается, носят исключительно разрешительный, а не обязующий характер. То есть вы имеете право, если вам так хочется, продавать свободный софт или распространять его даром. Но никто не обязывает вас это делать, например, нарезать дистрибутивы первому встречному по его требованию. Последнее понимание идеи свободного софта подчас встречается среди лиц, отягощенных пережитками социализма в сознании...

Далее, принципы Ричарда Столлмена отнюдь не обязательны. Никто под дулом автомата не заставляет автора программы раскрывать свои исходники. Как не возбраняется и изменение статуса распространения программы: свободный Open Source волею разработчика (и это следствие того же авторского права) может превратиться во вполне закрытый продукт (с ограничениями, накладываемыми свободными лицензиями, о чем — в следующем разделе).

Конечно, Ричард Столлмен, максималист по натуре, своими высказываниями дает некоторый повод к превратному его пониманию. В частности, один из сквозных его тезисов — все программы должны быть свободными, не более, чем пожелание (или, если угодно, этический императив), на самом деле не означающее, что авторов несвободных программ будут понуждать к чему-нибудь, с их точки зрения, противоестественному.

Вообще, следует отделять сами по себе принципы Столлмена (под каждым из которых я готов подписаться обеими руками) от их трактовки им же. Однако тут мы плавно переходим к третьей степени свободы.

И это свобода в понимании Free Software Foundation и проекта GNU (GNU is Not UNIX, проект воспроизведения функциональности этой ОС с "чистого листа"), каковые опять же являются детищами Ричарда. Вследствие понятной аберрации Free Software часто рассматривается как эквивалент Free Software Foundation. Однако это не так.

Конечно, сам Столлмен сотоварищи в качестве Free Software рассматривает именно свободный софт в понимании FSF — все прочее, по его мнению, принадлежит миру Open Source. Однако это тот самый результат расширенной трактовки собственных принципов: если придерживаться буквы последних, то становится ясным, что подавляющее большинство программ из разряда Open Source столь же свободны, как и разработки под эгидой FSF. Однако к этому вопросу мы еще вернемся.

А пока следует подчеркнуть, что свободный софт в понимании FSF — отнюдь не единственная разновидность свободного софта вообще. Более того, сам Столлмен, несмотря на приписываемый ему (и не совсем безосновательно) экстремизм, отнюдь не настаивает на том, что его свобода — единственно возможная. Да и на сайте FSF есть целый список условий свободы, которые рассматриваются как "совместимые" со свободой в высшем понимании (FSF, разумеется; как тут не вспомнить: "Все свободы свободны одинаково, но некоторые свободнее других"). Иными словами — все свободы совместимы между собой, несовместимы они лишь с не-свободой.

Однако тут мы столь же плавно переходим к вопросу о лицензиях, что будет темой одного из следующих разделов. Перед этим же нужно вернуться к вопросу о гранях открытости.

Грани открытости

В начале этой главы я упомянул о так называемых открытых системах (Open Systems) — понятии, семантически близком к термину Open Source Software. И тем не менее, понятия эти — разные и в принципе между собой не связаны. Потому что под открытой системой понимается просто-напросто "система, которая состоит из компонентов, взаимодействующих друг с другом через стандартные интерфейсы" (Жан Мишель Корну, один из авторов руководства Французской ассоциации пользователей UNIX; цитировано по работе С. Д. Кузнецова "Операционная система UNIX" на **http://www.citforum.ru/operating_systems/unix/contents.shtml**).

Что же такое стандартные интерфейсы? Это некоторые соглашения, которые определяют функциональность компонентов системы. Например, стандарты на командные оболочки описывают некоторую минимальную функциональность программ этого класса, стандарты на системные вызовы ядра — то, что должны делать эти системные вызовы (например, открывать файл или читать его содержимое и т. д. При этом на реализацию функций того или иного компонента системы никаких ограничений не накладывается.

Стандартные интерфейсы для открытых систем регламентируются набором соглашений POSIX (Portable Operating System Interface), разрабатываемых

двумя организациями: IEEE (Institute of Electrical and Electronics Engineers Inc., **http://www.ieee.org/portal/site**) и Open Group (**http://www.opengroup.org**). Соглашения эти, получившие (с легкой руки все того же RMS) звучный титул стандартов POSIX, опирались, в первую очередь, на опыт разработки систем UNIX. И следовательно, все UNIX-подобные системы, как проприетарные (Solaris, например, или AIX), так и свободные (Linux или FreeBSD) по определению являются POSIX-совместимыми, представляя собой конкретные реализации соответствующих стандартов (подробнее о них будет говориться в следующей главе).

Собственно, термины "UNIX-подобная ОС" и "POSIX-совместимая ОС" можно было бы рассматривать в качестве синонимов. И на протяжении всего данного сочинения я так и делаю. Тут, однако, нужно иметь в виду несколько обстоятельств.

Во-первых, термин "UNIX" представляет собой зарегистрированную торговую марку. И продолжающееся по сей день крестоносное сутяжничество SCO против IBM (и попутно — против всего движения Open Source и Free Software), исход которого остается неясным, в немалой степени будет способствовать его дискредитации. Имея в своем осадке лишь тот положительный момент, что в очередной раз привлекло внимание к понятию Open Systems как системам, следующим открытым (т. е. общедоступным) стандартам. Достаточно вспомнить попытку демонстрации "украденного" из UNIX кода в Linux-ядре.

Во-вторых же, и главных, понятие POSIX-совместимости, строго говоря, выходит за рамки UNIX-подобия, т. е. сходства с неким первозданным UNIX. В той или иной мере соответствие POSIX-стандартам (хотя и неполное) признается разработчиками ОС QNX, генетически с UNIX никак не связанной. Да и Linux представляет собой попытку воспроизведения функциональности UNIX с "чистого листа", не только без использования кода, но и без доступа к нему. При этом Линус Торвальдс опирался не столько на устройство самой системы UNIX, сколько именно на стандарты POSIX. Впрочем, подробнее об этом будет говориться в следующей главе.

Следует заметить, что и для Windows линии NT/2000/XP декларируется соответствие стандартам POSIX. Однако, как это в обычае у фирмы — разработчика этих продуктов, стандарты эти понимаются тут весьма своеобразно и трактуются весьма расширительно в плане "улучшения". А потому отнесение Windows к POSIX-совместимым системам, по меньшей мере, спорно.

Все стандарты POSIX (а в это семейство входит несколько групп соглашений, например, стандарты на интерфейс прикладных программ, утилит и оболочек и т. д.) являются открытыми в понимании, близком к Open Source. То есть они общедоступны — (почти) любой "человек с улицы" может полу-

чить к ним доступ и создать в соответствии с ними свою открытую систему (откуда и пошел термин Open Systems).

Однако открытость основополагающих стандартов ни в коей мере не подразумевает открытости созданных в соответствии с ними операционных систем. И само по себе соответствие POSIX-стандартам не влечет их свободного (тем более — бесплатного) распространения. Кратко говоря, под Open Systems, в отличие от Open Source, можно понимать просто системы, основанные на открытых (общедоступных) стандартах. А уж на каких условиях эти системы распространяются — определяется лицензиями, которые вольно было приписать им разработчиками.

Кое-что о лицензиях

В рамках настоящего сочинения нас будут интересовать только лицензии, относимые к классу свободных (не обязательно в понимании Free Software Foundation). И здесь нужно начать с того, что лицензий этих — немалое количество: ведь только все диктатуры похожи друг на друга, тогда как свобода реализуется каждым по-своему.

Наиболее известна General Public License (сокращенно GPL; оставим название без перевода, поскольку смысл английского оригинала интуитивно более понятен, чем любой отечественный эквивалент). Созданная в первоначальном своем варианте Столлменом (при участии профессиональных юристов), она ныне принята для Free Software, разрабатываемого в рамках проекта GNU и при участии FSF. Кроме того, по ее условиям распространяется ядро Linux и множество программ независимых разработчиков.

Каждый желающий может ознакомиться с текстом GPL на сайте FSF — на английском, правда, языке, да еще и юридическом английском американском (**http://www.fsf.org/licensing/licenses/gpl.html**). Правда, ее русские переводы есть на **http://www.citforum.ru/operating_systems/articles/gpl_rus.shtml** и во множестве других мест, перечислять которые было бы долго. Ни один из русских переводов GPL не имеет статуса официального (как, впрочем, и ни один из нерусских), но наибольшим авторитетом обладает перевод Елены Тяпкиной (указанная ранее ссылка, а также **http://www.infolex.narod.ru/gpl_gnu/gplrus.html**).

В двух словах суть GPL повторяет, понятное дело, общие принципы свободы Столлмена, с одним важным дополнением: любое программное обеспечение, разработанное на базе GPL-софта, может распространяться только в сопровождении открытых исходных текстов и под той же лицензией. То есть разработчик, включивший в свое творение хоть строку кода, подпадающего под GPL, не имеет права распространять его под какой-либо другой лицензией. Хотя, повторю еще раз, никаких иных ограничений, ни в

плане взимания денег, ни какого-либо иного коммерческого использования — на него не накладывается. А под сопровождением открытыми исходниками подразумевается их физическая доступность: на твердых ли носителях, на публичных FTP-серверах и т. д.

Лицензия GPL вызывает наибольшие споры в мире Open Source, от почитания единственно пригодной для распространения свободного софта до активного неприятия, вплоть до появления термина "вирус GPL". Однако мы в эти споры вдаваться не будем, отметим только, что она являет собой объективную реальность, и что под ней распространяется огромное количество программ: от утилит GNU-проекта до таких крупных продуктов, как ядро Linux, GIMP, GNOME и множество других. В общем, вероятно, суммарно много большее, чем под всеми прочими свободными лицензиями, вместе взятыми.

Вследствие уже отмеченной аберрации FSF не просто тесно ассоциируется с самим понятием Free Software, но подчас, осознанно или неосознанно, с ним отождествляется. Однако это не так, и GPL — далеко не единственная лицензия, под которой свободный софт распространяется. Ибо на другом полюсе свободы софта располагается BSD-лицензия, впервые сформулированная в университете Беркли для распространения системы, именовавшейся в те годы BSD UNIX. Ныне под этой лицензией распространяются такие ОС, как Free-, NET- и OpenBSD, все их производные (типа DragonFly), а также изрядное количество программ независимых разработчиков.

Лицензия BSD (точнее, модифицированная лицензия BSD — иногда говорят о лицензиях BSD-стиля, различающихся в деталях, но единых в главном) столь же строго следует принципам свободного софта, как и GPL. Ей-богу, ничего противоречащего сути принципов Столлмена в ней не найти — впрочем, она очень короткая, в чем легко убедиться, ознакомившись с ее текстом, правда, опять-таки, по-английски (**http://www.xfree86.org/3.3.6/COPYRIGHT2.html#5**). То есть все подпадающие под нее программы могут свободно использоваться, модифицироваться и распространяться. Главное отличие ее от GPL: BSD-лицензия не обязывает к непременному свободному распространению продуктов, разработанных на ее основе. То есть закрыть код свободных программ, защищаемых BSD-лицензией, не вправе никто. Однако собственные разработки, на них базирующиеся, вполне могут распространяться как любой коммерческий софт, не только за деньги, но и без исходных текстов.

Чтобы понять, как это выглядит в реальности, рассмотрим наиболее показательный пример. Как известно, Mac OS X основана на микроядре Mach (разработанном в университете Карнеги-Меллона и распространяемом под собственной лицензией BSD-стиля) и системном окружении, заимствованном из FreeBSD, что в совокупности и дает ее ядро Darwin. Поскольку и ядро Mach, и его системное окружение суть свободные программы, то и

Darwin имеет тот же статус, распространяясь открыто и свободно: его можно скачать и пользовать из командной строки (или в сочетании со свободными же реализациями Иксов) в свое удовольствие. Однако графический интерфейс пользователя и прочие прибамбасы, которые и определяют своеобразие Mac OS X, являют собой собственные разработки Apple, и об их свободном распространении речи не идет.

Из прочих свободных лицензий известны: лицензия X-консорциума, близкая к ней — Массачуссетского технологического института (MIT) и лицензия, под которой распространяется Mozilla, а также серия "университетских" лицензий. Список их приведен на соотвтетсвующей странице сайта проекта FSF (**http://www.fsf.org/licensing/licenses/index_html#GPLCompatibleLicenses**), представление о содержании можно получить из обзора Елены Тяпкиной (Сравнительный анализ основных лицензий Open Source: GPL, LGPL, BSD, MIT, Mozilla public license, Apache software license — **http://www.libertarium.ru/libertarium/18586**). Все они, приближаясь либо к GPL, либо к BSD-стилю, в полной мере отвечают принципам Столлмена. Не случайно самые строгие пуристы от FSF именуют их GPL-совместимыми. Что на практике означает возможность использования софта под этими лицензиями в рамках единых проектов.

Таким образом, многоликость явления Free Software находит отражение в многообразии лицензий, под которыми свободный софт распространяется. Что, однако, обычно не мешает им мирно уживаться друг с другом: так, базовый комплект FreeBSD (так называемый Distributions), подпадающий в основном под BSD-лицензию, включает в себя ряд компонентов проекта GNU, защищаемых лицензией GPL и называемых Contributions (в их числе такой важный, как компилятор gcc). А базовая система Linux, бóльшая часть которой представлена GPL-программами, включает, например, библиотеку zlib, распространяемую на условиях лицензии BSD-стиля.

Более того, свободные лицензии вполне могут сосуществовать и с коммерческими, ярким примером чему является интегрированная среда KDE. Она основана на библиотеке Qt, которая сама по себе является коммерческой (и стоит, надо заметить, немалых денег). Однако для некоммерческого использования она бесплатна и как бы свободна, распространяясь на условиях близкой к GPL лицензии — QPL. Не возьмусь судить о том, как это реализовано юридически, однако факт остается фактом: на библиотеке Qt разрабатываются и такие проприетарные программы, как браузер Opera (а это, несмотря на бесплатность при некоторых условиях, все же коммерческий продукт) или Safari — штатный браузер коммерческой Mac OS X, и KDE с ее приложениями, к чистоте и свободе лицензии которой ныне не осталось претензий у самых ярых апологетов FSF.

Следует заметить, что пуристы от FSF выделяют среди свободных лицензий еще и группу GPL-несовместимых. Самыми яркими их примерами являются

исходная лицензия GPL, содержавшая ранее так называемую оговорку о рекламе (требование упоминания имени Калифорнийского университета Беркли), и лицензия, под которой, начиная с версии 4.4, распространяется XFree86 (требующая аналогичного упоминания системы в продуктах, ее использующих). Учитывая, что система BSD действительно была разработана в Калифорнийском университете Беркли, а XFree86 — единственная открытая графическая система для всего POSIX-семейства (Xorg, распространяемый по лицензии GPL — не более, чем ее порождение), я лично не вижу в этом никакого бесчинства. Должен сказать, что я не одинок в своем недоумении. Так, разработчики NetBSD считают лицензию XFree86 нормальной лицензией BSD-стиля и продолжают использовать эту систему в качестве графической среды своей ОС. А Патрик Фолькердинг, разработчик дистрибутива Slackware Linux, так и заявил, что переходит на Xorg исключительно в целях совместимости с другими дистрибутивами, но никак не из лицензионных соображений.

Вернемся, однако, к свободным лицензиям. Как уже было сказано, все их многообразие укладывается на линию GPL-BSD. Первая однозначно требует, чтобы все программы, включающие в себя свободный код, оставались свободными всегда. Вторая же столь же однозначно разрешает закрытое распространение собственных разработок, пусть даже и основанных на свободном коде.

Я не буду вдаваться в дискуссию о том, какая из свободных лицензий больше отвечает идеалам свободы и демократии. Более интересным видится мне вопрос: а откуда вообще взялось понятие свободного софта?

Истоки Free Software

В решении этого вопроса нам поможет одно интересное наблюдение: большинство "университетских" лицензий по своей сути тяготеют к формулировкам BSD-стиля. Почему?

Ответить на этот вопрос нетрудно. Дело в том, что Ричард Столлмен, формулируя (в первой половине 80-х годов прошлого века) свои принципы свободы софта, явным или неявным образом, но — изобретал велосипед. Ибо принципы его суть не что иное, как условия, на которых испокон веков распространялись результаты любых научных исследований (к слову сказать, и Ричард деятельность свою начинал как сотрудник вполне академической организации — лаборатории искусственного интеллекта MIT). Каковые, по определению, должны быть общедоступны, открыты в отношении методики, и которые можно использовать любым образом (за исключением приписывания себе их авторства и ограничения доступа к ним других).

Использовать — в том числе и для коммерческих продуктов, реализуемых через вполне закрытые технологии. Согласитесь, было бы нелепым требовать от создателей дизеля или двигателя внутреннего сгорания полной открытости их результатов на том основании, что они базируются на общедоступных законах термодинамики. А ведь именно это и констатирует BSD-лицензия, имеющая сугубо академическое происхождение.

Не случайно первые работы над проектом BSD, полностью вписывающиеся в идеологию Free Software, начались задолго (лет за 10) до того, как Столлмен сформулировал свои знаменитые принципы в 1984 г.

Аналогию между научными разработками и Free Software можно углубить. Что является неотъемлемым атрибутом профессионально выполненной научной работы? Атрибутом этим является научный аппарат. То есть для наук экспериментальных — публикация методов измерений, позволяющих независимое воспроизведение результатов; для наук феноменологических типа той же геологии — описания полевых наблюдений; для истории — ссыного аппарата (дающего читателю теоретическую возможность порыться в первоисточниках).

Именно научный аппарат делает возможным проверку и воспроизведение выводов исследователя профессионалом в той же области науки. А "смежнику", не владеющему деталями материала по конкретной проблеме, позволяет безбоязненно ссылаться на полученные другими результаты, в небезосновательном расчете на то, что ошибки будут выявлены специалистами соответствующего профиля.

Разумеется, наличие научного аппарата не гарантирует высокого качества научной работы. Однако оно позволяет выполнить (хотя бы в первом приближении) ту самую публичную экспертизу, за позитивными примерами которой далеко ходить не нужно — имя им легион.

Ту же роль, что научный аппарат в исследованиях, в области Free Software выполняют открытые исходники. Их наличие само по себе не гарантирует высокого качества программы. И, разумеется, пользователю, далекому от программирования, исходные тексты на Си мало чего скажут. Однако, как и в науке, пользователь вправе ожидать, что ошибки в открытой программе будут выявлены людьми, разбирающимися в этом получше него. Не только выявлены, но и исправлены, а не возведены в ранг непременной особенности данной программы. А он, пользователь, вследствие открытого характера разработки об этом узнает своевременно — при наличии желания, разумеется. И, надо сказать, все эти ожидания в большинстве своем оправдываются...

Собственно говоря, именно публичная экспертиза кода (или, проще говоря, механизм обратной связи пользователей и разработчиков) — одна из причин, почему открытость исходников (т. е. явление Open Source) почти неизбежно влечет за собой свободу распространения программы в соответствии с

принципами Столлмена. Действительно, какой смысл имеет выявление ошибок (свобода изучения) без возможности их исправления (свободы модификации) и предоставления доступа к скорректированной версии (свободы распространения)?

Показательно, что все успешные проекты, некогда коммерческие, но подвергшиеся процедуре открытия исходников на определенных, не вполне соответствующих GPL- или BSD-лицензиям, условиях, рано или поздно эволюционировали в сторону полной свободы в понимании Free Software. Здесь достаточно вспомнить StarOffice, Netscape или GRASS. Причем характерно, что на базе одного и того же открытого кода вполне мирно уживаются как полностью свободные программы (OpenOffice.org, Mozilla или многочисленные Gecko-клоны), так и программы, сохранившие свой проприетарный статус (первозданные StarOffice и Netscape). Программы же с открытыми исходниками, не ставшие истинно свободными (например, Solaris или True64 UNIX), очень быстро потеряли и свой открытый статус (хотя с Solaris, возможно, опять произойдет очередная метаморфоза).

Кто оплачивает банкет

В контексте этого раздела осталось обсудить одну тему, близко связанную с предыдущей: а кто вообще должен (и должен ли?) финансировать такие родственные области человеческой деятельности, как разработка Open Source и фундаментальная наука? И кто их финансирует в настоящее время?

Должно, однако, для начала заметить, что родственность Open Source и фундаментальной науки — не мое изобретение. Блестящее обоснование этого тезиса содержится в статье Николая Безрукова "Разработка программ с открытыми исходниками как особый вид научных исследований" (**http://www.citforum.ru/open_source/academ**).

У читателя большей части популярных публикаций на тему Open Source вполне может возникнуть впечатление, что разработка программ этого класса осуществляется на голом энтузиазме, напоминающем ударников-стахановцев. Что, конечно, очень благородно, но в наш век вызывает законное недоверие.

Проблема финансирования Open Source включает в себя три аспекта. Кто может быть заинтересован (и может ли?) в финансировании разработки Open Source. В какой форме заинтересованные стороны могут осуществлять финансирование разработки Open Source. Каким образом финансирование это может распределяться среди разработчиков.

Должен сразу признаться: ответов на эти вопросы я не знаю. Однако они совершенно те же, что возникают при финансировании фундаменталь-

ной науки. А поскольку в этой сфере мировым научным сообществом накоплен весьма большой, хотя и не всегда положительный, на мой взгляд, опыт, по аналогии рискну высказать некоторые соображения.

Но сначала зададимся иным вопросом: а нужно ли финансирование разработки программ с открытыми исходниками вообще? Ведь теоретически предполагается, что это предприятие самоокупаемое, а то и просто прибыльное. Однако теоретически, как говаривал дед Щукарь, она лошадь, а практически она падает...

Напомню, что на протяжении всей этой главы я пытался подчеркнуть, что открытость исходников некоей программы отнюдь не подразумевает бесплатности. Тем не менее, практика такова, что подавляющее большинство открытых и свободных ОС распространяется условно бесплатно. То есть интеллектуальная собственность источником дохода служить не может (как это имеет место быть в случае проприетарного софта).

Прибыльность разработки Open Source обычно обосновывается туманными фразами о сопровождении и поддержке программных продуктов, распространяемых бесплатно. В смысле — по себестоимости носителей и документации.

Тем не менее, достоверных известий о фирмах или персонах, обогатившихся за счет массовой техподдержки Open Source, у меня нет. Что понятно: люди, покупающие дистрибутив Linux (для примера), имеют целью разобраться в системе. И либо этой цели достигают, и тогда в техподдержке не нуждаются. Либо бросают это занятие, и тогда не нуждаются в поддержке тем более. Если же речь идет о корпоративных пользователях, не думаю, что разумный руководитель рискнет перевести весь свой документооборот на Linux или BSD, не имея в штате квалифицированного специалиста по одной из этих систем. Так что мечты о заработке на техподдержке столь же наивны, как и мечты о самоокупаемости фундаментальной науки, блестяще-утопично сформулированные на заре перестройки Максимом Максимовым в статье "Реанимация" (журнал "Знание — сила", 1989 г., № 11).

Конечно, само по себе издание и распространение дистрибутивов прибыль приносить может. Так же, как ее приносит, скажем, издание детективов и их продажа. Однако к аналогии с книжным бизнесом я обращусь несколько позднее.

А пока: кто заинтересован в развитии Open Source и, соответственно, мог бы финансировать его разработку? Кроме его разработчиков и пользователей, разумеется. Но ведь разработчики Open Source, хотя и заняты своим делом в большой степени из любви к искусству, также хотят есть-пить. А пользователи ведь пользуют Open Source в значительной мере ввиду его бесплатности, практической или теоретической. Что удовлетворению означенной потребности разработчиков отнюдь не способствует...

Первая сторона, заинтересованная в развитии Open Source, — пользователи любого коммерческого софта. Впрочем, несмотря на внешнюю парадоксальность, это очевидно: только конкуренция со стороны открытых и свободных программ может подвигнуть коммерческих разработчиков на совершенствование своей продукции.

Как ни странно, в качестве второй стороны, наиболее заинтересованной в развитии открытого софта, видятся производители коммерческих UNIX-систем (хотя, подозреваю, что сами они своего счастья не всегда понимают). Почему — обосновать не трудно.

Молодому человеку, привыкшему к Linux (FreeBSD, OpenBSD и т. д.) и пришедшему на службу со школьной (или университетской) скамьи, гораздо легче перейти на Solaris или AIX. А достигнув по службе должного положения, он, безусловно, приложит все усилия к тому, чтобы внедрить POSIX-системы в родном трудовом коллективе, и по тем или иным причинам далеко не всегда можно обойтись свободными их реализациями, придется обращаться к коммерческим.

За иллюстрацией этого достаточно спуститься в не столь уж далекое прошлое — 1995 год. Когда (не заставшим того времени поверить в это трудно, но факт имел место быть) в качестве реальной альтернативы для массовых настольных систем рассматривались OS/2 и Windows 95. Причем ни у кого не вызывало сомнений технологическое превосходство первой. Но Windows 95 пришла в дома, а OS/2 — нет (о причинах распространяться здесь неуместно). И через несколько лет на работу вышло поколение, вскормленное и вспоенное в "окнах". Результат не замедлил последовать: кто и когда последний раз видел OS/2 на пользовательском десктопе?

Наконец, третья заинтересованная сторона — это государство, причем любое (правда, применительно к нашему государству об этом и не подумаешь). Причины этого также достаточно тривиальны: здесь и баланс между монополизацией и свободной конкуренцией, и массовый независимый аудит программной продукции, и снижение себестоимости рабочего места госчиновника... список легко продолжить. И не на последнем месте в нем окажутся соображения государственной безопасности: как бы "подоконная" кольчужка коротка не оказалась...

Теперь посмотрим, горят ли заинтересованные стороны желанием оказать развитию Open Source посильную финансовую помощь.

Конечно, странно было бы ожидать от массы Windows-пользователей благотворительных пожертвований в FSF и аналогичные организации (это нередко существенный источник финансирования, примером тому проект OpenBSD). Тем не менее, наиболее продвинутая часть их косвенно в финансировании Open Source участвует. Хотя бы тем, что проявляет интерес к публикациям на эту тему, и бумажным, и сетевым. В результате компьютерные издания печатают больше статей соответствующего профиля, регулярно выплачивая

авторам гонорары, чем и способствуют материальному благополучию писателей-POSIX'ивистов (навроде вашего покорного слуги).

Все это я говорю к тому, что романтическое представление о создателях свободного софта как об энтузиастах-одиночках (со всеми вытекающими из этого следствиями, как положительными, так и отрицательными) подчас весьма далеко от истины. Многие из них — и сотрудники коммерческих фирм, и выходцы из университетской среды — занимаются этим делом в рамках своих прямых должностных обязанностей, получая за это зарплату.

Значит ли это, что движение Open Source имеет достаточную финансовую базу? Отнюдь. Государственные программы имеют обыкновение рано или поздно заканчиваться по самым разным причинам. Так, не с падением ли мировой системы социализма связано прекращение финансирования проекта BSD в 1991 г.? Вливания коммерческих фирм зависят от конъюнктуры рынка. А интерес широких народных масс ко всему, связанному с Linux и Open Source, в значительной мере лишь дань моде.

Так что какие-либо дополнительные источники финансирования не помешали бы движению Open Source. Однако уже упоминавшаяся аналогия между ним и фундаментальной наукой наводит меня на мысль: а пойдут ли они ему на пользу? Ибо второй вопрос после получения финансирования: как это финансирование будет распределяться среди собственно разработчиков?

Можно ли заработать на Open Source

И здесь мне хотелось бы начать с цитаты из статьи Владимира Попова "Размышления на тему: заработать на Open Source" (**http://posix.ru/openway/opensource_popov1**):

"Критерий рентабельности заведомо не приложим к научной, гуманитарной, медицинской и многим другим сферам деятельности человека (не говоря уж о сфере искусства)".

Возникает вопрос — а приложим ли этот критерий собственно к открытому программному обеспечению? Что особенно актуально в свете все более частых попыток заработать на Open Source. Конечно, само слово "заработать" в этом контексте имеет двойной смысл, но к этому я еще вернусь в конце. А пока все же попробую ответить на свой вопрос. Несколько забегая вперед, придется обратиться к истории (каковая послужит предметом следующей главы).

С чего началась дорога, получившая потом название UNIX way? Как всем известно, началась она с проекта Multics. Что это было такое? Это была совместная разработка Bell Labs, General Electric и Массачусетского техноло-

гического института (MIT). То есть вполне академический проект. Где работали основоположники UNIX? Работали они в той же Bell Labs, хотя и принадлежавшей коммерческой компании, но имевшей целью все же научные разработки. А кто сказал, что научная работа не может финансироваться частным капиталом? Только не те, кому посчастливилось получить Нобелевскую премию.

Из сказанного можно сделать вывод: создание UNIX как открытой (в смысле, соответствующей открытым стандартам) системы и ее производных, представленных открытым и свободным софтом во всех его проявлениях, происходило практически полностью в сугубо академической среде. А если вспомнить о том, что вся софтверная индустрия базируется, в сущности, на математических алгоритмах, развивавшихся в рамках чистой математики со времен Евклида и Бируни, становится окончательно ясно: UNIX way и Open Source есть порождение той области человеческой деятельности, которую именуют чистой, или фундаментальной наукой. Впрочем, задолго до меня к тому же выводу пришел Николай Безруков в упоминавшейся мной статье.

Таким образом, поставленный ранее вопрос сводится к более общему: а можно ли заработать на фундаментальной науке? И тут впору вспомнить о двух смыслах русского слова заработать. Первый — это получать некоторую плату (в Кодексе законов о труде при советской власти она так и называлась — заработная, хотя далее мы увидим, что это определение не вполне адекватно) за свою работу. И второй — это извлечение прибыли, т. е. предпринимательская деятельность.

Если в наш вопрос подставить первый смысл слова "заработать", то ответ на него будет сугубо положительным. В обоснование чего можно привести не только немецких профессоров XIX века (весьма обеспеченных по тем временам людей), но и судьбу Линуса Торвальдса: ведь в сущности на протяжении нескольких лет он получал свою зарплату в университете именно за то, что разрабатывал Linux.

За чей счет выплачивается зарплата научного работника? Как стало, надеюсь, ясным из предыдущего раздела: научная работа вообще оплачивается обществом в целом. А уж посредством чего и кого эта оплата осуществляется — вопрос другой.

Мы уже видели, что посредником в оплате научной работы может быть государство в целом или отдельные государственные ведомства. Это могут быть корпорации, превосходящие по масштабам несредние даже государства. И, наконец, общество (или отдельные, наиболее прогрессивные, его представители) могут выступать финансистами научных работ и непосредственно — в виде пожертвований.

Выработанные за века существования науки как важной сферы человеческой деятельности (и за многие уже десятилетия — как массовой профессии) три формы распределения финансовых источников: прямая выплата так называемой зарплаты (так называемой, потому что суть явления лучше передается словами "жалование" или "оклад содержания"), финансирование в форме безвозмездных кредитов — грантов и гонорарная оплата результата имеют свои достоинства и недостатки. Однако никакого иного механизма пока не придумано и в мире свободного софта.

Так что зарабатывать на кусок хлеба своей работой на ниве Open Source вполне можно (правда, боюсь, не в нашей стране). А вот может ли быть открытый софт объектом предпринимательской деятельности? То есть служить для извлечения прибыли. Здесь однозначный ответ дать трудно.

По аналогии с научной работой можно видеть, что сама по себе разработка открытого софта никакой прибыли принести не в состоянии: что-то я не слышал об акционерных компаниях по разработке специальной теории относительности или квантовой механики. Прибыль начинается на стадии так называемого внедрения научных разработок в жизнь, т. е. когда наука перестает быть наукой и становится технологией. В софтверной индустрии с таким внедрением можно сопоставить тиражирование, обучение и сопровождение программной продукции. Как тут обстоит дело с прибыльностью?

Очевидно, что норма прибыли при тиражировании свободных программ стремится к нулю. Ведь себестоимость носителей практически нулевая, сами программы общедоступны, и наварить здесь практически невозможно. Вспомним пример из книги Линуса Торвальдса: когда стоимость завоза воды превышает допустимый пользователями уровень, кто-нибудь обязательно протянет водопровод. Так что единственное, что тут остается — это пытаться компенсировать норму прибыли ее массой. То есть объемом продаж. А объем этот напрямую связан с количеством пользователей Open Source. Причем, в первую очередь, пользователей-индивидуалов: в силу специфики свободных лицензий компания уже в 100 человек вполне может наладить тиражирование своими силами и в потребных масштабах. Так что и места под солнцем для Linux-предпринимателей оказывается не так много.

И тут тиражирование тесно смыкается с обучением, поскольку существовать без него не может. А под обучением я понимаю не только (и даже не столько) сертифицированные курсы, сколько самую элементарную печатную документацию. Однако и здесь для Linux-компаний (назовем их так) не так уж все здорово. Ибо обычные книгоиздатели занимаются тиражированием уникальных, т. е. безальтернативных, авторских произведений (будь то научные монографии или детективные романы) или, по крайней мере, тех, которые воспринимаются обществом в этом качестве.

Сопровождающая же софт документация представляет собой лишь один из многих вариантов получения информации об оном софте. И с развитием Интернета роль ее все более снижается. Так что единственной побудительной причиной к приобретению коробки с руководством может быть только литературный талант написавшего ее автора: необходимую сумму знаний пользователь легко может получить другими путями.

Наконец, поддержка. Традиционные формы поддержки можно разбить на два вида: индивидуальное или в масштабе (рабочей) группы товарищей и корпоративная поддержка. Первая форма, блестяще зарекомендовавшая себя в мире Windows, в мире Open Source явно не сработает. Ибо начинающий пользователь Linux или очень быстро перестанет быть начинающим, или бросит это занятие, или получит в свое распоряжение идеально настроенную под его задачи систему, в которой можно будет, ничего не меняя, работать веками.

Корпоративная поддержка... Уже ограниченность списка по-настоящему удачливых компаний в этой сфере свидетельствует о том, что ниша эта не столь уж обширна, как кажется.

Это я не к тому, что поддержка Linux — дело бесперспективное. Хочу лишь подчеркнуть, что сфера эта столь же ограниченна, как и область тиражирования и обучения. А с распространением Linux к тому же будет все менее рентабельной.

Как же заработать на Open Source

Тут я постараюсь несколько сгладить мрачно апокалиптическую картину, нарисованную мною в предыдущем разделе, относительно перспектив коммерческого использования свободного софта. Но для этого нам придется вернуться к тому, кому и зачем нужен Linux (буду говорить так для краткости, но на самом деле все сказанное относится и к FreeBSD, и к другим BSD-системам).

Однажды на одном из форумов я затеял опрос: для чего пользователи переходят на Linux и прочие свободные ОС POSIX-семейства? И, как и ожидалось, смысл большей части ответов можно резюмировать так: чтобы получить надежную и устойчивую систему, идеально заточенную под конкретные задачи данного пользователя.

Другое дело, что задачи у всех бывают разные. Большинство обращается к Linux или для разработки софта, или для администрирования. Или для того, чтобы в домашних условиях учиться тому или другому. Некоторые авторы полагают, что только для этих целей свободные POSIX-системы и пригодны. Но при этом забывают еще об одной категории, самой многочислен-

ной, — так называемых простых, или конечных пользователях. А у них задачи еще более разнообразны. И если перспективы Linux в области игр или мультимедиа не вполне ясны, то как рабочий инструмент для очень многих и многих он оказывается идеальным.

И тут впору вспомнить, что любой дистрибутив Linux или BSD-система представляет собой лишь полуфабрикат индивидуальной системы для практического использования. И любой из них может быть превращен пользователем в закаленное и отточенное орудие для выполнения данной задачи, имеющее все необходимые функции — и ни одной лишней.

Реально ли это? Мой личный опыт показывает — более чем. Может ли это выполнить каждый отдельно взятый пользователь (не имеющий, напомню, специального образования и специфических навыков) на отдельно стоящем персональном компьютере? Теоретически, конечно же, может. Для этого только что и нужно, как прочитать несколько толстых книжек по Linux и особенно UNIX (объемом в какие-нибудь тысячу с небольшим страниц каждая), научиться сочинять простенькие шелл-скрипты и макросы для текстового редактора, ну и освоить еще несколько мелких дел.

Правда, возникает вопрос: а когда этот пользователь будет заниматься своей непосредственной работой? Ну, это его личное дело. Главная загвоздка тут в другом: в один далеко не прекрасный день такой пользователь с ужасом обнаруживает, что копаться в конфигах или разбираться с опциями компилятора для него стало интереснее, чем переводить контракты, подводить квартальные балансы или даже вычислять условия плавления базальтовых магм.

И в этот миг на свете станет еще одним переводчиком, бухгалтером, геологом меньше — и одним POSIX'ивистом больше. Что, само по себе, конечно же, замечательно, да вот только если так пойдет дело, кто же будет хотя бы начислять им зарплату? Да и за что, ведь вся их деятельность превратится в процесс обеспечения самих себя новыми задачами по настройке и совершенствованию собственной системы...

И вот тут наступает Час POSIX'ивиста. Именно его может позвать наш бухгалтер (переводчик, геолог) для оборудования своего рабочего места. Которое будет включать в себя не просто установку системы, а полный комплекс по ее обработке (рашпилем там или алмазным надфилем, это уже зависит от задач и субстрата, сиречь исходного дистрибутива). Причем в той степени, какая потребуется, чтобы избавить пользователя от необходимости приобретать хоть какие-то знания о внутреннем устройстве системы.

Возникает вопрос, а можно ли это выполнить в рамках UNIX-подобной системы? Ведь традиционно считается, что пользователь Linux должен читать горы руководств, разбираться в правах доступа и т. д. Отвечаю: именно в POSIX-системе такое возможно. Потому что обычно индивидуальный пользователь ее — не просто пользователь, но и сам себе администратор. То

есть он вынужден устанавливать и настраивать систему, устанавливать и обновлять прикладной софт и т. д.

Здесь же речь идет о создании некоего комплекса, монофункциональной системы с сознательно урезанными до необходимого уровня возможностями. Пользователь такой системы не должен в сущности даже иметь root-акаунта. Все, что от него требуется — это уметь включить питание, запустить пару-тройку приложений или утилит с требуемыми опциями (а создание скриптов, обеспечивающих такую возможность, — одна из задач нашего POSIX'ивиста) и выйти с сохранением данных и корректным завершением сеанса, а уж о корректности всего остального должен позаботиться POSIX'ивист.

И устанавливать программы такому пользователю не придется. Весь комплекс необходимого для его задач софта будет установлен единовременно. Обновления? А нужны ли они, если комплекс этот будет тщательно продуман изначально, подогнан как под задачи, так и под железо? Ведь классические программы в стиле UNIX way меняются мало (в смысле качества давно уже лучше некуда, а в смысле функциональности — в том-то и суть UNIX way, чтобы не прикручивать к утилите find системы заварки кофе). По-настоящему (не удовлетворения любопытства для) необходимость в обновлениях связана с обеспечением безопасности и появлением нового оборудования, не поддерживаемого наличным софтом. Но в данном случае ни то, ни другое не актуально: о безопасности можно позаботиться заранее, а оборудование не меняется до полной физической амортизации.

Будущее коммерческого использования свободных POSIX-систем не в дистрибуции Linux на десктоп каждой секретарши на смену Windows, дабы она могла им управлять, как кухарка государством. А в создании специализированных монофункциональных систем на базе некоего дистрибутива общего назначения.

О продолжении банкета

Руины прекративших свое развитие открытых проектов на просторах Интернета служат свидетельством недостаточного финансирования разработки свободного софта.

Впрочем, столь же частая причина прекращения открытых проектов — это просто потеря интереса к ним со стороны разработчиков. И последний фактор не столь уж трагичен, как может показаться. Более того, он следствие того положительного, что заложено в самой идеологии UNIX, и неотрывен от нее.

Как будет ясно из дальнейших глав, в основе идеологии UNIX лежит принцип разложения любой сложной задачи на несколько максимально простых, атомарных действий, интегрируемых с помощью специфических приемов. И потому все классические UNIX-утилиты (и большинство программ, следующих этому направлению, именуемому UNIX way) монофункциональны. То есть каждая из них умеет делать обычно только одно, но зато уж делает это очень хорошо.

А теперь представьте себе ситуацию: студент пишет маленькую, но полезную монофункциональную утилиту, например, для массированного переименования файлов. Причем один из основных его мотивов в этом случае — просто повышение собственной профессиональной квалификации. Постепенно утилита эта доводится до практически достижимого совершенства, и что дальше?

А дальше перед разработчиком есть два пути. Первый — это рутинное исправление мелких ошибок, внесение мелких усовершенствований, приведение в соответствие с новыми версиями ОС, системных библиотек и тому подобного взаимосвязанного софта. Но ведь наш молодой специалист за это время уже вырос в профессиональном отношении, у него могли появиться совсем другие интересы... И такое рутинное сопровождение программы для него оказывается просто скучным.

Второй путь — это расширение функциональности. Программу для переименования файлов можно научить эти файлы как-то еще обрабатывать. А там и работать с их содержимым, скажем. Ну и заодно уж подавать кофе в постель и сообщать прогноз погоды. В результате чего наше отточенное орудие переименования превращается в неповоротливого монстра, который все умеет делать... плохо.

И ладно бы, второй путь противоречил всей идеологии UNIX. Но ведь порочность его доказана практикой, всей историей проприетарного софта. Производители неизбежно вынуждены расширять, с позволения сказать, функциональность своих продуктов. Ведь как иначе убедить в необходимости покупки Office 2003 пользователя, которому зачастую вдоволь хватило бы возможностей Lexicon, как не открывающимся перед ним сказочным богатством функций? За которые приходится расплачиваться гигагерцами и гигабайтами и, в конечном счете, снижением общей производительности работы...

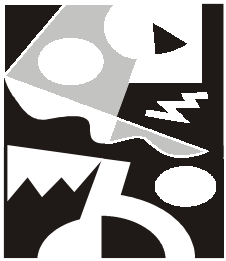
И потому тó, что разработчики свободных программ не идут по пути расширения функциональности своих программ, — фактор, безусловно, положительный. А тó, что в итоге они могут потерять интерес к дальнейшей разработке... Что ж, все в жизни имеет оборотную сторону, как сказал один

гражданин, у которого умерла теща, и пришлось раскошеливаться на похороны (Остап Бендер).

Тем более, что окончательно и бесповоротно открытые проекты умирают крайне редко. Все востребованные сообществом разработки так или иначе не только поддерживаются, но и развиваются в рамках заложенного в них потенциала. Доказательством чему судьба не только vim, существующего с доисторических времен и столь же долго (и беспрестанно) совершенствуемого, но и простенького joe: происхождение его теряется во мраке веков, имени первого разработчика никто уже и не припомнит, а новые версии его время от времени выходили на протяжении многих лет. А совсем недавно он получил новую жизнь и кардинально новые возможности...

Так что напрашивается и еще одна аналогия — между программами Open Source и эпическими произведениями прошлого, от "Илиады" до "Старшей Эдды": и тем, и другим уготована вечная жизнь в памяти людской. По крайней мере, до тех пор, пока они сохраняют актуальность для современного им общества. А после — после они становятся достоянием истории...

Глава 2



UNIX, Linux и BSD

- Что такое ОС.
- Что необходимо для ОС.
- Кое-что о стандартах.
- Кое-что о GNU, или не GNU ли Linux?
- Немного о дистрибутивах Linux.
- О FreeBSD сотоварищи.

Рассказ у нас пойдет в особенности о Linux и BSD. И любознательный читатель, надеюсь, многое узнает об устройстве этих ОС и их использовании.

Что такое ОС

Однако прежде чем переходить к сути дела, хорошо бы ответить на вопрос, а что же такое операционная система. Для чего следует рассмотреть существующие, явно или неявно, мнения на сей предмет.

А распространенных мнений по вопросу, что такое операционная система, два: минималистское и максималистское. Согласно первому, операционная система — это программа, именуемая ядром ОС. Применительно, например, к Linux под это определение подпадает только ядро. А все, что существует в составе любого Linux-дистрибутива помимо оного, суть системные утилиты и пользовательские приложения, к самой ОС отношения не имеющие.

Не менее важно и то, что на основе одного и того же (или сходного) ядра могут быть созданы разные операционные системы, которые все признают полноценными (и самостоятельными). Типичный пример — ядро Mach, на коем базируются или базировались и NextOS, и Mac OS X, и Darwin, и до недавнего времени Hurd, и безвременно оборвавшиеся Xmach и Yamitt. Вряд ли у кого повернется язык отнести все перечисленные имена к од-

ной операционной системе. Особенно показательно сравнение Mac OS X и OpenDarwin с их практически одинаковым ядром...

Если отвлечься от Linux и обратиться к BSD-миру, то в основе ядра и FreeBSD, и NetBSD, и OpenBSD, и коммерческой BSDi лежит одно и то же ядро 4BSD, вернее, его облегченная версия 4.4BSD-Lite. И хотя в дальнейшем все они развивались самостоятельно, но их взаимовлияние — факт неоспоримый: многие прогрессивные особенности, реализованные в FreeBSD пятой ветки, пришли из BSDi, иные же имеют источником проект NetBSD. Однако никому ведь не приходит в голову считать клоны BSD одной операционной системой (хотя, в подтверждение высказанного в предыдущем абзаце, с точки зрения пользователя разницы между ними меньше, чем между такими Linux-дистрибутивами, как Mandrake и Slackware).

Linux, правда, избег этой участи — сепарации по разновидностям ядра. Однако значит ли это, что ядро идентично во всех его дистрибутивах? Отнюдь. Конечно, любой (любой ли?) Linux-дистрибутив теоретически может функционировать на каноническом ядре (том, что берется с сайта **http://www.kernel.org**). Однако практически все крупные разработчики дистрибутивов достраивают свои ядра почем зря собственными разработками. Или включают в них достижения независимых творцов, о количестве которых можно получить представление, просмотрев на том же **http://www.kernel.org** каталог people. Однако на этом основании никто (за одним исключением, о котором я скажу далее) не утверждает, что, скажем, Red Hat и Gentoo — разные операционные системы.

Максималистская точка зрения на трактовку понятия "операционная система" последовательно проводится Microsoft с ее Windows любого рода. Согласно ей, ОС — это не только ядро, но и все его системное окружение, и графический интерфейс, и даже программы, которые испокон веков относились к категории пользовательских приложений (браузеры, например). Вспомним не столь уж давнюю тяжбу по поводу того, является ли Internet Explorer неотъемлемой частью MS Windows. А согласно сегодняшней политике MS, версий IE как отдельного приложения вообще более не будет: все его новые реализации жестко привязываются к грядущим реализациям собственно Windows. В том числе к механизмам безопасности, по всей видимости, встраиваемым в ядро (а где им еще быть?). То есть неявным образом утверждается, что ядро ОС и ее приложения едины, как народ и партия при советской власти...

Парадоксально, но такая же позиция поддерживается с крайнего фланга противоположной линии фронта, со стороны проектов GNU и Debian (ныне представленном в виде Debian GNU/Linux). Если обратиться к интервью со Столлменом на **http://www.citforum.ru/open_source/gnu_plans/**, то можно в явном виде встретить утверждения о том, что текстовый редактор emacs — часть операционной системы, и графический интерфейс (в дан-

ном случае оконная система X-X Window System) — часть операционной системы, и браузер — тоже часть операционной системы. Прямо по Биллу Гейтсу...

Некоторый резон во второй точке зрения имеется. Что особенно ясно видно на примере Windows. Во-первых, с теоретических позиций: вычленить ядро — задача нетривиальная (а в будущих версиях, похоже, и невозможная). А практически — что останется от Windows 95/98/ME (о линии NT/2000/XP ничего не скажу), если такое удастся? Не голый ли получится DOS? Хотя нужно заметить, что попытки освобождения ОС Windows от якобы неотъемлемых компонентов (того же Internet Explorer) предпринимались неоднократно, и попытки успешные.

Если же вернуться к примеру Linux, как наиболее показательному (и наиболее обсуждаемому), то суть точки зрения Столлмена и апологетов Debian сводится к тому, что именно системное окружение, механизм управления пакетами и прикладные программы, а также методика их подбора, тестирования и критерии качества (своего рода инфраструктура) и являют собой собственно операционную систему. А уж поверх какого ядра все это хозяйство функционирует — дело десятое. И иллюстрацией такой точки зрения является сам Debian (то самое обещанное исключение). Если обратиться к разделу портирования на **http://www.debian.org**, то там можно обнаружить информацию о портировании ОС (!) GNU Debian не только на ядро Hurd (исторически именно для него программы GNU и предназначались), но и на ядра NET- и FreeBSD.

Однако достаточно минутного размышления, чтобы сообразить: принятие второй точки зрения, в трактовке ли Microsoft или в интерпретации Столлмена и Debian Community, приводит к полному размыванию самого понятия операционной системы.

В сущности, за кадром высказывания Столлмена стоит утверждение, что в состав ОС GNU входит весь софт, распространяемый по лицензии GPL. Или даже весь софт с лицензиями, не вполне определенно называемыми GPL-совместимыми. Но в таком случае понятие операционной системы из технологического становится сугубо юридическим, ведь технологического способа определить степень совместимости лицензий до сих пор не придумано.

Что необходимо для ОС

Благо, наряду с двумя перечисленными точками зрения существует и третья. Формулировки ее в явном виде мне не встречалось (хотя в виде неявном, т. е. на практике, ее придерживаются разработчики систем BSD-клана). И потому возьму на себя смелость такую формулировку дать: операционная

система — это ядро и достаточный комплекс средств, необходимых для его функционирования на благо пользователя. То есть пользователь любой ОС должен иметь возможность, в первую очередь, устанавливать, не обращаясь ни к каким сторонним инструментам, необходимое ему программное обеспечение, запускать его и работать с ним. А теперь попробую дать развернутое обоснование этой позиции.

То, что любая ОС не способна функционировать без ядра — очевидно. Ядро, как и следует из названия, являет собой сердце любой операционной системы, отвечающее за взаимодействие пользовательских приложений (в самом широком смысле слова, включая средства администрирования) с аппаратурой компьютера. Однако с точки зрения пользователя это обычный исполняемый бинарный файл, функциональность которого определяется при конфигурировании, предшествующем сборке.

В функции ядра большинства POSIX-систем входят распределение процессорного времени между задачами, управление памятью (и процессом свопинга), взаимодействие с устройствами, доступ к файловым системам, обеспечение ввода/вывода данных, сетевая поддержка.

От всех остальных программ ядро отличается двумя важными особенностями. Во-первых, оно функционирует в отдельной области памяти, которая так и называется пространством ядра (kernelland), в которую пользовательские процессы доступа не имеют (обращаться, скажем, к устройствам ввода/вывода они должны посредством процедуры системных вызовов к соответствующим подсистемам ядра). Все прочие же программы располагаются в так называемом пользовательском пространстве памяти (userland).

Вторая особенность ядра как исполняемой программы в том, что, в отличие от всех других программ, оно всегда должно находиться в оперативной памяти физически. Ведь если часть ядра, управляющего виртуальной памятью, окажется выгруженной в раздел подкачки, система окажется без средства извлечения ее обратно. Пользовательское пространство остальных программ распределяется между физической оперативной памятью и областью свопинга. В сущности, у пользователя нет ни возможности, ни, скорее всего, необходимости определить, какую из частей единой виртуальной памяти программа использует в данный момент.

Из сказанного следует, что рост функциональности ядра с течением времени неизбежен для поддержки новых устройств, при сохранении обратной совместимости с устройствами старыми, новых файловых систем, сетевых протоколов и т. д. Что столь же неизбежно ведет к разрастанию ядра, расходу памяти и падению быстродействия.

Частично эту проблему можно решить индивидуальным конфигурированием ядра, описанным в *главе 14*. Оно позволяет отключить неиспользуемые в данной системе функции. Однако как быть с функциями, которые требуются время от времени? А то и вообще только могут потребоваться?

Общее решение этой проблемы было найдено в виде поддержки загружаемых модулей. Это фрагменты кода, обеспечивающие определенные функции ядра и функционирующие в его пространстве памяти. Но не перманентно, а загружаясь по мере необходимости: вручную, соответствующими командами или автоматически. И которые могут быть изъяты из памяти, когда в них минует надобность без перезагрузки системы — до следующего раза.

Соответственно этому, ядра могут быть разделены на монолитные (со встроенной поддержкой) и модульные. Впрочем, разделение это чисто теоретическое: во всех ядрах, в принципе поддерживающих модули (а это и Linux, и все BSD-системы), они в том или ином объеме используются. Чисто монолитные сборки ядра имеют смысл только для каких-то специальных задач.

Следующий шаг в том же направлении — так называемые микроядра, представителем которых является Mach. Их отличие в том, что опциональные фрагменты кода, например, драйверы устройств, не просто вынесены в отдельные модули, но и функционируют в пользовательском пространстве памяти. А собственно за ядром оставлены функции коммуникаций между ними.

Некогда микроядра виделись часто как база операционных систем будущего. Однако практические их реализации в большинстве случаев возлагаемых надежд не оправдали: их потенциальные достоинства (например, слабая зависимость от аппаратных платформ) с лихвой перекрывались недостатками (сложностью взаимодействия компонентов и, как следствие, низким быстродействием). И ныне микроядерная архитектура реализована в специализированной ОС QNX и Mac OS X. Однако в последнее время многие идеи, исходящие из проекта Mach, были реализованы в ядре DragonFlyBSD — ответвлении FreeBSD, ориентированном на работу в многопроцессорных системах.

Очевидно, что для работы ядро необходимо тем или иным способом загрузить. Это, как будет показано со временем, оно в принципе способно проделать собственными силами, т. е. загрузчик не оказывается неотъемлемой частью операционной системы. Однако практически жить без него настолько неудобно, что программа загрузки заслуживает чести быть стандартным компонентом ОС.

Далее, для функционирования системы недостаточно просто загрузить ядро, необходимо, чтобы оно запустило некий стартовый процесс. В UNIX-системах он имеет фиксированное название — init, за запуск которого отвечает одноименный исполняемый файл /sbin/init. Однако в реальных системах под этим именем могут выступать разные программы. И еще: в процессе загрузки следует выполнить некий комплекс мероприятий, определяемый набором сценариев, создающих пользовательское окружение. Так что сочетание инициирующей программы и стартовых скриптов вместе с загрузчиком — вторая из необходимых составляющих ОС.

Далее следуют утилиты поддержки функций ядра, обеспечивающие работу с устройствами, файловыми системами, сетевыми протоколами. Согласитесь, мало радости пользователю от поддержки ядром некоей файловой системы, если он не располагает инструментами для работы с ней. А отсюда третий непременный компонент, включающий средства обращения к физическим носителям, создания на них разделов и файловых систем, их проверки, монтирования и т. д. Если же ядро предусматривает поддержку нескольких файловых систем (как это имеет быть, например, для ядра Linux) в качестве родных (native), к каждой из них должен прилагаться свой комплекс обслуживающих утилит. При этом следует помнить: многие функции ядра могут быть реализованы как модули, и поэтому средства управления: загрузки, выгрузки, получения информации о модулях — также должны быть включены в эту группу.

Все перечисленные компоненты: ядро, загрузчик, средства инициализации и системные утилиты — необходимы для самообеспечения системы. Но ведь цель ее в конечном счете — выполнение пользовательских задач. В основе же любой пользовательской задачи лежит манипулирование пользовательскими данными, т. е. файлами. И поэтому соответствующий инструментарий для просмотра файловых систем, манипулирования файлами и их контентом, архивирования и компрессии (то, что можно назвать средствами боевого обеспечения) оказывается четвертым обязательным компонентом операционной системы. А поскольку файлы пользователя могут располагаться не только на локальной машине, сюда же примыкают и средства сетевого доступа (в том числе и доступа к Интернету).

Ядро, системные и пользовательские утилиты нуждаются в так называемых системных библиотеках, содержащих наборы программ для решения любой задачи (открытие, закрытие или сохранение файла, вывод результатов на экран и т. д.). Из них главной оказывается libc — библиотека функций языка Си (главного средства разработки во всех POSIX-совместимых системах). Однако не менее важны и некоторые другие библиотеки, например, свойств терминала. Это своего рода средства тылового обеспечения, практически незаметные, но, тем не менее, незаменимые. И потому они являют собой пятый обязательный компонент ОС.

Любые действия, как общесистемные, так и пользовательские, выполняются путем отдачи соответствующих командных директив. И потому интегрирующей надстройкой над всем описанным богатством выступает командная оболочка, она же — интерпретатор языка команд, по-простому — шелл (shell). Это шестой компонент ОС, роль которого невозможно переоценить: со временем мы увидим, что и система инициализации — лишь набор сценариев оболочки, и всякого рода приложения с интерфейсами — лишь надстройки над элементарными шелл-командами и их комбинациями.

И наконец, последний, седьмой из необходимых компонентов ОС — внутренняя система документации, дающая пользователю возможность изучения системы. Таковой испокон веков в UNIX выступает система man-страниц (Manual Pages). Несмотря на появление множества других форматов для представления документов, при всей своей архаичности она остается простым и универсальным средством оперативного получения исчерпывающей информации.

Таковы предельно минимальные требования к комплектации операционной системы. Именно реализация этой Великолепной Семерки компонентов обычно уникальна для каждой ОС и определяет ее своеобразие с точки зрения пользователя. Однако практически их оказывается недостаточно: для полнофункциональной системы необходимо еще два дополнительных компонента.

Первый — средства наращивания системы дополнительными программами, т. е. комплекс инструментов, объединяемых понятием пакетного менеджмента: установки, отслеживания и удовлетворения зависимостей, удаления. Эти действия могут выполняться вручную, посредством простой сборки программ из исходных текстов. И это универсальный метод, применимый в любой UNIX-подобной системе. В этом случае достаточно наличия компилятора для языка Си/Си++ и сопутствующего инструментария (линкера, ассемблера, средств ведения проекта). В свободных POSIX-системах такой инструментарий практически безальтернативен, включая пакеты gcc (компилятор) и binutils с сопутствующими утилитами типа make, automake, autoconfig. Они могут быть включены в операционную систему в качестве восьмого, но уже необязательного компонента.

Второй метод установки программ — компиляция по определенным правилам, освобождающим пользователя от необходимости отслеживать и разрешать зависимости. В этом случае компилятора и сопутствующего инструментария оказывается недостаточно, требуется еще и система автоматизации сборки, которая впервые появилась во FreeBSD под названием системы портов, и в дальнейшем ее аналоги широко распространились как в BSD-мире, так и в многих разновидностях Linux.

Третий метод распространения дополнительных программ — в виде бинарных (прекомпилированных) пакетов. Такие пакеты существуют во множестве различных форматов и, освобождая от необходимости в компиляторе и прочих средствах сборки, требуют для установки специального инструментария — пакетных менеджеров, — обеспечивающего, кроме собственно развертывания пакета, отслеживание и удовлетворение его зависимостей.

И порты, и системы пакетного менеджмента обычно специфичны не только для определенной среды, но даже отдельных ее разновидностей. В частности, именно пакетными менеджерами различаются между собой различные дистрибутивы Linux. С другой стороны, некоторые системы управления

портами и пакетами приобрели широкую популярность за пределами своих родительских операционных систем или дистрибутивов и стали фактически кроссплатформенными. И потому их следует относить не к базовым компонентам ОС, а к системам ее распространения (дистрибуции). Тем не менее, подчеркну, что наличие какой-либо системы управления пакетами (хотя бы ручной, в виде компилятора и сопутствующих утилит) абсолютно необходимо для функционирования любой ОС.

И последний из дополнительных (но опять-таки практически необходимых) компонентов ОС — средства для обеспечения работы в графическом режиме. Сам по себе UNIX и все его потомки и производные изначально графических средств внутри себя не имели. Эта функция возлагается на кроссплатформенную систему X Window System, именуемую также оконной системой X, или просто Иксами. Изначально не привязанная ни к аппаратной архитектуре, ни к какой-либо ОС, сама по себе она не имеет отношения ни к одному из дистрибутивов Linux, ни к BSD-семейству ОС, ни даже к UNIX вообще. Да и стандарты POSIX как будто бы ничего о ней не говорят: реализации Иксов регламентируются собственными стандартами, разрабатываемыми специальной организацией X-консорциумом (X Consorcium).

Тем не менее, Иксы в одной из их свободных реализаций для Intel-совместимых процессоров XFree86, разрабатываемой в рамках одноименного проекта (**http://www.xfree86.org**), или Xorg, разрабатываемой как порожденный проект (**http://x.org**), оказываются непременной частью всех открытых и свободных POSIX-систем.

Иксы заняли свое место в мире Open Sources и Free Software по одной очень важной причине — им фактически нет альтернативы в смысле доступа к графическим возможностям PC. И потому они оказываются практически неотъемлемой частью любой ОС POSIX-семейства, что Linux, что BSD. Они лежат как бы на грани между базовым комплектом этих систем и их опциональным обрамлением, выступая по отношению к приложениям графического режима примерно в том же качестве, что и базовые компоненты по отношению к утилитам и приложениям режима текстового. И об этом всегда нужно помнить. Как, впрочем, и о том, что Иксы (в виде ли XFree86 или в реализации Xorg) — это не Linux, не FreeBSD и не какая-либо другая ОС, это общее достояние POSIX-семейства.

И что мы получаем, если соберем вместе все перечисленное? А получаем мы практически то, что во FreeBSD охватывается понятием Distributions, вернее, той его частью, которая именуется Base и является единственным компонентом, обязательным при минимальной установке. Аналогичный базовый комплекс программ (так и называемый — Base) есть также в NetBSD и OpenBSD. Причем, если ядра в этих системах и различны, то программы системного обрамления чрезвычайно близки.

В Linux вычленить нечто подобное из многодискового дистрибутива несколько сложнее. Однако, напротив, можно прикрутить к его ядру некое подобие такой самодостаточной целостности. По аналогии с BSD, ее можно назвать Base Linux. Представление о составе можно получить, ознакомившись с LFS Book Герарда Бикманса — проектом, посвященным сборке собственной Linux-системы "с нуля" (**http://www.linuxfromscratch.org**).

Подведем итог столь длительных рассуждений. В состав любой POSIX-совместимой системы входит базовый набор из семи обязательных компонентов:

- ядро ОС;
- средства загрузки и инициализации системы;
- системные утилиты, обеспечивающие исполнение ядром его функций;
- минимальный набор пользовательских утилит;
- системные библиотеки;
- командная оболочка;
- система документации.

Они дополняются двумя опциональными (но практически столь же обязательными компонентами):

- набором средств установки пакетов;
- системой поддержки работы в графическом режиме.

Весь этот комплект программного обеспечения вполне резонно было бы назвать Base POSIX. И большая его часть описывается стандартами этого семейства.

Кое-что о стандартах

Из следующей главы можно будет узнать, что борьба за стандартизацию UNIX и его производных велась долго, упорно и безуспешно. Однако доля одного из сложившихся стандартов оказалась счастливой. И это тот самый POSIX, о котором столько говорилось, и объяснения которого предлагалось подождать до лучших времен. Настало время исполнить это обещание.

Набор соглашений POSIX был разработан международной организацией под названием IEEE. Термин Portable в названии первоначально означал, что соответствующая POSIX-спецификациям система может быть перенесена (возможно, с минимальными модификациями) на любую платформу, на машины с любой архитектурой, ведь тогда ОС UNIX-семейства функционировали не только на PC. Однако со временем не менее важным оказался

несколько другой аспект этого термина: любая прикладная программа, написанная в соответствии со стандартами POSIX, теоретически может быть перенесена на любую ОС POSIX-совместимого семейства. И нужно отметить, это один из тех редких случаев, когда теоретические ожидания блестяще подтвердились практикой.

Стандарты POSIX существуют в виде серии регулярно обновляемых документов (общим числом под два десятка), в которых описываются спецификации отдельных компонентов систем. Из них наибольший интерес для нас представляет документ IEEE Std 1003.1 **http://www.opengroup.org/onlinepubs/009695399/index.html**, в последней редакции которого (2004 Edition) были объединены ранее отдельные стандарты. В современном своем виде он включает четыре тома:

- Base Definitions volume (XBD) — определение терминов, концепций и интерфейсов, общих для всех томов данного стандарта;
- System Interfaces volume (XSH) — интерфейсы системного уровня и их привязка к языку Си, где описываются обязательные интерфейсы между прикладными программами и операционной системой, в частности, спецификации системных вызовов;
- Shell and Utilities volume (XCU) — определение стандартных интерфейсов командного интерпретатора (POSIX-shell), а также базовой функциональности UNIX-утилит;
- Rationale (Informative) volume (XRAT) — дополнительная, в том числе историческая, информация о стандарте.

Важно понимать, что стандарты POSIX жестко определяют базовую функциональность систем и приложений. Каковая, ради потери совместимости, и не должна изменяться. В то же время расширению функциональности они отнюдь не препятствуют. Так, любая из предусмотренных для POSIX-систем команда должна иметь определенный набор опций, предназначенных для выполнения базового набора действий. Однако никто не препятствует введению для данной команды дополнительных возможностей, реализуемых через опции, стандартом не предусмотренные. Важно только, чтобы первозданная функциональность команды оставалась в неприкосновенности.

Попробую пояснить на паре конкретных примеров. Так, в стандарте POSIX предусмотрена команда `split`, предназначенная для разделения текстового файла на отдельные фрагменты по размеру или числу строк. Это достигается указанием соответствующих опций, также описанных в стандарте (`-b` и `-l` соответственно). И именно это команда `split` обязана делать в любой системе, претендующей на звание POSIX-совместимой.

И, скажем, реализация команды `split` в Linux (вернее, в пакете coreutils проекта GNU, одном из компонентов Base Linux) выполняет свои обязанно-

сти буквально. А вот та же команда в BSD-реализации делает все то же, но имеет еще и дополнительную возможность разбиения текста на фрагменты по шаблонам (например, заголовкам). Для этого в ней введена опция `-p` (или `--pattern`), выходящая за рамки стандарта.

Еще более показательный пример расширения заложенных в стандарте возможностей при полном сохранении базовой функциональности — это командные оболочки, подробно рассмотренные в *главе 15*.

В стандарте POSIX предусмотрено, что оболочка POSIX-совместимых систем (еще раз прошу прощения) должна носить имя sh, располагаться в каталоге /bin корня файловой системы и выполнять совершенно определенный набор действий. Каковой, однако, показался многим разработчикам явно недостаточным, в результате чего были придуманы POSIX-совместимые оболочки bash и zsh, умеющие делать все то же, что и POSIX-shell, плюс многое (bash) или очень многое (zsh) другое.

Здесь приводятся лишь те сведения о стандартах POSIX, которые необходимы для понимания самого термина (и названия этой книги соответственно). Более подробную информацию можно получить из следующих документов:

- A Backgrounder on IEEE Std 1003.1, 2003 Edition — это общая информация о стандартах POSIX (**http://www.opengroup.org/austin/papers/backgrounder.html**);
- The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition — это описание самих стандартов POSIX (**http://www.opengroup.org/onlinepubs/009695399/index.html**);
- POSIX® 1003.1 Frequently Asked Questions (FAQ Version 1.10) — сборник часто задаваемых вопросов, имеющих отношение к стандарту (**http://www.opengroup.org/austin/papers/posix_faq.html**); доступен также русский его перевод (**http://www.nsdevice.narod.ru/sprog/posix_faq_rus.htm**).

А в онлайновом руководстве "Программирование в стандарте POSIX" В. А. Галатенко можно прочитать не только о титульной теме, но и о самих стандартах, причем на русском языке (**http://www.intuit.ru/department/se/pposix/**).

Кое-что о GNU, или не GNU ли Linux?

Между базовыми комплектами BSD-систем и Linux есть два важных различия, которые мы и рассмотрим последовательно.

Все системы BSD-клана на протяжении длительного времени разрабатывались именно как системные целостности (и что немаловажно, всегда срав-

нительно узкими коллективами разработчиков). Поэтому почти все составляющие их базовых комплектов — это неотъемлемые части этих ОС, созданные специально для них. Хотя есть и исключения, о чем я скажу чуть позже.

С Linux ситуация принципиально иная. Ибо большинство наиболее важных компонентов Base Linux, такие, как компилятор gcc, общесистемная библиотека glibc или командная оболочка bash, были разработаны до Linux, независимо от него и в рамках проекта GNU. Которому, по выражению Столлмена, не хватало только ядра для превращения в полноценную, противостоящую UNIX, операционную систему. И потому кажется, что термин GNU/Linux, на котором, применительно к этой ОС, настаивает Столлмен и FSF, имеет право на существование.

Однако лично мне он не нравится. Во-первых, он оскорбляет мой литературный вкус (так и предвижу появление дистрибутивов с названиями типа "Гнутикс"). Во-вторых, и это важнее, он не вполне правильный по существу. Ведь, если обратиться к истории (а в следующей главе мы это сделаем), можно увидеть, что не проект GNU ухватился за столь недостающее ему ядро. Напротив, это Линус Торвальдс для обеспечения работы своего ядра использовал отдельные компоненты из GNU-арсенала. В полном, к слову сказать, соответствии с духом и буквой GPL и движения FSF.

Так что заслуга построения Linux как цельной, самостоятельной и самодостаточной системы принадлежит Линусу. Ну и тому множеству разработчиков, которые приняли в этом участие, никто из них ведь не настаивал на том, чтобы в названии системы фигурировало его имя или название программы, которую он написал.

Кроме того, особенностью комплекса Base Linux (и это — второе важное его отличие от Distributions из FreeBSD и Base остальных BSD-систем) является его альтернативность. Причем даже для самых ключевых компонентов базовой системы: командной оболочки, главной системной библиотеки, средств работы с файловыми системами.

Так что альтернативность Base Linux — неотъемлемая черта этой операционной системы. И потому ОС Linux не только (а может быть, и не столько) ядро и набор базовых программ. Это, на мой взгляд, в первую очередь алгоритм для построения такого набора. Видимо, именно этим он и привлекает определенную категорию пользователей — возможностью соучастия в построении основы основ системы, недостижимую не то что в Windows, но даже в мире BSD с его камерным стилем разработки. Не зря дедушка русского линуксописания Владимир Водолазкий отметил в свое время, что быть просто пользователем Linux скучно. И действительно, рано или поздно любой линуксоид становится творцом, по крайней мере, в масштабах своей локальной машины. Правда, скорее всего, чаша сия не минует и пользователя BSD-системы...

Немного о дистрибутивах Linux

Термин "дистрибутив Linux" постоянно фигурировал ранее и столь же регулярно будет появляться и впредь, так что следует уделить некоторое внимание тому, что же он означает.

Надеюсь, что мне удалось убедить читателя в том, что комплекс программ, объединяемый понятием Base Linux, — это и есть операционная система Linux, достойная своего громкого имени. И в своем чистом виде способна не только обеспечить собственную загрузку, функционирование и наращивание, но и пригодна к решению ряда пользовательских, в том числе и довольно сложных задач. Однако далеко не все такие задачи могут быть решены средствами базового комплекса. А для иных, напротив, многие компоненты базового набора могут оказаться избыточными. И потому главное назначение описанного комплекса — служить фундаментом для построения систем, адекватных тем или иным пользовательским задачам. Именно такие системы, основанные на Base Linux, и представляют собой дистрибутивы этой системы.

В обиходе за такими адаптированными комплексами закрепился термин Distro, сопровождаемый, как правило, именем собственным вместе с родовым именем ОС. Примерами являются Red Hat Linux и Slackware Linux. Некоторые разработчики считают нужным в имени дистрибутива подчеркнуть GNU-происхождение большей части входящих в них программ. Отсюда появляются названия Debian GNU/Linux и подобные. А бывают и дистрибутивы (например, CRUX), в названии которых слово Linux вообще отсутствует , и это не значит, что они отвергают свою родовую принадлежность, просто разработчикам так показалось красивей.

Адаптация Base Linux под конкретные задачи осуществляется в двух направлениях. Основным является наращивание функциональности. Разработчики дистрибутивов добавляют в базовый комплекс дополнительные программные средства, предназначенные, например, для работы в графическом режиме вообще (оконная система X), программы, обеспечивающие графический пользовательский интерфейс (оконные менеджеры и интегрированные рабочие среды), инструментарий для работы с графическими и мультимедийными данными, пакеты для офисных работ. В результате образуется более или менее канонический набор программ, в прекомпилированном виде занимающий ныне обычно 3—5 дисков. Дистрибутивы такого типа и объема можно назвать полнофункциональными, или универсальными.

Второе направление — создание специализированных систем, служащих в качестве разного рода сетевых серверов, маршрутизаторов, брандмауэров и т. д. В них, напротив, часто можно видеть урезание базового комплекса Linux (понятно, что на машине, занимающейся только приемом и отправ-

кой почты, не нужен полный набор средств разработки). Дополняются они зато программами предназначения, соответствующего профилю системы.

Специализированные дистрибутивы чрезвычайно разнообразны. И отдельная отрасль среди них — так называемые дистрибутивы LiveCD. То есть системы, не нуждающиеся в установке на винчестер, но способные выполнять свои функции сразу же после загрузки с компакт-диска.

А функции их, нужно сказать, весьма разнообразны. Одни из LiveCD (наиболее показательный пример здесь — знаменитый Knoppix) предназначены для ознакомительных целей, представляя собой более или менее полное воспроизведение функциональности нормальной Linux-системы. Другие же монофункциональны и способны выполнять только какую-либо одну задачу. Примером тому служит MoviX, предназначенный исключительно для воспроизведения мультимедийных файлов (просмотр видео, прослушивание аудио), но зато уж делающий это очень хорошо.

Можно представить себе и еще одну разновидность специализированных дистрибутивов — пока эвентуальную, но с которой, как мне кажется, связано будущее применения Linux (и, возможно, BSD-систем) в сколько-нибудь широких масштабах. Это — АРМ (автоматизированные рабочие места) специалистов в областях, далеких от информационных технологий, от банковских операционистов до офисных делопроизводителей, монофункциональные системы, собранные и настроенные для выполнения одной задачи, но уже не административной (как многочисленные мини-дистрибутивы) или ознакомительной (как большинство LiveCD), но производственной. Ныне таковых практически не существует. Но упомянутый MoviX может рассматриваться как прототип таких АРМов, ибо является не чем иным, как рабочим местом потребителя мультимедийной продукции.

Далее речь пойдет только о полнофункциональных дистрибутивах общего назначения. Мир специализированных систем, во-первых, необъятен, а во-вторых, требует специфических знаний и навыков, которыми я не обладаю.

Так вот, полнофункциональные дистрибутивы отличаются друг от друга, по крайней мере, по одному из следующих критериев: комплектации, программе установки и (или конфигурирования) системе инициализации, логике построения, иерархии файлов и каталогов и системе управления пакетами. Последний критерий — наиболее общий, на основе его можно выделить два основных класса дистрибутивов: дистрибутивы пакетные (т. е. распространяемые в виде прекомпилированных пакетов) и дистрибутивы Source Based, целиком или в значительной своей части собираемые из исходных текстов. Впрочем, последние правильнее называть дистрибутивами портируемыми, какая-либо система автоматизированной сборки пакетов является их непременным атрибутом.

Поскольку систем управления бинарными пакетами существует не так уж много, по тому же критерию можно провести и более дробную классифика-

цию пакетных дистрибутивов. Здесь обособляется многочисленное семейство дистрибутивов, базируемых на rpm (Red Hat Packages Manager), семейство deb-дистрибутивов (прародителем которых был дистрибутив Debian) и разнообразные дистрибутивы с управлением пакетов в стиле Slackware. Что же касается систем Source Based, то почти каждая из них обладает уникальной системой пакетного менеджмента, в основе которой лежит идея портов FreeBSD.

Впрочем, взаимовлияние и проникновение идей в мире Open Sources таково, что удачные находки в области пакетного менеджмента распространяются по нему со скоростью лесного пожара. Так, пакетный менеджер apt, родившийся в недрах Debian, был очень быстро адаптирован для использования с пакетами rpm-формата и внедрен во многих клонах Red Hat, порты FreeBSD легли в основу всех систем автоматизации сборки пакетов из исходных текстов.

Иерархия файлов и каталогов долгое время была весьма специфичной для дистрибутивов Linux. По крайней мере, представители основных генетических линий их (такие как клоны Red Hat **http://www.redhat.com** или Debian **http://www.debian.org**) различались между собой, на горе как разработчиков, так и пользователей, достаточно отчетливо. Однако ныне активно развивается проект Filesystem Hierarchy Standard, который, можно надеяться, со временем нивелирует эти различия, по крайней мере, для главных общесистемных каталогов.

Программа установки и средства конфигурирования системы обычно считаются неотъемлемыми атрибутами самостоятельного дистрибутива. Однако это скорее теоретическая максима, к которой следует стремиться, нежели жизненная реальность. Ряд дистрибутивов, самостоятельность которых сомнению не подвергается (яркий пример Altlinux, **http://www.altlinux.ru**), наследуют инсталляторы от своего отдаленного прототипа (в данном случае Linux Mandrake, ныне — Mandriva, **http://www.mandriva.com/**). А такой самостоятельный дистрибутив, как Gentoo, программы установки не имеет вообще. Вернее, в качестве инсталлятора в нем выступает командная оболочка bash, а универсальным конфигуратором служит обычный текстовый редактор...

Система инициализации — это наборы стартовых сценариев, определяющих загрузку различных служб при запуске системы. Конечно, все многообразие стартовых наборов сводится к вариациям на две основные темы: мажорную System V и не то чтобы минорную, но более сдержанную BSD-тему (*о сути обеих разговор пойдет в главе 13*). Однако ни в рамках первой, ни в исполнении второй двух одинаковых схем инициации обнаружить, скорее всего, не удастся. И потому систему инициализации можно считать одной из существенных специфичных особенностей.

Наконец, комплектация пакетами и приложениями. Здесь можно наметить две тенденции: максимально возможный охват всего многообразия свободного софта в рамках отдельного дистрибутива и создание некоего ограниченного, но достаточного набора. Первая тенденция наиболее ярко реализована в Debian, Altlinux с Sysiphus и в Gentoo. Типичный представитель второго направления — Slackware (**http://www.slackware.org**) и его идеологические наследники типа CRUX (**http://www.crux.nu**) и Archlinux (**http://www.archlinux.org**).

Однако и в подходе к комплектации можно видеть конвергенцию признаков. С одной стороны, монструозные дистрибутивы, как правило, имеют облегченные варианты с более ограниченными наборами программ. С другой, исходно ограничивающиеся дистрибутивы (а тенденция к самоограничению наиболее отчетливо проявлена в CRUX) обычно пополняются независимыми разработчиками, результаты работы которых доступны, пусть не в составе установочных наборов, но уж где-нибудь в Сети — обязательно.

К чему я все это говорю? Да к тому, что, при всем внешнем различии дистрибутивов Linux (а при беглом сравнении, например, Linux Slackware и Linux Mandrake кажутся просто разными операционными системами), общего между ними гораздо больше, чем особенного. И потому пользователь с равным успехом может использовать любой дистрибутив из числа хорошо собранных, разумеется. Перефразируя графа нашего, пахаря: с каждым хорошим дистрибутивом пользователь будет счастлив одинаково, с каждым плохим — несчастлив по-своему. Остается только отделить зерна от плевел, дистрибутивы хорошие от дистрибутивов плохих. К счастью, подавляющее большинство известных мне дистрибутивов относится к первой категории. На отрицательных примерах я останавливаться не намерен, потому все упомянутые в этой книге дистрибутивы пользователь может числить в хороших (если прямо не оговорено).

К тому же обычно нет ни малейших препятствий к пересадке положительных особенностей одного дистрибутива на почву иного. И это проделывается не только сборщиками систем, но и индивидуальными пользователями. В результате любая Linux-система, каково бы ни было ее генетическое происхождение, в процессе эксплуатации все более индивидуализируется, становясь похожей скорее на своего пользователя, чем на своих родителей.

О BSD сотоварищи

Прояснив вопрос с дистрибутивами Linux, обратимся теперь к BSD-системам. Ибо понять их специфику лучше всего в сопоставлении с Linux и в противопоставлении ему. Как, впрочем, и наоборот: в предыдущих разделах мы видели, что понимание того, что такое Linux, выкристаллизовывается именно в сравнении с BSD-системами.

Для начала назовем наиболее распространенные BSD-системы. Это (хронологически) NetBSD, FreeBSD и OpenBSD (не считая коммерческой BSDi, впрочем, и распространена она мало, по крайней мере, на Руси ее пользователи исчисляются единицами). Они связаны общностью происхождения, от системы, именовавшейся BSD UNIX, а в дальнейшем X.XBSD, исторически позднейшая версия которой 4.4BSD. От последней и происходят современные BSD-системы, Net- и FreeBSD непосредственно, а OpenBSD — как отпочковавшаяся от NetBSD. А недавно BSD-семейство пополнилось еще одним полноправным членом — системой DragonFlyBSD.

Впрочем, и история BSD-систем — вопрос отдельный и очень интересный, мы к нему еще вернемся в следующей главе. А в контексте сегодняшнего разговора подчеркну, что Free-, Net- и OpenBSD, не говоря уже о DragonFly, — это не подвиды одной системы, как дистрибутивы Linux, а именно самостоятельные ОС, каждая со своим ядром и базовым комплектом системных и пользовательских утилит. Хотя парадоксальным образом, с точки зрения пользователя разницы между ними меньше, чем между Linux Slackware и Mandriva Linux, например.

Итак, первое: если бесчисленные дистрибутивы Linux суть вариации на тему одной и той же ОС, то редкие BSD-клоны — самостоятельные, хотя и родственные, операционные системы. И причина их обособления — ориентация на разные сферы применения.

Так, NetBSD испокон веков развивалась в русле классических традиций UNIX (и POSIX), как максимально независимая от платформы переносимая система. Как говорят, для переноса NetBSD на машины с процессорами AMD64 потребовались считанные дни.

FreeBSD, напротив, возникла и долгое время развивалась с прицелом на оптимизацию под наиболее демократичную платформу — 32-разрядные Intel-совместимые процессоры. И с возможностью работы на наиболее близких к ним, но 64-разрядных процессорах Alpha. Безвременная кончина последней архитектуры сделала эту линию бесперспективной. Однако 64-разрядные наработки были аккумулированы во FreeBSD пятой и особенно шестой ветки: кроме 64-разрядных процессоров AMD и Intel, она в состоянии работать также на Sun Sparc. Тем не менее, именно ориентация на массовые Intel-совместимые процессоры и сделала FreeBSD наиболее распространенным представителем своего клана.

Конек OpenBSD — это безопасность в самом широком смысле слова.

Наконец, специфика DragonFly — не столько в сфере применения (она позиционируется как в равной степени пригодная и для серверов, и для рабочих станций): целевое ее назначение — работа на многопроцессорных машинах, и особенно на машинах с многоядерными процессорами.

Вторая отличительная черта BSD-систем — монолитность. Если любой дистрибутив Linux являет собой более или менее тесную интеграцию ядра и базовых пакетов разного происхождения, то в каждом BSD-клоне ядро и комплекс средств его обеспечения представляют собой единое и (с некоторыми оговорками, о которых речь пойдет далее) неделимое целое, не расчленяемое на кванты-пакеты. Базовый комплекс Net- и OpenBSD поставляется в виде единого тарбалла. А во FreeBSD он хотя и разбит на фрагменты по 1,44 Мбайт, но это сделано исключительно из соображений удобства скачивания и записи (наследие тех времен, когда операционную систему еще можно было установить целиком с дискет). DragonFly же вообще переносится в процессе инсталляции в том же виде, в каком эта система присутствует на установочном компакт-диске.

Из этого вытекает третья особенность BSD-систем — их внутренняя безальтернативность. Если в Linux, как уже говорилось, чуть ли не любому компоненту, кроме ядра, можно подобрать функциональный аналог, пользователь BSD привязан к тому комплекту, который идет с ядром его системы. Характерный пример — FreeBSD. Там переход с одной ветви на другую, более новую (что эквивалентно смене версии ядра в Linux), требует полной пересборки базовой системы (механизм make world).

Монолитность базовой структуры BSD-систем отнюдь не означает, что пользователь должен в принудительном порядке держать на своей машине все ее компоненты, в том числе и заведомо ненужные. Просто для освобождения от балласта тут используются другие механизмы, например, списки исключений при обновлении через cvsup и выполнении процедуры make world.

И потому пользователь может индивидуализировать свою систему ничуть не в меньшей степени, чем при последовательной сборке Linux из исходников. В некоторых случаях это приводит к появлению разновидностей BSD-систем, формально напоминающих Linux-дистрибутивы. Известны такие производные FreeBSD, как PicoBSD (система на одной дискете, способная, тем не менее, выполнять функции не только сетевой станции, но даже Dial Up-сервера), Frenzy — работающая с компакт-диском полнофункциональная система, предназначенная для сетевого администрирования, или PC-BSD — десктоп-ориентированный вариант FreeBSD, снабженный графическим инсталлятором.

Тем не менее, BSD-системы избежали сегментации по дистрибутивному принципу. Приведенные примеры, во-первых, крайне немногочисленны, во-вторых, имеют сугубо специальное назначение, в-третьих, их развитие все равно остается в рамках генеральной линии FreeBSD: и PicoBSD, и Frenzy, и PC-BSD с точки зрения внутреннего устройства представляют собой самую обычную FreeBSD.

Тем не менее, клонирование BSD-систем также иногда имеет место, только происходит оно иначе, чем в Linux. Разновидность какой-либо BSD-системы, удалившись от своего предка, превращается в самостоятельную ОС, со своим ядром и базовым комплексом программ. Ранее я упоминал об отделении OpenBSD от родительской NetBSD. А в наши дни мы присутствуем при начале расщепления FreeBSD: летом 2003 г. от генеральной ее линии ответвилась система DragonFlyBSD, внешне почти неотличимая, но совершенно иная с точки зрения внутренней архитектуры.

Существует и другой путь размножения BSD-систем: вследствие цельности и сбалансированности базового комплекта их приложений последний подчас используется как инфраструктура, надстраивающая совершенно иные (не связанные генетически с 4.4BSD) ядра. В частности, микроядро Mach, разрабатывавшееся вплоть до второй половины 90-х годов университетами (сначала Карнеги-Меллона, а затем штата Юта).

Наиболее известный (и единственно работоспособный) пример такой надстройки — Mac OS X в ее свободной, освобожденной от проприетарных компонентов, ипостаси OpenDarwin (**http://www.opendarwin.org/**). Однако в процессе вялотекущей разработки можно обнаружить еще несколько идеологически близких систем — xMach, похоже, прекративший развитие, и Yammit, недавно разделивший его участь.

Есть случаи и иного подхода: перенос инфраструктуры Linux на ядро какого-либо BSD-клона. И тут на память приходят амбициозные проекты Debian GNU/FreeBSD (причем сразу в двух вариантах) и Debian GNU/NetBSD. И, наконец, к этой же категории примыкает попытка переноса системы портежей Gentoo на ядро и системное окружение FreeBSD взамен ее собственной системы портов (каковая в свое время послужила прототипом портежей).

Все боковые BSD-ответвления (за исключением DragonFlyBSD) на сегодняшний день могут рассматриваться как сугубо экспериментальные проекты: думаю, и сами их разработчики не надеются на всеобщее признание и широкое распространение. А упоминаю я об этих проектах по двум причинам. Во-первых, чтобы показать, что мир BSD не столь однообразен, как это может показаться на первый взгляд. А во-вторых, все эти экспериментальные проекты, вне зависимости от их успешности, отрабатывают какие-либо новые варианты, рациональная составляющая которых будет, несомненно, аккумулирована в операционных системах основных линий развития.

Вообще, взаимовлияние систем внутри BSD-клана — тоже отличительная его особенность. Удачные решения одной из ОС очень быстро распространяются среди ее сородичей. Пример — система портов FreeBSD, заимствованная в OpenBSD с самого начала, а в дальнейшем (в виде системы pkgsrc) распространившаяся и на NetBSD. С другой стороны, многие отличитель-

ные особенности пятой ветки FreeBSD, напротив, уходят своими корнями в NetBSD (и даже в коммерческую BSDi). Есть и примеры внедрения во FreeBSD новшеств из ее отпрыска — DragonFly.

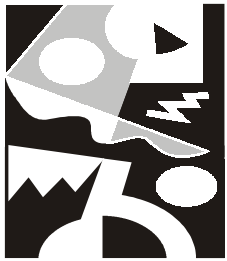
Так что мир BSD-систем в значительной мере сохраняет свое идеологическое единство. Более того, его влияние распространяется и на Linux: я неоднократно подчеркивал, что Source Based дистрибутивы Linux в значительной мере основываются на идейном наследии FreeBSD.

Все сказанное не имеет целью доказать преимущества BSD-систем над Linux (впрочем, как и противоположное утверждение): споры такого характера я полагаю беспредметными. Просто я хотел подчеркнуть, что на Linux мир свободных POSIX-систем не исчерпывается, все они находятся в постоянном взаимодействии друг с другом, и понимание особенностей BSD-систем может способствовать более глубокому освоению Linux, и наоборот. А уж что выбрать для личного употребления — дело целей, задач, обстоятельств, вкуса, а может быть, даже и случая.

Вообще проблема выбора первой системы — широка и многогранна, и потому будет детально рассмотрена *в главе 5*. В ней я постараюсь обосновать, почему именно FreeBSD, наряду с парой-тройкой дистрибутивов Linux, видятся мне самыми благоприятными объектами для изучения POSIX-систем вообще. Но, повторяю, это не более, чем мое личное мнение. Если после прочтения этих страниц вы сможете сформировать свой собственный взгляд на вопрос выбора системы и (или) дистрибутива — одну из целей своего сочинения я буду считать достигнутой.

Следует уделить внимание вопросу, как же POSIX-системы дошли до жизни такой. Для чего придется погрузиться в глубины истории, подобно тому, как это сделал Гэндальф, расследуя происхождение Кольца Всевластия...

Глава 3



Вопросы истории POSIX'ивизма

- Bell-прелюдия.
- Берклиада UNIX-кода.
- Пусть расцветают все цветы.
- Упорядочивание стилей работы.
- Увертюра Линуса.
- Свободная берклиада: продолжение истории.

На протяжении предыдущих глав то и дело упоминались названия различных операционных систем: Linux, FreeBSD, а также некоторые другие. А внимательный читатель компьютерной литературы время от времени (правда, теперь — все реже) встречается, с одной стороны, со звучными именами типа Solaris или UnixWare, с другой же — с аббревиатурами вроде AIX или HP-UX. Все вышепоименованные ОС имеют ряд общих черт, позволяющих объединить их в единое семейство. Имена ему суть многи: величают эти системы и UNIX, и юниксоидами, и UNIX-подобными ОС. Хотя наиболее строгий титул им всем будет — POSIX-совместимые операционные системы. Откуда же они взялись? Об этом и пойдет нынче разговор.

Bell-прелюдия

Совсем-совсем недавно (а конкретно, 1 января 2000 г.) все прогрессивное человечество широко отметило (в узком кругу) тридцатилетний юбилей первого UNIX. Разумеется, это не значит, что операционная система UNIX (а некогда это действительно была конкретная операционная система) волшебным образом возникла в этот день, как Афина Паллада из головы Зевса. Создание ОС — процесс, несколько длящийся во времени, и потому дату рождения ее установить затруднительно. Просто системные часы всех UNIX-машин во всем мире отсчитывают свое время (в секундах) с той самой знаменательной даты. И это — одно из многочисленных соглашений, о

которых пойдет речь в этой главе (и которые имеют непосредственное отношение к нашему сюжету).

Вообще говоря, о первой пятилетке истории UNIX (1969—1974 гг.) написано немало, поэтому ограничусь основными фактами. Началось все с того, что Bell Labs (в то время подразделение корпорации AT&T) совместно с General Electric и Массачусетским технологическим институтом разрабатывала очень прогрессивную по тем временам ОС под названием Multics, многопользовательскую и многозадачную, как легко догадаться из ее названия. Причем — безуспешно, на причинах чего здесь останавливаться не будем. Однако разработка эта имела некое побочное следствие: один из участников проекта, Кен Томпсон, написал под нее игру Space Travel.

В 1969 г. проект Multics был закрыт, и играть Кену стало не на чем. Благо, в закромах Bell Labs обнаружилась машина PDP-7, которая и была попутно использована под игровые цели. Правда, игра Кена в родной операционке не работала, так что заодно пришлось написать для нее и ОС. Созданная на базе Multics, она получила название Unics, ибо, в отличие от прототипа, пользователь у нее первоначально был один. Позднее к Кену присоединился Денис Ритчи, в честь чего и название трансформировалось в UNIX.

На дальнейшую судьбу UNIX огромное влияние оказали юридические коллизии того момента. Еще до создания системы корпорация AT&T подверглась антимонопольному преследованию, в результате чего потерпела поражение в правах: на деятельность ее был наложен ряд ограничений. В частности, она не имела разрешения торговать программными продуктами, в число коих попадала и новорожденная операционная система UNIX. Так что та на протяжении первых 5 лет своего существования вела исключительно внутриутробное существование в лоне породившей ее компании.

Правда, за это время UNIX сформировалась как концептуальная целостность: понятие файла как универсального интерфейса доступа ко всему на свете, командный интерпретатор (шелл) как столь же универсальное средство взаимодействия пользователя с системой и набор запускаемых из него монофункциональных утилит, комбинации которых позволяли решать весьма сложные задачи. Получила UNIX и свою файловую систему, очень простую (в частности, в ней отсутствовало представление о типах файлов, все они трактовались просто как последовательности байтов) и, одновременно, очень по тем временам эффективную. Главное же, система была почти целиком (хотя и не сразу) написана на компилируемом языке высокого уровня — специально для нее созданном Си, что создавало предпосылки для ее переноса на почти любые аппаратные платформы. Тем не менее, до настоящего UNIX, каким мы его представляем нынче, было еще далеко.

Все нововведения в первозданном UNIX находили отражение в его титулатуре. Сначала они знаменовались сменой версий: было их чуть ли не 10. Затем количество перешло в качество, и версии стали системами — System III,

скажем, потом System V, которой, кстати, суждено было стать последней. Дальнейшие модификации в ее недрах получали имена реализаций: System V Release 3, System V Release 4. Последняя также оказалась знаковой, она (под идеограммой SVR4) легла в основу большинства современных коммерческих UNIX. Впрочем, я существенно забежал вперед.

Следует помнить, что все это происходило в бездушном и бездуховном мире чистогана, где, как известно, все покупается и все продается. А вот продавать-то UNIX как раз было нельзя. Однако не пропадать же добру, созданному как бы и самопроизвольно, но — на коммерческом оборудовании. И компания AT&T, юридический владелец UNIX, начиная с 1974 г., стала передавать исходные тексты в университеты (преимущественно Америки, но также и некоторых других стран) и прочие учреждения — "в образовательных целях", как обычно говорится в источниках.

Это не было свободным распространением в том смысле, который потом вложили в это понятие апологеты движения Open Source, начиная с Ричарда Столлмена и кончая героями Free Software Foundations. Правда, UNIX (вернее, в то время — не более, чем ее прототип) передавалась целиком в исходных текстах с правом их изучения, модификации, доработки и прочего потрошения. Однако для производства таких действий требовалось обладание лицензией на исходный код UNIX. Лицензия же могла быть предоставлена владельцем (т. е. AT&T) вместе с самой системой, но — уже за деньги (хотя, как пишут, и символические).

Главное же отличие от принципов Open Source заключалось в том, что условия этой самой лицензии не допускали дальнейшего свободного распространения ни системы целиком, ни каких-либо ее компонентов, содержащих исходный код UNIX. "Что, собственно, и создало сюжет". Вернее, интригу всего дальнейшего, почти детективного, сюжета...

Однако до юридических коллизий было еще далеко. А пока университеты с радостью приобщались к новой операционной системе, в которой были реализованы все передовые идеи того времени. И к тому же, в принципе способной функционировать практически на всем спектре тогдашнего оборудования. Напомню, что речь идет о середине 70-х годов прошлого века: Стив Джобс еще не помышлял о продаже калькулятора и использовал родительский гараж по прямому назначению, а Билл Гейтс не освободил еще мир от засилья CP/M.

Берклиада UNIX-кода

Одним из первых учреждений американского наробраза, получившего доступ к исходникам UNIX, оказался университет Беркли (штат Калифорния). Именно сюда UNIX "вписался с точностью патрона, досланного в патрон-

ник" (Олег Куваев). И здесь с лихвой выполнил свои "научно-образовательные цели"...

В Беркли, в условиях открытого общения профессиональных специалистов в области зарождавшихся Computer Science, система UNIX медленно, но верно превращалась в то, чем она стала ныне. И в значительной мере, именно усилиями трудящихся из университета, объединенных в Computer System Research Group (CSRG).

Что же принципиально нового привнесли берклианцы в UNIX? Во-первых, конечно же, интеграцию в ядро системы протокола TCP/IP. Того самого, на котором базируется весь сегодняшний Интернет. Собственно, ради этого американские компьютерщики в штатском и финансировали работы университета Беркли, ведь первоначально Интернет задумывался как отказоустойчивая система правительственной связи на случай советского ядерного удара. Однако это — совсем отдельная история...

Следующим принципиальным вкладом Беркли была реализация файловой системы. Поскольку именно это весьма важно для нас, простых пользователей обычных персоналок, остановлюсь на этом вопросе подробнее.

Извинение

Возможно, сказанное покажется не вполне понятным начинающему пользователю, скажем, Linux. Надеюсь, что все недоразумения разрешатся при дальнейшем чтении моего сочинения, на протяжении которого к вопросам устройства файлов и файловых систем придется возвращаться неоднократно.

Я уже говорил, что основные характеристики файловой системы были сформированы уже в первом UNIX. Эта файловая система (получившая впоследствии название s5fs, т. е. файловая система System V) базировалась на трех китах: понятиях суперблока, таблицы так называемых inodes и области блоков данных.

Суперблок — это часть файловой системы, описывающая ее в целом: положение на физическом носителе, размер минимального кванта информации (логического блока) и их суммарное количество, число свободных и занятых блоков и т. д.

Таблица inodes (индексных, или информационных, узлов — однако лучше сохранить этот термин без перевода, опять-таки, как своего рода идеограмму) — это метаданные (т. е. данные о данных) файлов файловой системы. Для каждого файла она содержит уникальный идентификатор, по которому он находится системой, число ссылок на него (фигурально говоря, количество имен файла — как будет сказано в *главе 8*, любой файл может иметь несколько имен), размер, атрибуты файла — принадлежности (владельца файла, группы, к которой он принадлежит, и прочих), доступа (права чтения, исполнения, изменения), времени (последнего обращения, изменения данных и метаданных) и еще некоторые.

Адреса блоков данных файла, т. е. собственно физического расположения его контента на диске, также описываются в таблице inodes. Ну, а сами блоки данных (т. е. полезная для пользователя информация), как легко догадаться, и составляют наполнение одноименной области.

В устройство файловой системы s5fs изначально были заложены три существенных ограничения с точки зрения надежности, экономичности и быстродействия.

1. Суперблок ее имелся в единственном экземпляре, и его утрата (например, вследствие физического повреждения носителя) автоматически влекла недоступность всех данных.
2. s5fs была образована некими квантами информации — логическими блоками, имевшими размер от 512 (размер физического дискового блока) до 2048 байтов. Очевидно, что увеличение размера блока вело к повышению быстродействия дисковых операций. Однако не менее ясно, что при этом дисковое пространство расходовалось неэффективно: ведь любой файл, сколь мал бы он ни был (а в UNIX количество очень маленьких файлов весьма велико), занимал логический блок целиком.
3. Ограничение быстродействия сказалось позднее, когда "винчестеры стали большими". Легко сообразить, что единственная таблица inodes при непрерывности области данных на разделах большого объема требовала значительных перемещений считывающих головок диска, ведь каждая операция чтения файла или его записи требовала обращения сначала к метаданным файла, а потом — к его данным.

Это — что касается физики хранения данных. Логически же файловая система UNIX изначально приобрела древовидную (или иерархическую) структуру, основанную на разделении каталогов и всех прочих файлов. Каталоги образовывали как бы скелет файловой иерархии и содержали только идентификаторы файлов и их имена — это и были те самые ссылки, о которых говорилось чуть раньше. Причем вследствие изначально принятого формата каталога длина имени файла (как это ни покажется парадоксальным пользователям нынешних Linux или BSD, привыкшим к файлам с именами длины немерянной) ограничивалась 14 символами. Ведь каталог — это, в сущности, база данных для идентификаторов файлов и их имен, состоящая, как и любая БД, из записей и полей. Так вот, каждая каталожная запись в s5fs имела фиксированную длину в 16 байтов, из которых два первых отводились под идентификатор, так что на имя этих байтов оставалось только 14.

Разработки университета Беркли сняли многие ограничения как физики, так и логики s5fs. Во-первых, единое пространство файловой системы было разделено на части, именовавшиеся группами цилиндров, каждая из которых имела копию суперблока, самостоятельную таблицу inodes и область данных. Это давало большой прирост в быстродействии файловых операций за счет минимизации перемещения дисковых головок. Плюс к тому, дубли-

рование критически важной части файловой системы — суперблока — весьма способствовало надежности.

Далее, было введено понятие внутренней фрагментации. То есть каждый логический блок файловой системы разделялся на части (фрагменты), которые могли адресоваться независимо. И, соответственно, если файл имел размер менее логического блока, то реально он занимал не его целиком, а только один (или несколько) таких фрагментов, остальные же сохранялись свободными и могли быть использованы для других целей. Но дисковые операции все равно осуществлялись поблочно, так что на их быстродействии фрагментация почти не сказывалась.

Все это привело к потере совместимости новой файловой системы с файловой системой изначального UNIX. И потому (снявши голову, плачут ли по волосам?) разработчики из Беркли решились на еще один радикальный шаг — изменение формата каталогов. Каждая запись в них разделилась на постоянную часть — идентификатор, размер переменной части и размер имени файла — и переменную, содержащую собственно имя файла. В результате максимально возможная длина последнего возросла до 256 символов, что с точки зрения здравого смысла можно считать практически безграничным — вряд ли кому придет в голову изобретать имя файла длиннее трех строк стандартной текстовой консоли (и, тем паче, запомнить таковое).

Утрата совместимости в формате каталогов, казалось бы, должна была стать препятствием для использования программ, ведь подавляющее большинство их уже тогда создавалось под абстрактный UNIX, а трудно представить себе пользовательское приложение или системную утилиту, не нуждающуюся в считывании имен файлов. Однако в Беркли было предусмотрено и это: путем описания специальных процедур открытия и чтения каталогов, независимо от формата последних. Что во всю заиграло позднее, когда файловых систем в UNIX стало множество.

Новая файловая система получила имя FFS (Fast File System, быстрая файловая система), что подчеркивало ее быстродействие относительно исходной s5fs. Я столь подробно остановился на ее отличиях от прототипа по двум причинам. Во-первых, из-за важности их для пользователей. И во-вторых, потому что реализованные в FFS принципы разделения файловой системы и фрагментации ее блоков были ассимилированы во всех последующих файловых системах, применяемых ныне в любых разновидностях UNIX. К этим принципам восходят и группы блоков в файловой системе Linux (ext2fs), и экстенты (extents) JFS, и, особенно, allocations group в XFS, выступающие здесь уже как почти самостоятельные файловые субсистемы.

И все потому, что университет Беркли, представляя собой обычное научное сообщество, отнюдь не делал секрета из своих разработок. Распространяя их на условиях, общепринятых в академических кругах всех времен и народов. То есть все эти результаты публиковались в простых научных журналах и

были доступны для воспроизведения всем желающим (и, что немаловажно, могущим). Позднее такие условия распространения софта, с одной стороны, легли в основу лицензии BSD. А с другой, Ричард Столлмен, в юности не чуждый академической науке, опираясь на них, изобрел очень похожий велосипед и назвал его GPL (General Public License).

Распространялась и сама берклианская разновидность UNIX как программный продукт. Группа CSRG, начиная с 1976 г., внедряла в народе свои достижения — на магнитных лентах, под названием Berkeley Software Distribution, что, с указанием номера версии, и дало в дальнейшем имя системе. Это было первичное понимание аббревиатуры BSD, которое не следует путать с возникшим позднее BSDi — Berkeley Software Development, Inc. (или Berkeley Software Design, Inc., мне попадались обе расшифровки этой аббревиатуры), фирмой, выпускающей коммерческий клон BSD-систем, называемый, строго говоря, BSD/386 (хотя в обиходе за этой системой закрепилось то же имя — BSDi).

Первоначально (под именем 2BSD и по цене 50 американских рублей) распространялся только пакет собственно берклианских наработок, включающий, в частности, командную облочку C-Shell и культовый редактор юниксоидов vi. Однако, начиная с 3BSD, берклианские ленты включали уже и модернизированную UNIX-систему. Особенно обильные побеги дала ее четвертая ветка — 4.0BSD, 4.1BSD с несколькими подвариантами, 4.2BSD, 4.3BSD и, под занавес истории, 4.4BSD.

Наиболее существенным этапом был выпуск в 1982 г. системы 4.2BSD: именно в ней была реализована файловая система FFS, да и большинство прочих специфических особенностей BSD-систем. Не случайно тип раздела FreeBSD (и DragonFlyBSD) по сию пору идентифицируется как раздел 4.2BSD, а до недавнего времени этот же идентификатор имели и разделы Net- и OpenBSD. С этого момента стало реальностью сосуществование двух самостоятельных линий развития UNIX — System III/V и BSD UNIX.

А системе 4.4BSD была предначертана особая судьба, о чем я расскажу несколько позднее.

Пусть расцветают все цветы

Конечно, система UNIX развивалась не только в Беркли. Во-первых, не забрасывала свое детище сама AT&T во всех ее проявлениях. Правда, разработка системы плавно перемещалась от Bell Labs к иным подразделениям, типа группы поддержки UNIX и т. д., однако в мои цели не входит описание истории этой корпорации, и потому названия их я опускаю.

Тем не менее, из недр AT&T (в обобщенном понимании этого термина) последовательно выходят System III, System V, System V Release 2 (SVR2) и,

наконец, System V Release 3 (SVR3); этой системе, разработанной в 1987 г., суждено было стать последним представителем чистой линии первозданного UNIX. Ибо уже System V Release 4 (SVR4), появившаяся в 1989 г., включила в себя множество новшеств из разработки 4BSD — интеграцию с TCP/IP, поддержку FFS, берклианскую реализацию механизма виртуальной памяти и многое другое. В частности, легший впоследствии в основу стандарта шелл Корна, несмотря на совместимость с первозданным шеллом (Bourne Shell), заимствовал из C-Shell такие особенности, как управление заданиями, историю команд, поддержку псевдонимов и т. д. (впрочем, развитие шеллов — отдельная история, к которой я еще вернусь в *главе 15*).

Благо, как уже было сказано, берклианские разработки распространялись в соответствии с принципами открытой науки — практически свободно. И если AT&T не всегда заимствовала их на уровне реализаций (т. е. непосредственно кода), то идейное их влияние на первозданный UNIX было несомненным.

Во-вторых, портируемость UNIX оценили производители специфического оборудования, имена им были IBM, Sun, Hewlett-Packard, DEC и еще несколько забытых. Базируясь на SVR3, BSD, позднее — на SVR4, они адаптировали систему под собственные архитектуры, создав такие варианты, как AIX, SunOS, HP-UX, Digital UNIX соответственно. Все это были коммерческие или, как нынче модно говорить, проприетарные, операционки, однако, будучи жестко привязанными к аппаратуре фирм-производителей, в свободной продаже (подобно нынешним Windows) они не присутствовали.

В-третьих, чисто софтверные фирмы подняли мутный вал уже коммерческих UNIX. Это были, в частности, Interactive Systems и Santa Cruz Operations (косвенный предок скандально прославившейся ныне SCO). Не имея собственных аппаратных платформ, они адаптировали UNIX под платформы общераспространенные, коими были в то время сначала PDP, а затем и IBM PC. Кажется, именно SCO UNIX стала первой представительницей семейства UNIX, портированной на машины с процессором Intel (конкретно — на i386).

Как ни странно, в числе первых на этом поприще отметилась и Самая Великая Софтверная фирма всех времен и народов. Ею была создана система XENIX для тех же i386-машин. По словам видевших ее, это было нечто вроде однопользовательской реализации UNIX — как всегда, Microsoft и тут, подобно товарищу Ленину, пошла своим путем...

Наконец, Беркли был не единственным университетом, приобщившимся к миру UNIX. Разработки в этом направлении велись в нескольких академических учреждениях США, а также иных стран (например, Франции и Австралии). Однако наибольшую известность стяжала деятельность университета Карнеги-Меллона, результатом которой явилась микроядерная UNIX-совместимая система Mach (одно время она виделась операционной систе-

мой будущего — на ней базировались и Digital UNIX, и знаменитый NextStep, всерьез поговаривали об использовании ее в разработке OS/2). А Ричард Столлмен положил ядро Mach в основу своего перманентного долгостроя — GNU/Hurd, **http://www.gnu.org/software/hurd/**.

Однако главная слава Mach оказалась посмертной. Ибо, во-первых, под ее воздействием Энди Танненбаум написал свою игрушечную систему MINIX, которая вдохновила Линуса Торвальдса на написание Linux. А во-вторых, микроядро Mach нынче составляет сердце Mac OS X. Но в рамках описываемых событий это — опять же дела грядущего.

А пока можно было констатировать, что в конце 70-х годов прошлого века UNIX развивался в соответствии со знаменитым лозунгом Великого кормчего китайского народа: "Пусть расцветают все цветы!" В итоге, кроме двух существенно различных базовых системы — System V и BSD, расцвело не менее дюжины вариантов, в той или иной мере различающихся между собой. Мною были упомянуты далеко не все из них, а лишь те, которые уцелели и поныне. Или — те, о которых мне довелось в свое время хоть что-то прочитать. На самом деле UNIX-клонов в те времена было гораздо больше.

Упорядочивание стилей работы

Постепенно различия между клонами становились все более значительными. Во-первых, в основе существовавших вариантов UNIX лежали разные базовые системы — SVR3, SVR4, 4BSD. Во-вторых, каждый разработчик считал своим долгом либо адаптировать систему под возможности собственной аппаратной платформы, либо просто внести те или иные усовершенствования. А поскольку практически все существовавшие системы были не только коммерческими, но и закрытыми, усовершенствования эти слабо согласовывались между собой. И в любом случае, реализовывались различным образом.

Это ставило под угрозу один из краеугольных камней UNIX-идеологии — портируемость приложений. А традиции писать программы под абстрактный UNIX никто не отменял. И начался процесс, который, вслед за товарищем Мао, можно назвать "борьбой за упорядочивание трех стилей работы". Однако председателю КПК было попроще, в данном случае речь шла не о трех, а едва ли не о тридцати трех стилях...

Тем не менее, процесс упорядочивания пошел. Первый шаг в этом направлении резонно было бы сделать основоположнику UNIX. И AT&T создала документ, описывающий спецификации, которым должна соответствовать система, претендующая на звание UNIX — SVID (System V Interface Definition, т. е. определение интерфейса System V). И даже разработала

программу, которая проверяла претендента на соответствие этим спецификациям, — System V Verification Suite.

Нетрудно было предположить, что в этих спецификациях нашли отражение только те особенности BSD, которые были включены в каноническую System V. А ведь BSD была столь же полноводным источником, из которого черпали все производители UNIX. Их такое положение дел не очень устроило. И потому было создано несколько организаций, разрабатывающих свои версии стандартов для операционных систем, расширенные по сравнению со SVID. Наибольшее признание из таких разработок заслужил стандарт POSIX (Portable Operation System Interface based on uniX), описанный вкратце в предыдущей главе. В рамках же исторического обзора добавлю только, что сама аббревиатура POSIX была придумана все тем же Ричардом Столлменом, и, по его словам, должна произвноситься как "пазикс" (с долгим "а").

Именно на стандарты POSIX в первую очередь и опирался Линус Торвальдс, создавая свою ОС по мотивам MINIX. Однако это уже следующая история.

Увертюра Линуса

Как было сказано ранее, UNIX в разных его проявлениях получил широкое распространение в университетах всего мира. Более того, он стал основой базового академического образования в области Computer Science (и не только — UNIX кое-где учат и студентов-геологов, например). И потому возникла необходимость в соответствующих учебных пособиях, каковые и появились во множестве.

Однако обучать операционной системе без самой системы — штука столь же сложная, как и учить плаванию посреди пустыни Кызыл-Кум. Система же UNIX (даже в берклианском своем варианте) оставалась отнюдь не общедоступной вследствие ограничений на распространение, привязки большинства вариантов к дорогим аппаратным платформам и, наконец, просто своей сложности.

Это противоречие разрешил Энди Танненбаум, профессор Амстердамского университета, написав "пионерский" вариант UNIX — ОС MINIX, предназначенную исключительно для образовательных целей. И распространял ее в виде набора дискет вместе со своим учебником по теории операционных систем.

По свидетельству очевидцев, в MINIX все было как в настоящем UNIX, за одним исключением: она была неспособна к выполнению реальных пользовательских задач. Многие особенности UNIX в ней не были реализованы

сознательно — Танненбаум полагал их слишком сложными для неокрепших студенческих умов.

Довольно быстро сложилось сообщество пользователей MINIX, которых такое положение дел не устраивало. И система начала обрастать всякого рода доработками (патчами), превращавшими ее в среду для реального использования. Однако распространять такую модернизированную систему было нельзя: MINIX не являлась Free Software в собственном смысле этого слова. Каждый пользователь должен был приобретать книгу Танненбаума, получая в придачу ОС, и в индивидуальном порядке накладывать на нее патчи.

Именно таким образом поступил в 1990 г. некий студент университета Хельсинки — Линус Торвальдс, когда ему захотелось поизучать внутреннее устройство процессора своей новой машины (i386) и одновременно приобщиться к UNIX, которую он изучал в своем вузе: купил книгу Танненбаума вместе с MINIX, скачал и установил соответствующие патчи.

Далее ему потребовался терминальный доступ к университетской машине, чтобы выходить в сеть, не выходя из дома (как я его понимаю...). В MINIX подходящей программы штатно не обнаружилось, и Линусу пришлось сочинить ее. Однако под управлением ядра MINIX она работала плохо, так что волей-неволей потребовалось новое ядро. Его нужно было куда-то размещать — так родилась новая файловая система ext (extended, расширение для файловой системы MINIX). Для запуска терминальной программы не худо было иметь какую-то среду — для этой цели Линус портировал на свое ядро командную оболочку bash, разработанную в недрах проекта GNU. Ну и все это хозяйство требовалось чем-то собирать: не перезагружаться же обратно в MINIX каждый раз, как потребуется внести коррективы в исходники. Так что к bash пришлось присоединить еще и единственный свободный из наличных компиляторов Си — GNU C Compiler (известный как gcc). А итогом всего этого явилось объявление в августе 1991 г. о создании новой операционной системы. Видите, как все просто?

Новая ОС получила название Linux — в определенной степени случайно: так именовался домашний каталог Линуса на том FTP-сервере, на котором она впервые была выложена для свободного доступа. Сначала — под лицензией, сочиненной Торвальдсом собственноручно. Однако вскоре, не собираясь особо заморачиваться юридическим крючкотворством, с одной стороны, но желая оградить свое творение от враждебных посягательств — с другой, он изменил лицензию на GPL, под которой распространялись такие важные ее компоненты, как bash и gcc.

Впрочем, история создания Linux давно получила широкую известность — в том числе и в версии ее разработчика. И пересказывать ее я не буду — книга Линуса доступна на русском языке как в официальном бумажном издании, так и в нескольких онлайновых вариантах. Остановлюсь только на тех моментах, которые кажутся мне наиболее существенными.

Во-первых, следует подчеркнуть, что Линус Торвальдс не занимался адаптацией MINIX (или какого-либо иного варианта UNIX): в своей работе он руководствовался описаниями системных вызовов, данными в соответствующем стандарте POSIX, не привязанными к какой-либо конкретной реализации. В результате Linux не является чистым клоном UNIX: его можно считать первой настоящей POSIX-системой в полном смысле этого слова. А черты сходства с какими-либо вариантами System V или BSD обусловлены только тем, насколько полно все они воплощают соответствующие стандарты.

Во-вторых, Linux создавался на машине с процессором i386 для архитектуры Intel и первоначально — только для нее. Более того, долгое время автор вообще сомневался, что его система когда-либо сможет быть портирована на любую иную аппаратную платформу. И потому соответствие стандартам в данном случае преследовало целью не переносимость Linux самого по себе, а в первую очередь возможность компиляции в этой ОС всего ранее созданного программного ассортимента для UNIX и POSIX-совместимых систем вообще.

В этой связи можно только порадоваться интуиции Линуса: если во время создания его системы стоял вопрос о возможности сборки сторонних программ в ней, то ныне более актуальна задача переноса Linux-софта на другие UNIX-подобные платформы.

В-третьих... а третье тесно связано со вторым: Линус оказался создателем уникального метода разработки масштабных проектов Open Source, того самого, который Эрик Раймонд позднее назовет методом большого базара. Впрочем, справедливости ради, следует отметить, что и в данном случае изобретался велосипед, аналогичный способ привлечения дармовой рабочей силы использовал Том Сойер в своих приключениях.

Ранее свободное программное обеспечение создавалось либо в рамках университетских проектов при правительственном финансировании, либо энтузиастами-одиночками, либо небольшими группами таких энтузиастов. Это было причиной заведомо нестабильного состояния: прекращение госфинансирования, утрата интереса со стороны разработчиков и масса прочих факторов в любой момент могли если не остановить такой проект, то, по крайней мере, заморозить его на долгое время.

Линус Торвальдс же обеспечил своему проекту, помимо практического бессмертия, еще и непрерывность развития. Выложив свою систему в открытый доступ, он, кроме того, предложил всем желающим (и, конечно же, могущим) принять участие в ее дальнейшем совершенствовании. В том числе (и в первую очередь) тех ее фрагментов, которые его лично мало интересовали (касающихся пользовательских приложений). А соответствие системы стандартам гарантировало, что такой призыв встретит понимание со стороны многочисленных UNIX-программистов, ведь в любом случае их

усилия не пропали бы даром и могли быть использованы в любой иной POSIX-совместимой системе...

В результате деятельности аморфного, неорганизованного, "базарного" коллектива разработчиков, не имеющих первоначально никаких источников финансирования, ядро Linux очень быстро обросло такими функциями, как поддержка сетей, протокола TCP/IP, оконной системы X, на нее были портированы все аналоги классических UNIX-утилит и приложений, созданные как в рамках проекта GNU, так и независимыми разработчиками. А затем настал черед и чисто пользовательских приложений, в том числе офисных, графических и мультимедийных. Следствием чего было привлечение не только новых разработчиков, но и конечных пользователей.

И ныне по числу пользователей Linux много превосходит все прочие POSIX-совместимые системы, вместе взятые. Причем если среди последних представлены почти исключительно разработчики программного обеспечения и администраторы компьютерных систем разного ранга, то Linux все больше проникает на десктопы тех юзеров, которых принято называть конечными, т. е. профессионально не связанных с IT-индустрией.

Свободная берклиада: продолжение истории

Не стояло на месте и развитие BSD-систем. Мы расстались с ними в исторический момент, на рубеже 80—90-х годов. До этого BSD UNIX (а система эта носила тогда это имя) развивалась более-менее во взаимодействии с прочими ветвями этой системы, давая время от времени боковые побеги, в том числе и коммерческие, такие как SunOS, например, или A/UX (уже встарь были попытки приобщения Macintosh к миру UNIX, вылившиеся в конечном счете в Mac OS X).

Однако к рубежу 90-х годов выяснилось, что исходного UNIX-кода в составе берклианской ветви UNIX осталось не так уж и много. И родилась идея создания полностью открытой операционной системы, распространяемой свободно и в исходных текстах. К этому же времени прекратилось финансирование группы CSRG, и она столь же благополучно распалась. Но дело ее не пропало. Его подхватили, расширили, укрепили и закалили в боях многие из бывших членов CSRG.

Именно созданием общедоступной UNIX-системы, причем на общедоступной платформе, сиречь Intel x86 (ведь эпоха персоналок уже началась), озаботились Вильям и Линна Джолитц. Базируясь на одном из побегов ветви 3BSD — BSD Net/2, они дописали недостающие компоненты и создали 386BSD — первую BSD-систему, претендовавшую на звание открытой в собственном смысле слова. И еще это был первый берклианский побег,

портированный на PC (IBM-совместимые компьютеры, как их тогда еще задумчиво называли).

Система эта не была еще вполне готовой к употреблению. Однако она активно исправлялась и улучшалась весьма широким кругом разработчиков, благодаря чему возник patchkit — корректирующий набор, позволяющий превратить 386BSD в работоспособную операционку.

Однако поддержание такого комплекта заплат оказалось задачей хлопотной и не очень благодарной. Вероятно, именно поэтому основоположник 386BSD Вильям Джолитц постепенно отошел от этой деятельности, и к началу 1993 г. система "сильно страдала от почти годичного пренебрежения к ней автора". Об этом свидетельствует очевидец и активный участник событий, Джордан Хаббард, о чем можно прочитать и по-русски (**http://www.freebsd.org.ua/doc/ru_RU.KOI8-R/books/handbook/history.html**).

Тем не менее, история не закончилась. В том же 1993 г. три последних координатора "заплаточного проекта" — упомянутый Джордан Хаббард, Нейт Вильямс и Род Граймс, решили "привести промежуточный снапшот 386BSD в порядок, исправив множество проблем, которые механизм patchkit не мог решить". Это предполагалось сделать путем предоставления промежуточных "очистных" снапшотов. Однако планы тройки по борьбе с заплатами "были грубо оборваны, когда Билл внезапно решил прекратить поддержку проекта без каких-либо ясных комментариев, что должно быть сделано".

Однако и это не очень повредило делу. К проекту присоединились Джулиан Элишер и Дэвид Гринмен. Именно последнему принадлежит заслуга изобретения имени нового проекта — FreeBSD и приобретения на него права собственности.

Вахта же Хаббарда выразилась в том, что он "связался с Walnut Creek CDROM с мыслью о путях последующего улучшения каналов распространения FreeBSD для множества невезучих без доступа к Интернету. Walnut Creek CDROM не только поддержал идею распространения FreeBSD на CD, но также пошел далеко вперед и предоставил проекту компьютер для работы и быстрый доступ к Интернету. Без почти беспрецедентной веры Walnut Creek CDROM в то время полностью неизвестный проект, вряд ли FreeBSD зашла бы так далеко и так быстро, как сегодня".

В декабре 1993 г. совместные усилия проекта FreeBSD и Walnut Creek обрели зримое воплощение в виде системы FreeBSD 1.0, распространявшейся как с FTP-серверов (вспомним: тогда это был почти единственный способ получения софта, за исключением коммерческого "коробочного"), так и на компакт-диске.

Ветка FreeBSD 1.x базировалась все на той же системе Net/2, что и произведение Джолитца, из которого она включила многочисленные дополнения. Кроме того, существенным компонентом ее стали утилиты и прило-

жения проекта GNU. Все это были открытые и свободные разработки. Однако исходная лента Net/2 изначально содержала некоторое количество проприетарного UNIX-кода. И это были именно критически важные фрагменты, превращавшие берклианские разработки в цельную работоспособную систему.

На основе 4BSD в это же время развивались еще два проекта: BSD/OS и NetBSD. Первый имел коммерческий статус и ныне воплотился в 386BSD — UNIX-подобную систему, распространяемую за деньги, но с исходными текстами (хотя последние — за отдельную мзду).

Проект NetBSD зародился даже несколько раньше, чем FreeBSD (первый выпуск датируется апрелем 1993 г.), и также имел целью реализовать полностью открытый и свободный вариант UNIX. В отличие от FreeBSD, с упором на мультиплатформенность: трудно поверить, что в начале 90-х ему предрекали успех.

Тут-то господа правообладатели, чуя наживу, и напомнили разработчикам из Беркли о своих правах. А права на исходный код UNIX и торговую марку, нареченную этим именем, приобрела у AT&T фирма Novell. В тот момент одержимая, подобно сиятельному Камильбеку из "Повести о Ходже Насреддине", хватательным рвением.

Начался вялотекущий судебный процесс о легальности версии Net/2 из Беркли. В результате из системы, лежащей в основе FreeBSD и NetBSD, были изъяты все следы частнособственнического кода (а, повторяю, речь шла именно о критически важных фрагментах). Гильотинированная версия получила имя 4.4BSD-Lite, и всем претендентам на BSD-наследие было рекомендовано в добровольно-принудительном порядке перейти на ее использование.

Катастрофа свободных BSD-систем казалась неизбежной. Но "приключения никогда не кончаются". И потому снова слово Джордану:

"Тогда проект FreeBSD приступил к сложнейшей задаче буквально пересоздания с нуля на основе абсолютно новой и довольно неполной системы 4.4BSD-Lite. Версии Lite были в прямом смысле light (легкими) отчасти потому, что группа CSRG удалила большие куски кода, необходимого для создания реально загружающейся системы, и фактически порт 4.4BSD для платформы Intel был очень неполным".

Реинкарнация недостающих фрагментов заняла около года. И в итоге первая версия FreeBSD 2.0, снискавшая значительный успех (к лицензионной чистоте которой не смог бы придраться ни один сутяга), вышла в декабре 1994 г. Именно она положила начало традиции, не прерывающейся и поныне.

Аналогичным путем и примерно в то же время были решены лицензионные проблемы и разработчиками NetBSD. Однако ни та, ни другая системы уже

не были лидерами в мире свободных Linux-клонов: на этом месте прочно утвердился Linux.

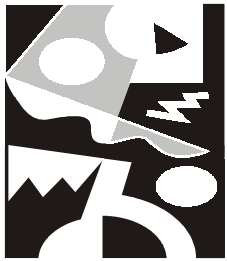
Существует мнение, что если бы не юридические коллизии, отсрочившие выход свободных BSD-систем более чем на год, в разработке Linux не было бы необходимости. Не могу с этим согласиться: если бы Linux не было, его следовало бы выдумать. Потому что мир без него был бы более однообразным. И к тому же, это единственная система, дающая практически каждому пользователю чувство сопричастности к разработке (хотя бы потенциально).

Однако вернемся к нашей берклиаде. Некоторое время оба свободных BSD-клона развивались параллельно каждый в своем направлении. NetBSD портировалась на все новые (точнее, в основном все старые) платформы, тогда как разработчики FreeBSD все более эффективно использовали возможности самой распространенной из них. Однако в 1996 г. произошло первое ветвление в BSD-мире: от NetBSD отделился самостоятельный проект, OpenBSD, в которой мультиплатформенность надстроена ориентацией на максимальную безопасность.

FreeBSD также дала несколько побегов, из которых заслуживает упоминания PicoBSD (**http://people.freebsd.org/~picobsd/picobsd.html**) — система на одной (без преуменьшения) дискете. Тем не менее, она вполне функциональна: имеются варианты сетевой и модемной рабочей станции, сетевого роутера и даже сервера для модемного подключения.

Однако главное ветвление FreeBSD случилось прямо на наших глазах. В середине июня 2003 г. Мэтт Диллон (Matt Dillon) объявил о начале работы над новой ОС семейства BSD — DragonFlyBSD (**http://www.dragonflybsd.org**), отколовшейся от FreeBSD четвертой ветки. Она позиционируется как система общего назначения для серверов и рабочих станций. Однако узкая ее специализация — работа на многопроцессорных машинах. А ведь массовое нашествие двухядерных процессоров на пользовательские машины — не за горами. Впрочем, история DragonFlyBSD пишется сейчас, на наших глазах, и останавливаться на ней здесь было бы опрометчиво.

Глава 4



Почему Linux не Windows

- Linux — это не Windows.
- Почему компьютер — не видеомагнитофон.
- Рецепты против принципов.

Я льщу себя надеждой, что мое сочинение будут читать (в том числе и) совсем начинающие пользователи Linux (и уж совсем несбыточная мечта — что для кого-то это окажется первым действием компьютерной оперы). И потому, начиная с этой главы, позволю себе дать введение в набор базовых понятий POSIX-совместимых систем вообще — своего рода обзор вечных истин, как я их понимаю. И первая из них — осознание специфики POSIX-систем.

Linux — это не Windows

Возможно, мое утверждение покажется вам банальным. Но если, по совету Ходжи Насреддина, вы будете думать над ним неотступно, то проникнетесь его величием и блеском. Итак, изрекаю:

Linux — это не Windows, а Windows — не Linux. И те приемы, что хорошо (эффективно) показывают себя в Windows, отнюдь не обязаны быть столь же действенными в POSIX-совместимой системе. Как, впрочем, и наоборот.

Приведу простой пример. Первейший инструмент Windows-пользователя — это программа, которую в этой ОС (точнее, в русскоязычной ее ипостаси) называют текстовым процессором. На самом деле текстовый процессор в общепринятом понимании этого слова — нечто совсем другое, поэтому впредь программы этого класса будут именоваться визуальными процессорами (принцип WYSIWYG представляет собой их главную отличительную черту) или word-процессорами. Так вот, практически первое, что чуть ли не инстинктивно делает в Linux мигрант из Windows, — это тянется к знакомому пистолету. То есть пытается отыскать среди изобилия Open Sources что-

то, хотя бы отдаленно напоминающее ему Word (WordPerfect, WordPro, Lexicon — ненужное в скобках зачеркнуть).

И к чести Open Source сообщества нужно заметить, что нынче его усилия увенчаются успехом: тройкой программ, вполне знакомых видом и почти таких же нравом, что и привычный ему Word, с функциональностью от идентичной до несколько ослабленной, но в большинстве случаев более чем достаточной: OpenWriter из комплекта OpenOffice.org, KWrite из аналогичного KDE-набора и AbiWord из эвентуального пока офиса для среды GNOME (хотя сам по себе AbiWord — сугубо кроссплатформенное приложение).

Рискну предположить, что вторым по значимости пользовательским инструментом в "подоконной" среде окажется электронная таблица (речь идет о людях, пользующих компьютер в основном для работы). К услугам нашего мигранта оказываются соответствующие средства из тех же офисных пакетов: OpenCalc, KSpread, Gnumeric.

И так далее. При желании он отыщет и графическую среду, внешне сходную с Windows (а при минимальных настройках просто неотличимую), и Explorer-подобный файловый менеджер, и браузер наподобие Internet Explorer, и почтового клиента в диапазоне возможностей от Outlock Express до (почти) The Bat!, и так далее — вплоть до медиаплееров всякого рода и вида. Те, кто не верит, обратитесь к таблице соответствия программ Windows—Linux (строго говоря, Windows—POSIX на **http://www.linuxrsp.ru/win-lin-soft/table-rus.html**).

Хорошо это или плохо для начинающего Linux-пользователя? Конечно, хорошо, скажете вы, и я не смогу с вами спорить, памятуя свои первые шаги в Linux, посвященные лихорадочным попыткам применить в мирных целях тогдашние StarOffice или Applixware. Ведь нынче начинающий пользователь Linux может скрасить свой горький эмигрантский хлеб сладостью знакомых сред и классово близких приложений. Чем он, скорее всего, и воспользуется по полной программе.

Однако скоро у нашего героя зародится мысль: а не напоролся ли он на то, за что боролся? И какой смысл был ему менять уютное и привычное место на "подоконнике" на такое же, только видом сбоку? Ибо OpenOffice покажется ему неповоротливым и тормозным, KOffice — падучим и слабо совместимым, GNOME Office — просто недо-офисом и т. д. Где же обещанная ему мощь UNIX на персональном компьютере?

И постепенно к нему приходит понимание, что мощь UNIX осталась где-то рядом, за пределами мира графических интерфейсов и WYSIWYG-программ — в глубинах командной строки, в буферах текстовых редакторов, управляемых зубодробительными комбинациями клавиш, в непонятных строках скриптов и конфигов. И тогда у него остается два выхода: или бе-

жать обратно, на обжитый "подоконник", как муж возвращается к нелюбимой, но хозяйственной жене от страстной, но безалаберной любовницы. Или все же, если очень нужно, очень хочется, или просто гордость не позволяет возвращаться битым, — стиснуть зубы и начинать работать в bash и vim, искать файлы find и тексты grep, а главное — читать man, info, doc и прочие how to.

Так не лучше было бы для нашего подоконника, если бы кто-нибудь сразу объяснил ему: нет ничего более нелепого, чем ставить Linux ради того только, чтобы сочинять служебные записки в OpenWrite, а финансовые отчеты — в OpenCalc. Или паче того, ради профессиональной обработки изображений в Gimp — на т. е. более подходящие инструменты (и, добавлю по секрету, более подходящие аппаратные платформы). Что сила POSIX-систем для пользователя (о разработчиках или сисадминах тут речи не идет) — в изощренных средствах создания и обработки текстов, а также в мощнейших коммуникационных возможностях. А не это ли, как я уже отмечал в преамбуле, требуется большинству пользователей от компьютера "по делу", а не ради развлечения?

Прошу понять меня правильно: я не призываю отказываться от OpenOffice.org. Более того, я всецело "за": эти средства помогут, помимо относительно безболезненного вхождения в новый дивный POSIX-мир, не чувствовать себя чужими на празднике жизни окрестных "подоконников" с их doc-файлами и Excel-таблицами. Я лишь прошу вас помнить о том самом внешне скромном UNIX-инструментарии, отточенном веками (в масштабах времени компьютерной эпохи) и именно для работы с текстами и коммуникаций.

Помнится, на заре своего приобщения к Linux первое, что я делал после установки системы, — были инсталляция StarOffice и прикручивание к нему русских буковок (тогда это не выглядело столь тривиально, как сейчас). А нынче? Нынче я месяцами не вспоминаю об OpenOffice.org или любом ином офисном пакете — пока не придет doc-файл, который нужно не просто прочесть, но и поправить с сохранением форматирования.

Итак, резюмирую затянувшийся базар. Первое, что должен постигнуть начинающий пользователь POSIX-системы, — ему придется осваивать "вечные истины" POSIX-мира: понятия о файлах, процессах, пользователях, принципы командного интерфейса и т. п. И что, настраивая обои в KDE или стуча по клавишам в OpenOffice.org, он должен морально к этому готовиться. А еще лучше — закрыть глаза и сразу броситься с головой в ледяную воду командных строк и командных редакторов.

Далее, можно сказать, что суть POSIX'ивистского подхода — в извечном противопоставлении и неразрывном единстве дедукции и индукции, анализа и синтеза. И здесь отчетливо проступает академическое происхождение POSIX-совместимых систем: если пользователь Windows в своей повседнев-

ной деятельности руководствуется набором готовых рецептов, более или менее обширным, то эффективное использование Linux или BSD начинается с постижения некоторых общих принципов. Подобно тому, как любая наука начинается, вопреки утверждениям классиков марксистской философии, не с анализа фактов, а с некоторого первичного их обобщения, т. е. синтеза.

Почему компьютер — не видеомагнитофон

В последнее время общим местом стало сопоставление компьютеров с бытовой техникой по простоте использования. Мне такое сопоставление всегда казалось некорректным, и я постараюсь обосновать свое мнение.

Действительно, что такое видеомагнитофон (телевизор, радиоприемник — нужное подчеркнуть)? Это исключительно инструмент для потребления. Потребления продукции, созданной кем-то другим (случай продукции собственной рассмотрим чуть позже). Причем продукции, в отличие от продуктов питания, не жизненно важной, потребляемой исключительно с целью развлечения (случай профессиональных теле- или радиообозревателей здесь не рассматривается).

А потому развлекаемый, если так можно выразиться, пользователь-потребитель вправе требовать от такого инструмента минимума сложностей в использовании. Ну не желает он понимать принципы работы или передачи сигнала на телевизор. Потому как иначе это превратится в работу, а его задача — как раз отдохнуть от своих (в том числе и производственных) проблем за любимой "Великолепной семеркой" или "Тремя мушкетерами" (нужное вписать).

Более того, пользователь-потребитель видеомагнитофона имеет полную возможность обойтись без всяких знаний о его устройстве. Потому как его потребность не забивать голову техническими подробностями сполна удовлетворяется производителями такого рода техники. Иначе ее просто не стали бы массово покупать: вспомним тех же радиолюбителей с паяльниками, много ли их было в процентном отношении?

Так что развлекаемый пользователь вполне может обойтись минимумом самых простых рецептов, как-то: вставить кассету, нажать кнопку "Вперед", после просмотра нажать кнопку "Стоп". Хотя и тут требуется некий минимум подготовки, например: какой стороной кассету засовывать. Иначе возникнет нештатная ситуация, требующая уже дополнительных рецептов (типа использовать деревянную линейку) и дополнительных знаний (на что этой линейкой давить).

Однако все это лишь до тех пор, пока происходит пассивное потребление кем-то созданной продукции. Если же пользователь, насмотревшись передач типа "Сам себе режиссер" или там крутой порнографии, решит создать собственный шедевр в том же духе, ситуация тут же меняется.

Во-первых, ему потребуется инструмент для созидания, а не потребления. Сиречь камера. Во-вторых — умение ею пользоваться. Под которым нужно понимать не только владение ее интерфейсом (грубо говоря, знания тех же кнопок запуска-останова), но и умение снимать. То есть понимание перспективы, освещенности, навыки сочинения какого-никакого сюжета.

А уж если такой пользователь в итоге изберет видеосъемку своей профессией — тут ему потребуется и многое другое. В том числе не лишними окажутся знания о физических принципах фото- и видеосъемки (**http://www.rwpbb.ru**).

Да и на видеомагнитофон такой пользователь (а он ведь по-прежнему пользователь видеомагнитофона по определению, не так ли?) будет смотреть совершенно иначе. Для него это будет уже не инструмент потребления, а аппарат, способный подчеркнуть собственное мастерство (или затушевать отсутствие такового). То есть превратится в деталь производственного цикла, в орудие производства.

И тут требования к удобству интерфейса даже видеомагнитофона отступают на второй план. Мало горя будет в том, что на конкретной модели кнопка запуска расположена не совсем там, где хотелось бы, если это единственный (или единственный доступный) инструмент, способный обеспечить требуемую функциональность.

Вернемся, однако, к рецептам и принципам. Пока пользователь видеотехники остается чистым потребителем, он вполне может обходиться минимумом рецептов. И здесь перед ним два пути: экстенсивный или интенсивный. Первый — копить все те же практические рецепты, наработанные эмпирически. Однако скоро их становится очень много, да и все рецепты по определению охватывают только стандартные ситуации, ничего нетленно-непреходящего с их помощью не создашь. И приходится нашему пользователю волей-неволей обращаться к истокам, т. е. базовым принципам.

Теперь обратимся к компьютерам. В отличие от видеомагнитофонов, они изначально создавались не для потребления, а для креатива (чего бы то ни было). И многими, как ни странно, и по сей день используются, главным образом, для работы. В том числе, а то и в первую очередь — для работы дома. Помнится, меня первая "персональная персоналка" обеспечила именно возможностью не ходить на службу.

Конечно, компьютер имеет и потребительски-развлекательную функцию. Каждый пользователь, профессионально с компьютером связанный (выполняющий свою профессиональную работу главным образом на компьютере),

отнюдь не прочь послушать музыку или посмотреть кино без отрыва "от станка". Однако функция эта не просто вторична по времени, она не критична и по значению. Думаю, если просмотр видеоролика будет мешать работе профессионально критичной программы, любой профессионал предпочтет смотреть его по видику (тому самому, развлекательно-потребительскому).

Очевидно, что для профессиональной работы важнее функциональность, а не удобство использования (мне безразлично, насколько удобно я не могу выполнить свою задачу). А развлекательная составляющая становится вообще вроде бесплатного приложения.

Это я все к тому, что даже и в развлекательном аспекте компьютер остается где-то инструментом креативным, со всеми вытекающими последствиями. Кроме того, он при этом практически не теряет универсальности. И потому ожидать, что он будет так же прост в обращении, как монофункциональный развлекатель — по меньшей мере, излишне оптимистично.

Конечно, чисто развлекательную функцию компьютера тоже можно упростить до состояния видеомагнитофона. Замечательным примером чему служит Linux-дистрибутив под названием MoviX. Это один из LiveCD, т. е. система на компакт-диске, способная с него не только запускаться, но и полноценно функционировать. Функции MoviX, правда, весьма ограничены. А именно: он умеет только крутить мультимедийные файлы. Но зато умеет это очень хорошо. И главное, ничуть не сложнее, чем бытовой агрегат соответствующего назначения.

Так что достаточно легкого движения рук — вставки диска MoviX в привод и комбинации из трех пальцев, чтобы волшебным образом превратить тысячедолларовый компьютер в элегантный видеомагнитофон или CD-плеер красной ценой в пару сотен. Благо, и обратное превращение ничуть не сложнее...

Но упростить производственную-то функцию компьютера все равно не удастся (работать вообще довольно трудно, как говорил, если не ошибаюсь, Антон Павлович Чехов). И потому лозунги типа: "С выходом Windows 3.0 (3.1, 95, 98, ME, XP и т. д.) обращаться с компьютером наконец-то стало также просто, как с бытовой техникой", которые я лично слышу уже более 10 лет, лукавы, как минимум, вдвойне. Во-первых, это самая обычная подмена понятий, если под обращением понимать не только развлекательную сторону, но и производственную (см. вышеуказанный тезис А. П., создавать видеофильмы никогда не будет также просто, как их просматривать). А во-вторых, сама регулярность появления таких лозунгов вызывает подозрения, что и с развлекательной строной компьютеров все еще сохраняется некоторая напряженка.

Рецепты против принципов

Однако опять вернемся к рецептам и принципам. Худо-бедно, но с развлечениями на компьютерах можно справиться посредством первых. А вот с производством любого рода? Рассмотрим это на примере более-менее близкой мне области — работы с текстами, претендующими на оригинальность (то есть выдумываемыми из головы).

Подобно нашему видеолюбителю, профессиональный текстовик, пересев с пишущей машинки (или с письменного стола со стопой бумаги и паркеровской авторучкой) за компьютер, быстро узнает множество простых рецептов, как-то: нажав клавишу <Insert>, он может забить неправильно введенный текст, клавишами <Delete> или <Backspace> можно уничтожить лишнюю букву (в отличие от замазки, без следов) и т. д.

Жить нашему текстовику становится лучше, становится веселей. Но еще не до конца. Потому как он узнает об управляющих последовательностях (с помощью которых может мгновенно переместиться в требуемое место текста и продолжить набор-редактирование), о глобальном поиске и замене, об автоматической проверке правописания и о многом, многом другом.

Однако суть работы текстовика при этом не меняется, как не меняется и стиль мышления. И то, и другое по-прежнему линейно, оригинальный текст создается от начала и до конца, как и за пишущей машинкой, расширяются только возможности возврата к написанному и внесения в него корректив. И потому следующая мысль — а не структурировать ли текст изначально, внеся соответствующую разметку рубрик, подрубрик, параграфов? И вот это — уже скачок качественный, ведь для структуризации (ненаписанного еще) текста последний весь, целиком, уже должен быть вот здесь, в ... (ну, сами знаете где).

К слову сказать, современные word-процессоры WYSIWYG-типа пользователя к такой структуризации отнюдь не стимулируют. За ненужностью, вероятно. В одной книжке про Word мне как-то встретилась фраза, что стилевая разметка — это штука очень сложная, которая по силам только высоким профессионалам. Простым людям, видимо, проще вручную придавать заголовку каждой главы кегль и начертание.

Впрочем, я опять отвлекся, перестройка мышления текстовика — тема совершенно отдельная. Вернемся к принципам. С созданными и отредактированными текстами подчас приходится продолжать работать: делить на фрагменты, соединять, извлекать части одного документа и вставлять в другой. И вот тут-то и обнаруживается неэффективность рецептов.

Возьмем простую задачу — создание единого документа из нескольких существующих, причем включенных в определенной последовательности. Мне этот пример кажется очень показательным, и я не устаю его повторять.

Можно открыть первый документ, перейти в его конец, щелкая мышью по контекстному меню (или даже воспользовавшись существующим рецептом — макрокомандами с привязанными к ним горячими клавишами), вставить туда второй документ и т. д. Быстрее, конечно, чем подклеивать бумажные листы силикатным клеем, но все ли это, что может компьютер?

Нет, если узнать (или вспомнить) несколько принципов более общего порядка. Любой документ есть файл. Любой файл может быть выведен на устройство вывода (экран или, скажем, принтер). А любой вывод может быть перенаправлен с одного устройства вывода на другое. Но ведь любое устройство вывода — тоже файл. Значит, вывод любого файла может быть перенаправлен не в файл устройства, а в другой, например, текстовый, файл. Остается только отыскать команду, которая это сделает. И такая команда легко находится — это `cat`. В результате конструкция

```
$ cat file1 file2 file3 > file-all
```

создаст нам результирующий документ в один присест. Причем составные части его расположатся в той последовательности, в какой нам нужно — в соответствии с порядком аргументов команды `cat`. И если мы заранее озаботились тем, чтобы структура наших рабочих файлов хоть как-то коррелировала со структурой итоговой работы (а это та самая структуризация мозгов, о которых я говорил чуть раньше), результирующий документ будет структурирован должным образом, причем без всяких дополнительных усилий.

Задача обратная — поделить наш правильно структурированный документ на отдельные части в соответствии с его внутренней структурой (например, разбить книгу на главы, как это обычно требуется для представления в издательство). Для автоматизации процесса нам достаточно знать о другом относительно общем понятии — регулярных выражений, причем лишь в той его части, которая описывается термином шаблон (pattern). И задача сводится к тому, чтобы отыскать в нашем файле строки, начинающиеся последовательностью символов "`Глава`", и каждую последовательность символов в промежутке записать (т. е. вывести) в самостоятельный файл. Что можно сделать многими способами, но один из них, штатный и очень простой, это команда `split` (в BSD) или `csplit` (в Linux).

Последний пример показывает, что хотя понимание принципов и не избавляет совсем от обращения к рецептам, но зато позволяет вычислить последние. Дело в том, что в BSD команда `split` универсальна и служит для разделения файла по любому параметру: размеру, номеру строки или шаблону. Одноименная же команда в Linux выполняет только первые две функции, опции `-p` (`--pattern`) в ней не предусмотрено. Что поначалу может расстроить. Однако если понимать, что, в принципе, разделение файлов по шаблону почти ничем не отличается от такового по номеру линии или раз-

меру (и то, и другое суть действия, основанные на анализе последовательности символов), остается только изыскать соответствующий рецепт. Что можно сделать просто в лоб — поискать слово "`split`" в каталоге с man-страницами командой `grep` (строго говоря, не слово, а последовательность символов, и командой `zgrep`, т. к. страницы эти обычно в gzip-сжатом виде). Чем и обнаруживается man-страница с описанием команды `csplit`, прочитать которую — уже вопрос элементарной грамотности.

Следующий пример соотношения рецептов и принципов касается установки пакетов. Пользователь любого прекомпилированного дистрибутива быстро усваивает простые рецепты управления оными. Таким рецептом в rpm-based дистрибутивах Linux является команда вида

```
$ rpm -ihv package_name.rpm,
```

прекрасно работающая в штатных ситуациях. Однако что делать, если пакет таким образом не устанавливается (например, вследствие нарушения зависимостей), работает не так, как ожидалось, или просто отсутствует? На помощь приходит понимание принципов распространения свободных программ (в исходных текстах) и принципов их сборки, универсальных и не зависящих ни от дистрибутива, ни даже от операционной системы. А понимание принципов сборки, в свою очередь, способствует проникновению в суть дистрибутив-специфичного пакетного менеджмента...

Но особенно явно превосходство принципов над рецептами выступает при конфигурировании всего и вся, от общесистемных опций до шрифта меню конкретного приложения. Рецептурный подход — использование специализированных средств настройки, оформленных в виде самостоятельных утилит или встроенных в прикладные программы. Самостоятельные утилиты такие многочисленные и разнообразные, не зря Владимир Попов как-то заметил, что число утилит конфигурирования давно превзошло количество конфигурируемых параметров. Интерфейс у них разный в разных дистрибутивах, да к тому же еще и может меняться от версии к версии. Так что доскональное знание какого-либо DrakX из Mandrake ничем не поможет при работе с sysinstall из FreeBSD, и наоборот.

Если же не поступаться принципами, достаточно раз и навсегда понять, что все параметры настройки системы описываются в соответствующих конфигурационных файлах, которые суть обычные тексты, могущие быть открытыми в любом текстовом редакторе и там модифицированными надлежащим образом. Что и проделывают, только в завуалированном виде, все настроечные утилиты.

Что касается объектов конфигурирования, то они определяются стандартными утилитами работы с текстами (например, командой `find` для поиска файлов по маске и командой `grep` для изыскания в них подходящих по

смыслу фрагментов). Ну и, разумеется, осмысленного анализа результатов того и другого...

Вопреки сложившемуся убеждению, для этого не обязательно быть UNIX-гуру или к таковому обращаться. Во-первых, тот же гуру даст, скорее всего, конкретный рецепт на злобу дня. Во-вторых, многие вещи в системе настраиваются один раз в жизни, и вполне возможно, что наш гуру благополучно забыл о том, как именно он это делал. Так что своей просьбой вы просто вынуждаете его вторично проделать ту самую цепочку логических рассуждений, отталкивающихся от общих принципов, которую вы легко (и с пользой для духовного самосовершенствования) могли бы проделать сами.

И еще к слову: с успехом (надеюсь) применяя принципиальный подход к жизненно важным для работы настройкам, можно не поступаться принципами и при настройке вещей развлекательного свойства. Например: звуковая карта определена и в ядре настроена правильно, соответствующий софт установлен и работает, но mpeg-файлы, скажем, воспроизводиться не желают.

Вероятно, в user-ориентированных дистрибутивах существуют какие-нибудь специальные утилиты для настройки этого хозяйства, и можно обратиться к ним. А можно просто вспомнить, что звук воспроизводится устройством, устройство есть файл, а файл имеет определенные атрибуты принадлежности (хозяину, т. е. пользователю, группе и всем прочим) и атрибуты доступа (в просторечии именуемые правом чтения, исполнения и изменения). И остается только проверить, а имеет ли данный пользователь должные права доступа к этому файлу? И если выясняется, что файл /dev/audio открыт для всеобщего использования во всех отношениях, посмотреть, а не есть ли его имя лишь символическая ссылка на файл реального устройства, отвечающего за воспроизведение звука, и проверить права доступа к нему.

И опять к слову: понятие атрибутов принадлежности и доступа, одно из краеугольных в UNIX-системах, существует и в Windows. Да вот только пользователи об этом часто не подозревают. Не потому, что чайники, а потому, что их от этого знания тщательно оберегают.

В результате к нештатным ситуациям (например, потере пароля — а кто от этого застрахован, все мы люди) пользователи Windows оказываются просто морально не готовы: нужно дергаться, даже (страшно подумать) лезть в книги (не для того ли мы отказывались от man- и info-страниц?) для поиска рецептов, соответствующих ситуации.

А в UNIX все просто, если помнить о файле (файлах) паролей, однопользовательском режиме (или возможности загрузки с внешнего носителя) и о том, что дисковые устройства нужно монтировать.

В общем, подведу итог. Рецептурный подход вполне приемлем при потребительско-развлекательных задачах. И оказывается, мягко говоря, не самым эффективным при задачах производственно-креативных. А отработав принципиальный подход на них, становится уже лень искать, какая кнопочка отвечает за масштабирование окна данной программы воспроизводства видео... Проще задать сиюминутную геометрию в командной строке или (раз и навсегда) в соответствующем конфигурационном файле.

В преамбуле я вскользь упоминал, что FreeBSD видится мне более подходящей системой для приобщения к POSIX'ивизму, нежели какой-либо из user-ориентированных дистрибутивов Linux. И теперь я могу высказать первый к тому резон: отличие этой ОС от Windows с первых же шагов проступает столь явственно, что не рождает даже мысли об использовании "подоконных" приемов работы.

Конечно, такие Source Based дистрибутивы Linux, как Gentoo, уже на стадии установки оставляющие пользователя наедине с командной строкой и текстовым редактором, еще более гармонируют с методом "большого болота". Однако для совсем уж неподготовленного болото может оказаться чересчур уж глубоким.

FreeBSD же предлагает начинающему пользователю достаточно взвешенный подход: на первых порах можно положиться на умолчания системы, не идеальные, но разумные. А в дальнейшем, по мере накопления знаний и умений, все больше и больше вмешиваться руками.

Тем не менее, основной десктопной платформой среди открытых ОС выступает Linux. И среди Linux также можно подобрать подходящие (можно сказать, модельные) объекты для постижения истин POSIX-мира. Многие поколения пользователей входили в этот мир через Linux Slackware. Однако в настоящей книге в качестве таких модельных систем выбраны два близкородственных дистрибутива: Archlinux и CRUX.

Может возникнуть вопрос: почему я остановился на относительно мало распространенных представителях этого семейства?

Во-первых, концепция их построения, в частности, обособление базовой системы от опциональных (и сугубо альтернативных) пользовательских приложений, подчеркивает системную целостность Linux как ОС.

Во-вторых, как это ни парадоксально, оба эти дистрибутива, казалось бы, отнюдь не ориентированные на начинающего пользователя, весьма просто устроены — почти так же просто, как и FreeBSD.

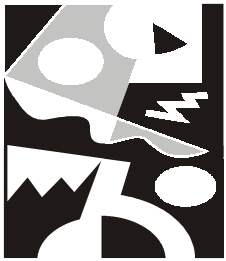
В-третьих, Archlinux и CRUX, несмотря на близкое (можно сказать, генетическое) родство, демонстрируют разный подход к дистрибутивостроению вообще. Первый — это полнофункциональная система, включающая необходимый набор пользовательских приложений (сохраняющая, тем не менее,

относительную компактность и обозримость), развертываемая за считанные минуты и позволяющая перейти к практической работе после нескольких несложных манипуляций. CRUX же — это скорее база для построения системы собственной, хотя и тут процесс можно осуществлять параллельно с выполнением пользовательских задач.

Наконец, я просто хотел привлечь внимание к этим достойным представителям Linux-семейства, отнюдь не пользующимся известностью, причем незаслуженно.

Я вовсе не навязываю свой выбор в качестве истины в последней (да и в любой другой) инстанции. Однако с чего-то начинать нужно, почему бы не начать с одной из перечисленных систем? Тем более, что начинающий POSIX'ивист должен четко понимать: выбрать с первого же раза дистрибутив или ОС, идеально подходящий к его задачам, потребностям, наконец, просто индивидуальным склонностям — можно только при исключительном везении. Скорее всего, ему придется перепробовать не одну систему, прежде чем окончательно определиться в своих предпочтениях. Однако полученные навыки и знания лишними не окажутся в любом случае.

Глава 5



Как научиться плавать: установка системы

- Подготовка к заплыву.
- Загрузка и запуск.
- Подготовка диска.
- Установка.
- Обеспечение загрузки.
- Обеспечение работы в графическом режиме.
- Особенности установки BSD-систем.
- Проблема выбора.

Итак, цели ясны, задачи определены — за работу, товарищи. Однако, чтобы научиться плавать, нужно лезть в воду, чтобы научиться работать в POSIX-системе, нужно начать работать в какой-либо их этих систем.

Подготовка к заплыву

Банальность этого утверждения осложняется одним моментом. Прежде чем начать работать в Linux или BSD, любую из этих систем необходимо установить. Поскольку спасение утопающих — дело рук самих утопающих, то устанавливать ее придется, скорее всего, тому самому пользователю, в планы которого и входит изучение UNIX. А вот тут оказывается, что сама по себе установка этой системы предполагает некоторые навыки обращения с ней.

На самом деле все не так. И современные ориентированные на конечного пользователя дистрибутивы Linux можно установить столь же просто, как и Windows. Именно для этого и предназначены программы-инсталляторы, определяющие специфику каждого дистрибутива и являющиеся предметом

гордости для их создателей, объектом восхваления или ругани — для пользователей.

Лучшие инсталляторы из хорошо сделанных дистрибутивов Linux позволяют установить систему буквально десятком щелчков мыши, потребовав предварительно лишь ответа на несколько вопросов. А то и на один-единственный: не желает ли пользователь проделать все на полном автомате? Или предлагают варианты ответов, один из которых считается наиболее подходящим. Известная шутка про Debian (установить его может и цыпленок, достаточно научить его клевать клавишу <Enter>) оказывается подчас не столь уж и далекой от истины...

Но тут возникает другая проблема: эти самые замечательные инсталляторы в своей неустанной заботе о благе пользователя настолько маскируют от него суть своих действий, что пользователь остается в полном недоумении: а что, собственно, они делают. И каким таким волшебным образом на винчестере, который только что был девственно чист, возникает операционная система, да еще и с многочисленными приложениями на 2—4 Гбайт. Функции их также не всегда понятны, взаимосвязи не известны, необходимость может показаться сомнительной. Оказывается, что в автоматическом или полуавтоматическом (от поклевывания <Enter>) режиме система установлена не совсем так, как этого хотелось бы пользователю.

Разумеется, не все дистрибутивы столь заботливы, некоторые требуют от пользователя понимания сути совершаемых действий. И в них пользователю предоставляются широкие возможности для вмешательства в процесс установки. Предельный случай — дистрибутив Gentoo Linux, инсталлятора просто не имеющий: весь процесс установки выполняется прямыми командными директивами. Однако начинающему пользователю это может показаться сложным, да так на самом деле и есть. Ведь и плавать обычно учатся пусть на северном берегу, но Черного моря, а не на южном — Баренцева...

Так что выбор в качестве первой пробы пера какого-либо user-ориентированного дистрибутива вполне оправдан. Однако понимание сути действий установщика любой дружественной системы от этого не становится менее важным. Тем более, что все они, в сущности, делают одно и то же.

Что именно? А давайте подумаем, что нужно сделать, чтобы установить ОС.

Для начала примем как данное, что установщик Linux — просто некая программа. Правда, как ни странно, работающая под управлением устанавливаемой им же ОС — Linux же. Каковой на винчестере пока не имеется.

1. То есть первый шаг к установке — загрузка с какого-либо внешнего носителя и запуск программы-инсталлятора.
2. Далее, кроме установочного носителя (в его качестве ныне обычно выступает компакт-диск), неплохо иметь накопитель, на который новая ОС будет устанавливаться. Таковым в большинстве случаев является жесткий

диск. Причем этот накопитель нужно подготовить неким определенным образом, доступным для понимания устанавливаемой системой. То есть, в терминах DOS/Windows, разметить (или разбить на разделы) и отформатировать.

3. Третий шаг — перенос с установочного носителя на диск собственно системы и всего, что ей (и пользователю) необходимо для счастья, — интуитивно понятен. Как понятно и то, что, вне зависимости от инсталлятора, этот процесс сводится к развертыванию архивов с CD и копированию их содержимого на винчестер. А то и просто к банальному копированию содержимого установочного диска на целевой носитель. Хотя нет, на установщик возложена еще одна, и очень важная, функция — обеспечение контроля зависимостей между пакетами.
4. Наконец, последнее — это обеспечение загрузки установленной системы после рестарта машины.
5. Хотя нет, есть еще и пятый шаг — это настройка графического режима работы. Сама по себе она ни к Linux, ни к любой BSD никакого отношения не имеет, однако выступает непременным атрибутом установщика любого дистрибутива, претендующего на роль user-ориентированного.

Многие инсталляторы имеют еще и дополнительные возможности: автоматического определения оборудования на целевой машине, более или менее автоматизированных настроек и т. д. Однако четыре перечисленных шага (инсталляторы дистрибутивов, не посягающих на user-ориентированность, настройки графики не предусматривают) делают в обязательном порядке все инсталляторы любых Linux-дистрибутивов. Вот и рассмотрим каждый из этих шагов поподробнее, для определенности, на примере установки Linux.

Загрузка и запуск

Загрузка Linux для установки этой системы осуществляется с компакт-дисков: времена загрузочных дискет, надеюсь, остались в далеком прошлом (ныне дискеты могут понадобиться только в особых случаях). Первый (а иногда и не только первый) диск любого дистрибутивного набора — загрузочный, и для старта достаточно вставить его в соответствующий привод, перезагрузить машину (тремя пальцами или одним — по ситуации), в ходе перезагрузки выставить в BIOS Setup соответствующую опцию (загрузка с CD) и ждать завершения процесса (что может занять не одну минуту).

Первое, что делает загрузчик Linux, — это загружает Linux, ту часть системы, которая называется ядром (некоторые считают, что ядро это и есть Linux, однако, как вы поняли из *главы 2*, я с этим не согласен). Ядро ОС

отвечает за взаимодействие всех остальных программ с "железом" целевой машины (в самом широком смысле слова), без такого взаимодействия никакие дальнейшие действия невозможны.

Со временем мы увидим, что ядро Linux (как и любой BSD-системы) может быть собрано (сконфигурировано) самыми различными способами, в зависимости от возможностей (т. е. того же наличного "железа") и потребностей (назначения системы). Для ядра, обеспечивающего установку, главное — это поддержка максимально широкого спектра оборудования из числа наиболее распространенного, ведь создатели дистрибутива заранее не знают, на какие машины его придется устанавливать пользователям.

Диагностика наличного "железа" также входит в компетенцию ядра и выполняется в ходе его загрузки. Ядро не скрывает результаты своей работы в этом направлении, именно об обнаруженных устройствах и информирует оно пользователя в сообщениях, мелькающих во время старта машины. Правда, разглядеть их сложно, но это не беда — с результатами диагностики можно будет ознакомиться и потом (впоследствии мы узнаем, как именно).

Не все оборудование критически важно определить на стадии установки. Очевидно, что к такому относятся: носитель дистрибутива, целевой накопитель (рискну предположить, что им будет винчестер) и его контроллер, память, как с точки зрения количества, так и качества — часто именно при инсталляции Linux выявляются проблемные ее модули. Большинство установщиков user-ориентированных дистрибутивов по умолчанию запускаются в графическом режиме, и потому немаловажным моментом диагностики является определение возможностей видеокарты и монитора. А поскольку графический режим установки почти немыслим без мыши, это устройство (вместе с клавиатурой) также неожиданно оказывается в списке критически важных.

Какие подводные камни могут встретиться на начальной стадии загрузки? Не так и много. Приводы CD-ROM и винчестеры уже давно столь стандартны, что ждать здесь осложнений не приходится (о некоторых возможных проблемах я скажу чуть позже). Вопрос недостатка памяти — актуальность практически потерял (обычных ныне 256 Мбайт с лихвой хватает для запуска любого инсталлятора).

Видеосистема? Здесь, конечно, неожиданности возможны. Однако разнообразие видеокарт осталось в прошлом, современные же видеочипы (благо, в каждый момент времени они известны наперечет), как правило, установщиками распознаются нормально (хотя и не всегда идеально). Для неизвестных же инсталлятору мониторов (т. е. не присутствующих в его базе данных или не определенных им автоматически) обычно выставляются минимально возможные частотные характеристики — страшилки о сгоревших вследствие завышения значений развертки мониторах отошли в область преданий.

Еще бывают проблемы с USB-мышами и, особенно, клавиатурами. Однако и эти проблемы отходят в прошлое. Правда, если они все же возникнут (как в некоторых версиях FreeBSD с USB-клавиатурами), решить их можно только временной (на период установки и начальной конфигурации) заменой престарелым аналогом.

Наиболее вероятные в настоящий момент грабли — это ATA RAID-контроллеры и, возможно, контроллеры SATA. Ныне ими оснащаются чуть не все материнские платы, а вот с поддержкой их в Linux (и особенно установочными ядрами дистрибутивов) дело не всегда обстоит лучшим образом.

Не могу не воспользоваться случаем и не молвить пару слов во славу FreeBSD: начиная с пятой ее ветки, проблем с контроллерами ATA RAID почти не бывает. А вот NetBSD, напротив, до недавнего времени знать о них не знала и не желала (насколько мне известно, начиная с версии 2.0 это, наконец, изжито). OpenBSD тут занимает промежуточное положение: диски на контроллерах ATA RAID она обычно распознает, но работать может только как со стандартными IDE-винчестерами.

Что делать в случае осложнений с "железом"?

Во-первых, сверить наличное оборудование со списком поддерживаемого данным дистрибутивом. Впрочем, это лучше делать до приобретения того или другого: если у производителей такого рода информацию раздобыть трудно, то она обычно доступна на сайтах разработчиков дистрибутива.

Во-вторых, внимательно ознакомиться с комплектацией дистрибутива. Многие из них оснащаются не одним ядром, а несколькими, в том числе и для поддержки всякого экзотического оборудования. Причем подчас (и здесь доброго слова заслуживает Slackware) с тщательно прокомментированными конфигурационными файлами, легко позволяющими понять, какое ядро для какого "железа" предназначено. Правда, не факт, что такие дополнительные ядра могут стартовать непосредственно с диска, тут-то и придется повозиться с загрузочными дискетами. Благо, процедура их создания под DOS/Windows описана в большем числе документов, чем возникает случаев необходимости обращения к ним.

В-третьих, сменить дистрибутив. Возможно, достаточным окажется взять более свежую версию того же наименования — поддержка нового оборудования, за некоторыми печальными исключениями, добавляется в Linux достаточно быстро.

Если же и это не помогло, остается только сменить "железо". Причем, возможно, только на время установки. Потому что всегда следует помнить: отсутствие поддержки какого-либо устройства на стадии установки отнюдь не означает невозможность его работы в Linux вообще. И вполне возможно, что видеокарта, оставшаяся неопознанной инсталлятором, будет благопо-

лучно настроена в дальнейшем. Так что главное — установиться, а там видно будет.

Практически единственное препятствие для установки, с которым я сталкивался, — те же ATA RAID. Да и то только в том случае, если с подключенных к ним дисков предполагается загрузка, или они должны нести корневую файловую систему. Иначе весьма велика вероятность, что диск на дополнительном контроллере ATA RAID, не видный при установке, можно прикрутить к системе, должным образом сконфигурировав и пересобрав ядро. Эта проблема имеет давнюю историю, и существуют обходные пути ее решения, однако задача эта — не для начинающего пользователя. Да и труда часто не стоит. Так что, возможно, что тут просто уже придется браться за отвертку и переставлять винчестер в основной IDE-разъем...

Если с базовыми компонентами машины все в порядке, происходит запуск собственно инсталлятора. Однако перед этим во многих дистрибутивах выполняется (уже не ядром, а отдельной программой — в Linux ею обычно является kudzu) диагностика оборудования, не критичного для установки, но весьма важного для пользователя: звуковой и сетевой карт, принтера, модема, сменных накопителей и т. д. И если оно будет опознано правильно, есть шанс получить их поддержку сразу по завершении установки.

Однако вернемся к инсталлятору. Как я уже говорил, обычно это более или менее красивая и удобная программа графического режима. Однако в случае затруднений с видеокартой может быть предложена установка в режиме текстовом. Чего пугаться не следует — функциональность установщика от этого (почти) не изменится, разве что все будет не так красиво...

Первые вопросы программы-установщика могут быть разными: здесь обнаруживаются и выбор языка, и уточнение типа и интерфейса мыши (хотя автоматика тут обычно справляется), и точное указание видеорежима, в котором будет происходить установка (при правильном опознании видеокарты, разумеется). Ответы на них обычно достаточно очевидны и могут даваться из соображений здравого смысла.

Единственная тонкость касается как раз языка. В одних системах (примером — ASPLinux, **http://www.asplinux.ru**) язык установки (предположим, русский) никак не связан с базовой русификацией системы (которая настраивается потом и отдельно). В других же, напротив, избежать последующей ручной русификации можно, только выбрав русский язык как используемый при установке (Mandrake, Altlinux). При этом подчас доходит до смешного: выбор русского языка при установке в некоторых ситуациях делает затруднительным ввод латиницы (и, как станет ясным из следующей главы, вход в систему) сразу после перезагрузки. Так что читайте документацию, ибо даже и user-ориентированная система может здорово спрашивать...

Подготовка диска

Как бы то ни было, промежуточная стадия после загрузки рано или поздно заканчивается. И наступает второй этап большого пути — подготовка диска. Это понятие включает три компонента: разбиение диска на разделы, создание файловых систем (или, как говорят в DOS/Windows, форматирование), монтирование в единую файловую иерархию. На сути их останавливаться не будем, подробному рассмотрению этих действий будут посвящены специальные главы (*глава 9* и *глава 10* соответственно). А пока лишь несколько практических рецептов.

Первое и самое важное — выбор режима подготовки диска. Их в большинстве установщиков предусмотрено три: полностью автоматический, полуавтоматический и (почти) ручной. И если у вас на диске уже установлена какая-либо другая ОС (берусь с двух раз угадать, какая) и (или) имеются данные, которые нужно сохранить, следует быть предельно внимательным (и читать все выводимые сообщения, а при необходимости не гнушаться и встроенной помощи).

Впрочем, забыл сказать о самом главном: если Linux устанавливается не на "чистую" машину, вся информация на диске, о потере которой вы будете сожалеть, должна быть в обязательном порядке сохранена на резервных носителях (быть может, именно установка новой ОС подвигнет вас на так долго откладываемый backup?). Потому что сама по себе процедура подготовки диска практически безопасна для наличной информации, при соблюдении сказанного в прошлом абзаце и с учетом того, что скажу в следующем. Однако любые аппаратные сбои, вплоть до отключения света, будут иметь необратимые последствия.

Вообще говоря, для первой установки я рекомендовал бы именно чистый диск, пусть и не первый (Linux в силах грузиться как с Master, так и со Slave на любой IDE-линии, а также с дисков SATA).

А возвращаясь к режимам установки, следует вспомнить и божественного Гомера: "Бойтесь данайцев, дары приносящих". В данном случае — полностью автоматического режима, хотя часто именно он рекомендуется как самый подходящий для начинающего. Почему? Да потому, что, как правило, автоматическая подготовка диска подразумевает, что разделы под Linux будут созданы по велению и хотению инсталлятора, без всякого учета наличной информации. Попросту говоря, все содержимое диска (установленная ранее ОС и ее данные) может быть уничтожено. Причем далеко не всегда предупреждение на этот счет покажется вам достаточно внятным.

Так что пользователям, и особенно начинающим, прибегать к автоматическому режиму следует только в случае "чистого" диска. Да и то, я бы воздержался от него без веских причин (острого дефицита времени, скажем).

Потому что именно начинающему пользователю весьма полезно познакомиться с процедурой дисковой разметки даже в том скудном объеме, в каком это допускают заботящиеся о нем установщики. И вообще, по моему скромному разумению, все автоматические режимы инсталляторов (это касается и дальнейшего, например, выбора пакетов) оправданы как раз не для начинающих пользователей, а для весьма искушенных (в конкретном дистрибутиве) системных администраторов. Знающих точно, когда на автоматику можно положиться, а в каких случаях следует вмешаться руками.

Сказанное относится и к режиму полуавтоматическому. Под ним обычно понимается то, что разделы для установки Linux будут созданы установщиком также по своему разумению, но не на всем диске, а только на его неразмеченном пространстве. Подчеркну, именно неразмеченном (если оно имеется, конечно), а не просто свободном от данных. Так что опасности для существующей информации этот режим обычно не представляет. Но и самообразованию пользователя он также не очень способствует.

И потому в качестве основного режима будем рассматривать тот, который называется ручным, заказным или каким-либо иным, близким по смыслу к эвфемистическим выражением. По его выборе пользователю предлагается самому определить, на сколько разделов будет разбит диск, какое пространство следует отвести под каждый, какие на них будут созданы файловые системы и куда они будут смонтированы. В принципе, все это вопросы довольно сложные и потому будут рассмотрены особо. Пока же наша цель — принять практическое решение.

В значительной мере решение это будет определяться тем, что нам предложит установщик. А их модули управления разделами и файловыми системами обычно предусматривают весьма широкий спектр возможностей. Очевидно, что среди них есть возможность удаления существующих разделов и создания новых: иначе как нам освободить место под Linux на полностью разбитом диске. В некоторых дистрибутивах возможно изменение существующих разделов без потери информации.

Освободив место под новые разделы, можно приступать к их созданию. Подробно стратегия разбиения будет рассмотрена в одной из последующих глав. Пока же важно запомнить, что обязательным считается наличие двух разделов: корневого и раздела подкачки, к которым я настоятельно рекомендую добавить еще и третий — для пользовательских данных.

Корневой раздел в первом приближении будем считать предназначенным для системы в целом (в дальнейшем мы увидим, что некоторые ее части целесообразно вынести на отдельные разделы, но пока голову на сей предмет ломать не будем). Размер его, собственно говоря, определяется намерениями на следующем этапе установки, однако чисто на вскидку определим его в диапазоне 2—5 Гбайт.

Раздел подкачки (swap-раздел) предназначен для обмена с оперативной памятью, в него выгружаются, фигурально говоря, давно не использовавшиеся фрагменты программ и данных. Рекомендуемый для него размер равен удвоенному объему системной памяти. Насколько такая рекомендация оправдана в настоящее время — вопрос спорный, однако и этим пока не будем заниматься: если не знаешь, как делать, делай, как приказано (рекомендовано в документации).

Величина раздела для пользовательских данных задается просто:

- по остаточному принципу;
- по принципу "сколько не жалко";
- столько, сколько нужно (если, конечно, вы в силах предугадать это).

И в любом случае я оставил бы некоторую часть дискового пространства: если по прочтении соответствующей главы (или в процессе последующей работы) обнаружится, что диск разбит не оптимальным образом, будет некоторая свобода маневра.

А еще установщик может предложить выбор, создавать ли ему разделы первичные или логические в расширенном разделе (формулировки тут могут быть самые разные и подчас не всегда понятные). Для Linux самого по себе это несущественно — требование непременной первичности для корневого раздела давно потеряло силу. Однако зависит это от конфигурации разделов ранее существующих (и тех, которые хотелось бы сохранить неприкосновенными после установки). Ибо первичных разделов на диске IBM-совместимой машины может быть не более четырех, и практически только один из них может быть объявлен расширенным, пригодным к разбиению на разделы логические.

Поскольку вопрос этот достаточно сложен, он также послужит предметом специального разбирательства. Пока же замечу только, что раздел для домашнего каталога я, по возможности, сделал бы первичным: это даст необходимую гибкость при смене дистрибутива, каковая, как будет показано далее, почти неизбежна в ходе становления начинающего POSIX'ивиста...

И последнее. Возможно, установщик будет настойчиво рекомендовать небольшой раздел для использования в качестве загрузочного. Если так, по возможности соглашайтесь, не пожалеете. И размер его определите в 30—50 Мбайт (хотя, например, Fedora откажется устанавливаться, если этот раздел будет меньше 70 Мбайт).

Большинство установщиков user-ориентированных дистрибутивов тесно сопрягают разбивку диска с форматированием и монтированием. То есть по создании раздела предлагается выбрать для него тип файловой системы и определить точку ее монтирования. Linux в качестве родных (native) поддерживает множество типов файловых систем: классическую Ext2fs и ее усовершенствованную разновидность — Ext3fs, ReiserFS, XFS, JFS.

И все они могут быть предложены инсталлятором на выбор. Так что вопрос этот весьма сложен, но поначалу, до формирования устойчивых личных предпочтений в этой области, можно согласиться с тем, что предлагается по умолчанию. С монтированием же очевидно, что корневой раздел должен быть смонтирован в точку корня (/), раздел для данных — в точку /home, загрузочный — в точку /boot. А раздел подкачки в монтировании не нуждается...

И еще один важный момент: устройство инсталляторов обычно таково, что собственно разбиение диска и форматирование разделов (т. е. действия необратимые, после которых откат к старому состоянию диска уже невозможен) выполняются не в момент нашего выбора, а несколько позже. И потому при ошибке можно (до некоторой критической точки) отказаться от сделанных изменений и начать все сначала. Впрочем, и тут можно дать только один совет: внимательно следите за сообщениями программы установки, обычно она выдает достаточно грозные предупреждения перед свершением непоправимого...

Установка

Третий этап установки Linux — это собственно установка, т. е. развертывание компонентов системы и помещение их на подготовленный плацдарм — смонтированные файловые системы на дисковых разделах. Однако этому предшествует стадия выбора компонентов, подлежащих установке.

Дистрибутивы Linux организованы по пакетному принципу. Пакет — это атом Linux-системы, наименьшая часть, на которую ее можно разделить. Так что без сильнодействующих технических средств пакет может быть добавлен в систему только целиком и также целиком удален.

Это не значит, что пакеты маленькие, размер их может быть самым разным. Как и состав: есть пакеты, включающие фактически одну-единственную монофункциональную программу, есть пакеты, образованные набором программ определенной направленности. А есть и такие, которые сами по себе являются самостоятельной системой. Примером чему оконная система X (X Windows System), или, в просторечии, Иксы.

В состав дистрибутива пакеты могут быть включены в трех формах:

- в виде исходных текстов (sources — исходников), которые перед установкой и использованием подлежат трансформации в исполняемый код (процесс этот называется сборкой);
- в виде набора правил для получения (из сети) и сборки готовых к использованию пакетов из исходников — так называемых портов, хотя в Linux такие наборы правил имеют собственные названия в каждом дист-

рибутиве; при этом сами исходники в состав дистрибутива входить не обязаны;

- в виде заранее собранных бинарных (или прекомпилированных) пакетов, которые необходимо только распаковать из архивов и поместить, куда нужно.

Дистрибутивы, содержащие исходные тексты программ или наборы правил для их сборки, именуются Source Based (хотя точнее было бы называть их дистрибутивами портируемыми), дистрибутивы с прекомпилированными бинарниками можно назвать пакетными дистрибутивами. Правда, четкую границу между портируемыми и пакетными дистрибутивами провести нелегко, многие из них позволяют установку и в том, и в другом виде. Так вот, в разных дистрибутивах приняты разные способы пакетирования собранных программ. Собственно, различие форматов прекомпилированных пакетов — один из критериев разделения дистрибутивов на группы, о чем говорилось в *главе 3*.

Соответственно, и для установки пакетов разных форматов существует собственный, более или менее специфичный для дистрибутива инструментарий.

В ходе первичной установки пользователя больше волнует проблема выбора пакетов. И это действительно нетривиальная задача. Потому что в современных дистрибутивах пакетов этих многие тысячи. А назначение их может показаться начинающему загадочным (хотя обычно списки предлагаемых для выбора пакетов сопровождаются описаниями, подчас даже на русском языке). В итоге пользователь оказывается в положении буриданова осла, только вот выбирать ему часто приходится отнюдь не из двух охапок сена, а из многих и многих скирд.

Кроме того, проблема выбора осложняется еще явлением, известным под названием зависимости пакетов. На практике это выглядит так, что выбор пакета A автоматически влечет за собой также установку пакетов B и C, количество зависимостей для каждого пакета может быть разным.

Существуют зависимости обязательные, или "жесткие", и опциональные, или "мягкие". Первые необходимы для установки и запуска инсталлируемого пакета. В большинстве случаев в эту категорию попадают системные библиотеки. Так, с главной системной библиотекой glibc в Linux (или ее аналогом libc в BSD-системах) жесткими зависимостями связаны абсолютно все (за редчайшим исключением) программы.

Мягкие зависимости в принципе не критичны для установки и работы пакета. Просто они добавляют ему некоторую дополнительную функциональность, которую сборщики дистрибутивов полагают полезной.

Как же установщики дистрибутивов позволяют решить пользователю проблему выбора и проблему зависимостей? Первая задача облегчается тем, что

в большинстве инсталляторов предусмотрены некоторые предопределенные наборы пакетов, собранные в зависимости от целевого назначения системы: графическая рабочая станция, например, офисный или домашний компьютер, рабочее место программиста. Предопределенные наборы пакетов призваны решать и вторую проблему пользователя — проблему зависимостей. Если же пользователь решается на попакетный выбор, проблема зависимостей встает перед ним в полный рост. Однако и тут его не бросят наедине с длиннющим списком. В подавляющем большинстве пакетных дистрибутивов существует та или иная система контроля зависимостей, которая не позволит установить компонент А без компонента В, если между ними существует "жесткая" зависимость. Однако на эту тему мы еще подробно поговорим, когда в *главе 14* дело дойдет до принципов и систем пакетного менеджмента.

Я остановился на этом вопросе подробнее, чем здесь следовало бы, и к тому же существенно забежал вперед. И сделал это лишь для обоснования нехитрого тезиса: если вы устанавливаете свой первый в жизни дистрибутив Linux из категории user-ориентированных, не следует рассчитывать, что попакетный выбор даст вам какие-либо преимущества — все равно в системе окажется немало лишнего (а чего-то необходимого может и не хватить). И потому проще положиться на один из предопределенных наборов пакетов: сэкономите немало времени и нервов.

Обеспечение загрузки

После того как пользователь тем или иным образом разобрался с пакетами, наступает четвертый этап установки — обеспечение загрузки новорожденной системы. Он также включает в себя несколько моментов.

Первый момент — это средство для последующей загрузки ядра Linux. Ведь теперь у нас не будет в распоряжении загрузочного диска (вернее, прибегать к этому средству не очень удобно, оно оправдано только в ремонтно-восстановительных целях). И потому стандартный способ загрузки ядра — это специальная программа, системный загрузчик. Теоретически рассуждая, можно обойтись и без него, записав в соответствующее место диска (которое тоже именуется вполне прозрачно загрузочным сектором) некий код, позволяющий обратиться к ядру Linux, как говорится, без посредников. Однако на практике так, насколько мне известно, никто не делает: это лишает возможности сосуществования Linux с какой-либо другой ОС (интересно, какой?), ну, и кое-чего другого.

И потому в Linux принято два системных загрузчика: традиционный Lilo и GRUB, имеющий шансы стать стандартным в мире Open Sources вообще. Оба они не просто системные, а мультисистемные, т. е. позволяют загружать не только Linux, но и Windows. Какой из этих загрузчиков использует-

ся по умолчанию в данном дистрибутиве, дело личных пристрастий его разработчика. По базовым возможностям они более-менее одинаковы, и если на машине не предполагается держать больше двух систем, с этой ролью справится и Lilo, и GRUB. Любой из загрузчиков может быть установлен двумя различными способами: в загрузочный сектор диска или в соответствующий же сектор Linux-раздела (корневого или, если таковой создавался, — boot-раздела). Первый способ обязателен, если GRUB или Lilo будут основными системными (или мультисистемными) загрузчиками, что очевидно, ведь тогда другого загрузчика у нас нет. Если же на машине есть уже подходящий загрузчик (а Linux в состоянии грузить и NTloader, и известный Partition Magic, и Acronis OS Selector, и BSD Loader — штатный загрузчик одноименного семейства), то его можно сохранить.

Впрочем, на тему сосуществования и совместной загрузки Linux и Windows я распространяться не буду, т. к. уже подзабыл, как это делается. Да и написано на сей счет немало. А мы двинемся дальше.

Дело в том, что установкой программы-загрузчика дело не исчерпывается, она должна быть еще должным образом сконфигурирована. Это делается прямым редактированием конфигурационного файла GRUB или Lilo соответственно. Однако обычно установщики дистрибутивов избавляют пользователя от этой докуки, и настраивают загрузчик автоматически. От пользователя требуется только ответить на несколько вопросов.

В принципе, в понятие обеспечения загрузки можно включить еще ряд вещей, как то: настройку стартовых сервисов, часового пояса, создание учетных записей пользователей и задание их паролей. Даже окончательная локализация системы подчас выполняется на этом этапе. Мы, однако, тут задерживаться не будем. И, таким образом, нам остается рассмотреть последний этап установки.

Обеспечение работы в графическом режиме

Изложение своих POSIX-максим я начал с того, что Linux — это не Windows. Тем не менее, придать ему "подоконный" образ можно, для этого нужно только обеспечить возможность работы в графическом режиме. Этой цели служит так называемая оконная система X (X Window System), которую мы в дальнейшем будем называть просто Иксами.

Не премину в очередной раз подчеркнуть, что сами по себе Иксы ни к Linux, ни к BSD никакого отношения не имеют. Так как создавались изначально для работы поверх почти любой операционной системы (не обязательно UNIX-подобной), в принципе способной работать с графикой. В частности, одна из двух свободных реализаций Иксов (XFree86 или Xorg,

общепринятых в Linux) — это точно те же системы, работающие над Free-, Net- и OpenBSD.

Забегая вперед, замечу, что Xorg — просто побег, отпочковавшийся от общего дерева XFree86 вследствие лицензионных соображений. И технологических различий между ними пока не прослеживается. Если, конечно, не считать таковыми разные имена запускающего и конфигурационного файлов. Подробнее эта тема будет рассмотрена в *главе 16*.

В большинстве дистрибутивов XFree86 или Xorg стандартно входит в любой из предопределенных пользовательских наборов пакетов (но не серверных, там она и не нужна, и даже вредна, из соображений безопасности). Однако Иксы мало установить, их нужно еще и должным образом настроить. В принципе, как это в обычае POSIX-мира, это делается правкой соответствующего конфигурационного файла. Однако установщики user-ориентированных дистрибутивов Linux, как правило, имеют собственные средства для такого конфигурирования.

Настройка Иксов сводится к двум основным моментам: настройке устройств ввода и настройке устройств вывода. С первыми (клавиатура и мышь) проблем, как правило, не возникает. Для клавиатуры обычно достаточно выбрать близкий тип из предлагаемого списка, а если локализация системы не была предопределена выбором языка инсталляции, дополнительную ее раскладку (например, русскую) и, возможно, клавишу (или клавишную комбинацию) для переключения с одной раскладки на другую.

Здесь замечу только, что при таком подходе к русификации иногда оказывается, что в текстовом и графическом режиме национальные раскладки, да и клавиша-переключатель, окажутся несколько разными. Впрочем, как мы со временем увидим, лечится это достаточно легко.

С любыми современными мышами — ничуть не сложнее. Если она была не совсем точно определена, достаточно скорректировать это выбором из предлагаемого списка. Иногда, правда, требуется отдельно указать протокол и интерфейс нашего "грызуна". И здесь достаточно помнить, что все современные мыши работают по протоколу Microsoft Intellimouse или просто Microsoft (в зависимости от того, есть у мыши колесико или нет); исключением, говорят, являются некоторые модели Genius, но и для них можно подобрать подходящий. А интерфейс — это порт, куда мышь втыкается, и ныне их осталось два: PS/2, постепенно отмирающий, и USB.

С устройством вывода (то есть сочетанием видеокарты и монитора) несколько сложнее. Для видеокарты необходимо знать производителя чипа (графического процессора), благо, их нынче: раз (Nvidia), два (ATI), три (Intel) и обчелся (Matrox), иногда — модель поточнее, а также объем видеопамяти. Точная же настройка монитора (включая оптимальные частотные характеристики) невозможна без документации: необходимы знания допустимых диапазонов строчной и кадровой развертки. Правда, некоторые дист-

рибутивы предлагают выбрать конкретную модель из своей базы данных оборудования. Однако далеко не всегда это дает лучший результат, чем ручное задание параметров, и именно вашей модели в этом списке может не оказаться.

Общая рекомендация при указании параметров монитора в сомнительных случаях — лучше занизить характеристики, чем завысить их. Хотя леденящие душу истории о сгоревших при неправильной настройке Иксов электронно-лучевых трубках отошли в область преданий: современные мультичастотные CRT при чрезмерных к ним претензиях просто вывалятся в черный экран (что неприятно — потребуется ручная правка конфига, но не смертельно). А вот с LCD-мониторами вообще одно удовольствие, для них частотные характеристики практического значения не имеют.

И последнее, самое важное замечание. Если Иксы по каким-либо причинам не удалось настроить при инсталляции, это не означает прощания с графическим режимом работы вообще. Более, чем вероятно, что обеспечить корректную работу XFree86 (как и Xorg) можно будет потом, запустив ее штатные средства автоконфигурирования или просто ручной правкой главного настроечного файла. *О чем мы и поговорим в главе 16.*

Особенности установки BSD-систем

В предыдущих разделах речь шла в основном об установке user-ориентированных дистрибутивов Linux. Однако я помню свои собственные слова о том, что BSD-системы — ничуть не худший объект для знакомства с POSIX-системами. И потому нужно сказать пару слов об особенностях их установки.

В принципе, установка какой-либо BSD-системы включает почти те же этапы:

- обеспечение ее загрузки со сменного носителя и запуска программы-инсталлятора;
- подготовка диска, совмещенная здесь с установкой загрузчика;
- собственно установка.

Настройка работы в графическом режиме в NetBSD, OpenBSD и DragonFlyBSD не предусмотрена, а во FreeBSD носит опциональный характер (и из последних ее версий исключена, хотя, возможно, и не навечно). Однако каждый из этапов имеет некоторую специфику.

Ближе всех к Linux инсталлятор FreeBSD, запускаемый при старте машины с установочного диска. Он предлагает пользователю разбить диск на разделы, определить точки монтирования файловых систем, записать собственный начальный загрузчик (впрочем, не запрещается и использование GRUB

или Lilo), выбрать компоненты базовой системы и дополнительные (не входящие в собственно FreeBSD Distributions) пакеты, произвести сетевые и прочие настройки (включая базовую русификацию). При желании можно сконфигурировать также XFree86, для этой цели служат тут штатные конфигураторы.

Собственно, главное, что требуется от пользователя — это знание номенклатуры дисковых накопителей и понимание специфики BSD-стиля разметки диска. При этом условии установка FreeBSD проходит столь же гладко, что и любого Linux. А получаемые при постинсталляционном конфигурировании настройки делают систему пригодной к немедленному использования, по мере накопления знаний и определению предпочтений эти настройки в дальнейшем можно будет довести до личного идеала.

Установочные программы Net- и OpenBSD имеют более аскетический вид. Начать с того, что OpenBSD не имеет штатного инсталляционного диска, а установочный диск NetBSD для архитектуры i386 лишь недавно (с версии 1.6.1) стал, наконец, загрузочным. В принципе в обеих этих ОС чуть ли не в качестве основного метода установки принимается загрузка с дискеты и последующая инсталляция в сети (по FTP- или HTTP-протоколу, с локального сервера).

Тем не менее, сама по себе установка и Net-, и OpenBSD сводится к разметке диска, выбору устанавливаемых компонентов базового набора и их записи. Как и при установке FreeBSD, обязательное требование — понимание специфики BSD-разметки вообще и знание особенностей номенклатуры устройств данной ОС. Плюс еще исключительная аккуратность при установке на диск с другой операционной системой и (или) данными: в отличие от FreeBSD, здесь изменение дисковой структуры происходит почти в реальном времени, и ошибка может привести к тяжелым последствиям.

Установка DragonFlyBSD имеет лишь одну существенную особенность: базовая система не распаковывается из архива (архивов), а переносится непосредственно с дистрибутивного диска специальной командой `cpdup` (что, впрочем, маскируется инсталляционной программой BSD Installer).

Всем сказанным я не хотел устрашить потенциального пользователя BSD-систем или внушить ему, что установка их дело чрезвычайно сложное и опасное. Отнюдь, требуются лишь некоторые предварительные знания и известная аккуратность. Последнее — дело характера, Что же касается минимума BSD-специфичных познаний, то их легко приобрести из чтения документации и прочих источников (например, этой книги). Так, практически вся официальная документация по FreeBSD, например, ныне доступна в русских переводах (**http://www.freebsd.org.ua/**), переводы документации по NetBSD также появились в последнее время в немалом количестве (**http://dreamcatcher.ru/**).

Проблема выбора

В рамках второй истины остается рассмотреть последний вопрос — о выборе водоема для обучения плаванию. Один из наиболее распространенных вопросов: какой дистрибутив Linux выбрать? Или более редкая его вариация: что выбрать, Linux или Free- (Open-, Net-) BSD?

Рискну предложить свой, претендующий на универсальность совет: нужно попробовать разные дистрибутивы (возможно, даже разные ОС). И выбрать подходящий вариант методом проб и ошибок. Тем не менее, с чего-то нужно начинать. И здесь самое главное — определиться со своими потребностями и возможностями.

В дальнейшем я буду исходить из того, что начинающий пользователь любой POSIX-системы ставит своей целью (в том числе и) ее изучение: поверьте, в конце концов без этого не обойтись. И здесь очень важно, как этот самый пользователь хочет (а главное, может) осуществлять это изучение.

Одно дело, если пользователь имеет возможность уделить некую толику времени (условно скажем, месяц) на доскональное знакомство с системой. Только после этого он начнет ее практическое использование. И совсем другое, если начинать практическую работу (хоть в каком-то объеме) ему нужно сразу после установки системы, а вопросами самообразования заниматься по ходу дела, при наличии свободного времени или по возникновении проблем, требующих для своего решения некоторых дополнительных знаний.

Кроме обстоятельств, на это влияют еще и индивидуальные особенности пользователя, индуктивный или дедуктивный стиль мышления. Фигурально выражаясь, одному человеку для понимания устройства двигателя внутреннего сгорания требуется предварительно досконально изучить учебник термодинамики, иной же не поймет, что такое цикл Карно, пока не переберет руками весь двигатель своего автомобиля.

Тем не менее, принудительная сила реальности такова, что большинству людей, вне зависимости от их природной склонности к дедукции или индукции, приходится заниматься освоением новой ОС в ходе ее практического использования и параллельно с ним. То есть сразу по установке системы ему нужно иметь некий минимум функций, настроек, приложений.

Именно на такую категорию пользователей рассчитаны такие пакетные user-ориентированные дистрибутивы, как Red Hat, Mandrake, их более или менее отдаленные клоны (Altlinux, ASPLinux). Каковые, казалось бы, и будут оптимальным выбором для большинства начинающих пользователей, не правда ли?

Однако все не так просто. Потому что в расплату за простоту установки и быстроту "вхождения в тему" пользователь получает по умолчанию некую Windows-подобную систему, перегруженную программами, ему, возможно, не нужными (назначение которых он рискует не узнать никогда). Систему, модификация которой отнюдь не прозрачна. По крайней мере, для начинающего, вопреки существующему мнению, вносить изменения в конфигурацию user-ориентированного дистрибутива гораздо сложнее, чем заниматься настройкой с нуля дистрибутива Source Based. Тем, кто не верит, предлагаю посмотреть конфигурационные файлы Suse (если вы пользователь Red Hat) или Mandrake (если вы пользователь Suse). И при указанных условиях попытаться внести в них осмысленные изменения.

С другой стороны, дистрибутивы, целиком собираемые из исходников, подобно Gentoo, для начинающего пользователя также, скорее всего, не приемлемы. И дело даже не в том, что уже установка таких систем требует определенных знаний: знания дело наживное, для их приобретения достаточно умения читать. А документированы Source Based дистрибутивы обычно очень хорошо, лучше, чем многие пакетные дистрибутивы (**http://www.gentoo.org/doc/en/index.xml**), есть немало и русскоязычных материалов (**http://www.gentoo.org/doc/ru/index.xml**). Нет, причина — в том, что любой Source Based дистрибутив требует кропотливой настройки до запуска первого практически нужного приложения. А мы уже выяснили, что свежеустановленную ОС обычно приходится начинать использовать для работы сразу.

Так что требуется некоторый компромисс между быстротой подготовки к использованию и гибкостью настройки, если не немедленной, в ходе инсталляции, то последующей. И такой компромиссный вариант реализуется в виде FreeBSD: сразу после установки этой ОС пользователь получает в свое распоряжение разумно сконфигурированную (по умолчанию), корректно русифицированную систему, снабженную необходимым минимумом (а при желании — и максимумом) приложений. Которая в дальнейшем легко может быть перенастроена в соответствии с его возросшими (или, наоборот, снизившимися, что очень немаловажно, чистое удаление установленных ранее излишков составляет в пакетных дистрибутивах Linux не меньшую проблему, чем в Windows) потребностями.

Конечно, для установки FreeBSD пользователю необходимо несколько больше предварительных знаний, чем для установки user-ориентированного пакетного дистрибутива Linux. Однако за источником таких знаний далеко ходить не нужно, проект FreeBSD прекрасно документирован, и большая часть его материалов доступна ныне и на русском языке. А серия руководств, как переводных (например, Иллюстрированное руководство по установке FreeBSD, **http://unix1.jinr.ru/~lavr/FreeBSDR**, не говоря уже о FreeBSD Handbook **http://www.freebsd.org.ua/doc/ru_RU.KOI8-R/books/**

handbook/index.html), так и исходно русскоязычных (например, FreeBSD: быстрое развертывание, **http://unix.ginras.rul/bsd/inst001.html**) делает инсталляцию ее ничуть не сложнее, чем развертывание Mandrake или Red Hat.

Если же выбор пользователя склонился все-таки именно к Linux, сочетание быстроты развертывания и дальнейшей возможности совершенствования он найдет, например, в Debian (**http://www.debian.org**). Конечно, это несколько своеобразная система, обладающая изощренными специфичными средствами настройки.

Теперь посмотрим, каковы возможности выбора для пользователя, предпочитающего более или менее глубоко изучить POSIX-систему, прежде чем применить ее на практике. И здесь, казалось бы, лучший выбор — это именно Source Based дистрибутивы Linux. Не буду с этим спорить: установка и индивидуальная настройка, например, Gentoo способна дать в этом плане очень много. Однако...

...Однако ценность любого дистрибутива Linux для первичного изучения POSIX-систем резко снижается этим самым фактором: тем, что изучению подвергается один из многочисленных дистрибутивов. И каждый из них имеет массу специфичных только для него особенностей, что знания и навыки, полученные при общении, скажем, с тем же Gentoo, могут оказаться лишь ограниченно применимыми в ином, даже идеологически близком, дистрибутиве.

И тут мы опять вспоминаем о BSD. Конечно, BSD-системы вообще имеют свою специфику по сравнению с Linux (и по сравнению с остальными, проприетарными, POSIX-системами, генетически происходящими от классического UNIX System V). Однако специфика эта, в сравнении с многообразием дистрибутивов Linux, весьма невелика. А уж внутри BSD-клана различия просто малы. И для начинающего пользователя очень существенно, что все наличные источники информации по любой BSD-системе посвящены именно этой ОС. Тогда как авторы толстых книг про Linux далеко не всегда последовательно подчеркивают, что в их описаниях является общим для POSIX-систем, что характерно для Linux вообще и что специфично для конкретного дистрибутива.

Какую из BSD-систем выбрать как объект углубленного изучения? Теоретически рассуждая, чуть ли не любую. Хотя следует учитывать, что OpenBSD и NetBSD относительно слабо освещены в русскоязычных источниках, имеют проблемы с русификацией. А главное, их достоинства (защищенность и кросс-платформенность) для индивидуального пользователя несущественны. Так что на выбор остаются FreeBSD или DragonFlyBSD. Мой личный выбор в пользу последней, но это сугубо дело вкуса...

Так что же, дружно бросаем Linux и переходим на BSD, спросите вы меня? Я не хотел бы, чтоб сказанное оставило такое впечатление. Хочу лишь под-

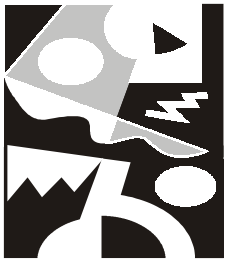
черкнуть, что нужно быть готовым к тому, что первый выбранный для установки дистрибутив Linux вряд ли окажется вашим окончательным выбором. Вполне возможно, что вам придется перепробовать несколько дистрибутивов (или даже пару-тройку ОС), прежде чем будет выбрана система, наиболее отвечающая вашим задачам (и гармонирующая с внутренним мироощущением).

И, тем не менее, рискну предложить еще один вариант выбора среди дистрибутивов Linux из бесчисленного их множества. И будет это Archlinux, занимающий в определенной мере промежуточное положение между пакетными и портируемыми системами. Сам по себе он распространяется в виде iso-образа, содержащего исключительно прекомпилированные пакеты. Из которых он и может быть развернут, благодаря простой, но гибкой установочной программе, буквально в считанные минуты. Даже стадия сборки собственного ядра, неизбежная в большинстве Source Based дистрибутивов, тут обязательной не является (хотя и не возбраняется): прекомпилированное умолчальное ядро обеспечивает близкое к оптимальному соотношение функциональности и компактности.

В то же время Archlinux включает в себя систему сборки пакетов из исходников, позволяющую пересобрать все установленные программы под свои задачи (и свое "железо") с помощью одной-двух команд. А прозрачность структуры этой системы позволяет пользователю с минимальными навыками программирования (и даже совсем без оных) легко восполнить недостающие в штатной поставке компоненты.

Вот, пожалуй, и все относительно второй из моих вечных истин — установки системы. Ко всем из затронутых здесь вопросов нам еще придется возвращаться, однако мне кажется, что сказанного достаточно для того, чтобы бестрепетно вставлять дистрибутивный компакт-диск Linux или BSD и смело жать на три клавиши. Делая тем самым первую свою попытку заплыва в стиле "кроль".

Глава 6



Все для блага человека: пользовательские акаунты

- Очередная преамбула.
- О себе, любимом.
- Атрибуты учетной записи.
- Доступ к атрибутам.
- Создание и изменение акаунтов.

Следующими из "вечных истин" POSIX-систем являются понятия файла, процесса и пользователя. Это — те три кита, на которых зиждется дивный POSIX-мир. Все, что существует в системе статически, суть файлы; все, что существует в системе в динамике, суть процессы; все, кто тем или иным образом взаимодействует с системой, ее процессами и ее файлами, суть пользователи.

Очередная преамбула

Вообще-то, рассуждать об этих материях (вернее, воспринимать такие рассуждения) — довольно трудно. Потому что, например, пользователь (со своими атрибутами) запускает некий процесс (атрибуты которого зависят и от пользователя, и от исполнимого файла, за этот процесс отвечающего), в результате создается некий файл, атрибуты которого (определяемые сочетанием атрибутов процесса) и устанавливают его отношение к данному пользователю.

Получается нечто вроде известной сказки про белого бычка, которая, как уже говорилось, в мире UNIX задумчиво называется рекурсией. И определить понятие процесса без представления, что такое файл и пользователь, столь же трудно, как определить понятие файла, не имея представления о

процессе. Однако мы, POSIX'ивисты, не боимся трудностей и, подобно Александру Филипповичу (Македонскому), разрубим этот узел, установив в нем точку отсчета. В качестве каковой резонно было бы принять себя любимого, сиречь пользователя.

О себе любимом

Время от времени появляются сообщения о выходе очередного Linux-дистрибутива, наконец-то, "с человеческим лицом". Не верьте им: лицо у POSIX-систем было человеческим всегда. Ибо одно из краеугольных понятий, на которых стоял, стоит и стоять будет POSIX-мир, — это понятие пользователя. А какое лицо может быть более человеческим, чем собственная морда?

В компенсацию своего эгоцентризма заметим, что круг пользователей собой любимым не исчерпывается. Потому что в POSIX-системе, даже если вы не делите ее с ближними (по дому или по работе), обязательно присутствует еще один пользователь — всемогущий root-оператор (именуемый в народе также администратором или суперпользователем). Правда, в условиях локального (особенно домашнего) компьютера в его роли, скорее всего, выступает наша же персона.

Однако есть в системе и пользователи, никакой реальной личности не соответствующие. Но, тем не менее, способные выполнять в этой системе некоторые вполне реальные действия. Мистицизм такого явления отражен в общем их названии — демоны (daemon). На самом деле ничего мистического в них нет, но это — тема особого разговора. Если же добавить, что существуют еще и псевдопользователи типа почтовых или FTP-клиентов, то может показаться, что понятие пользователя утрачивает всякую определенность.

И потому принимаем волевое решение: считать пользователем любого, кто имеет учетную запись в соответствующей базе данных (нетрудно догадаться, что называется она незатейливо — базой данных пользователей). Почему, строго говоря, и правильней говорить не о пользователях, а об их учетных записях (акаунтах, встречается еще и выражение "пользовательские бюджеты"), или, как более изящно выразился Иван Паскаль, учетных карточках. Однако фраза типа "учетная карточка пользователя имярек выполняет такое-то действие" по-русски звучит несколько странно (я бы даже сказал, зловеще). И потому будем по-прежнему применять личную форму: пользователь, он же юзер. Тем более, что в этой главе речи о демонах или псевдопользователях не будет.

Атрибуты учетной записи

Каждый пользователь имеет фиксированный набор закрепленных за ним атрибутов, собственно и составляющий содержание его учетной записи в базе данных. Каких? А давайте подумаем, какие атрибуты пользователю необходимы.

И тут самое время вспомнить, что даже на суперперсональной машине пользователей всегда больше одного. И значит, система должна их как-то различать между собой. Как? Да вполне по-человечески, по именам. И потому первый атрибут пользователя — это его имя, именуемое также login. Именно предложение ввести свое пользовательское имя — это первое, что видит пользователь POSIX-системы после ее загрузки.

Конечно, не нужно чересчур очеловечивать компьютеры (говорят, что они этого не любят). И считать, что он волшебным образом опознает вас по имени, данному при рождении (или крещении). Нет, имя пользователя — это просто некий набор (почти) любых символов. Конечно, и собственное имя в этом качестве использовать не возбраняется. Ведь единственное общее к нему требование — уникальность. Правда, в некоторых системах существует ограничение на длину login. И обычно принято ограничиваться символами алфавитно-цифровыми.

Так что имя пользователя в системе условно. Более того, для системы, по большому счету, имя это абсолютно до лампочки. Потому что опознает она пользователя на самом деле не по нему, а по поставленному ему в соответствие числовому идентификатору (UID, User IDentificator). Как правило, это просто порядковый номер пользовательской учетной записи, причем для обычного пользователя — начиная с некоего минимального числа. В одних системах это может быть 100, в других 500, в большинстве Linux-дистрибутивов 1000. Младшие идентификаторы система резервирует для своих целей: для тех самых демонов и виртуальных юзеров, о которых говорилось ранее. И еще: идентификатор 0 всегда и везде в POSIX-мире бронируется для root.

Стоит добавить, что и само имя пользователя не обязательно, достаточно, чтобы акаунт имел уникальный числовой идентификатор. Соответствие же его какому-либо имени обеспечивается не самой системой, а одной из дополнительных (хотя и необходимых) программ (системной библиотекой glibc — просто запомним пока это слово, как колдовское заклинание).

Так почему же все-таки имя пользователя считается непременным атрибутом его акаунта? Да все потому, что лицо у UNIX — человеческое. И созда-

тели системы прекрасно понимали, что запомнить имя alv для пользователя гораздо проще, чем идентификатор 1276. Вот ему и пошли навстречу.

Итак, для вхождения пользователя в систему ему необходимо указать свое имя (login). Однако как система опознает, что пользователь имярек — именно тот, за кого себя выдает? И не возникнет ли ситуация, как в хармсовском анекдоте, когда Гоголь переоделся Пушкиным, а Державин решил, что это и вправду Пушкин, и, сходя в гроб, благословил его?

Нет, не возникнет. Потому что система, с бдительностью часового, запросит у пользователя подтверждения того, что он именно тот, за кого себя выдает. По аналогии с вопросом часового, легко догадаться, что подтверждение это именуется паролем (password). В качестве какового выступает, опять же, некая последовательность символов (пущей секретности ради рекомендуется разбавлять их специальными символами, а для литер применять смесь нижнего и верхнего регистра), которую, теоретически рассуждая, никто, кроме пользователя, знать не может. Этот пароль являет собой третий непременный атрибут пользователя.

В данном случае фраза "никто не должен знать" понимается в буквальном смысле: даже всемогущий root в общем случае пользовательский пароль знать не должен и узнать обычными способами не может. Потому что пароль хранится в односторонне зашифрованном виде, и из его шифрованного представления вычислить вводимую последовательность символов нельзя.

Так вот, при входе пользователя (название этого процесса — авторизация, регистрация) система сначала проверяет, а зарегистрировано ли это имя (точнее, соответствующий ему числовой идентификатор) в базе данных ее пользователей. А затем, после ввода пароля, по той же базе данных определяет, соответствует ли этот пароль зафиксированному для данного пользователя. И в благоприятном случае разрешает вход в сокровенные свои недра, иначе же — после более или менее длительного промежутка времени, предлагает повторить процедуру авторизации (в некоторых системах количество возможных повторов ограничено).

О правилах выбора пароля написано немало, в том числе и экспертами по компьютерной безопасности. Поэтому я не буду отвлекать читателя своими дилетантскими рассуждениями. Замечу только, что даже самый простой, с точки зрения устойчивости к взлому, пароль — на несколько порядков лучше, чем его полное отсутствие. Хотя в принципе в Linux можно и обойти необходимость ввода пароля, и учредить беспарольный вход для любого пользователя (в том числе и для root). Однако делать это можно только на машине сугубо персонального назначения, не подключенной ни к какой сети вообще (в том числе и электрической). Да и то не стоит, дабы не выработалась вредная привычка. Если же ввод пароля очень напрягает, есть не-

сколько возможностей автоматизировать процесс регистрации, не погружаясь в болото беспарольного плюрализма.

Я описал здесь обычную процедуру входа в систему — через текстовое приглашение. Однако ныне в большинстве дистрибутивов по умолчанию устанавливается так называемая авторизация в графическом режиме. Она обеспечивается специальными программами — менеджерами входа (xdm, kdm, gdm). Однако сущность процедуры от этого не меняется: любой из них также затребует ввод логина и пароля.

Теперь посмотрим, что же происходит после успешного завершения авторизации. Ясно, что пользователь входит в систему не для чего-нибудь там, а дабы выполнить какие-то действия (немного поработать, например, пока кушать не позовут). А для этого ему требуется рабочая среда. И программа, таковую обеспечивающая, запускается системой после принятия логина и пароля. В подавляющем большинстве случаев такой программой будет командная оболочка, о чем подробно будет рассказываться в *главе 15*. Пока же отметим только, что имя этой командной оболочки и есть следующий атрибут пользовательского акаунта.

Скорее всего, деятельность пользователя в системе включает в себя ввод и обработку каких-либо данных. Которые нужно куда-то записывать. Поскольку POSIX-системы — многопользовательские по своей сути, данные нашего пользователя не должны путаться с данными других таких же юзеров. Из чего следует, что после успешной авторизации наш пользователь должен получить некоторое место для записи своих данных, куда другие без его дозволения соваться не моги. И действительно, такое место (именуемое домашним каталогом пользователя, символически обозначаемым как ~/ или $HOME) — следующий атрибут учетной записи.

И последнее. Многопользовательская система предполагает не только защиту пользователей от зловредного влияния других пользователей, но и обмен данными между группой пользователей-товарищей. Которая так и называется — группой пользователей, или просто группой. И любой пользователь по умолчанию включается минимум в одну группу, может быть, свою собственную, не имеющую других членов. Хотя в большинстве Linux-дистрибутивов такая группа обычно называется `users` и включает в себя всех реальных пользователей, кроме root (а в BSD, например, каждый пользователь по умолчанию приписывается к своей собственной, одноименной ему, группе). И имя этой группы — еще один непременный атрибут учетной записи пользователя.

Как и в случае с именем пользователя, имя группы — лишь одна из метафор, очеловечивающих поля пользовательского акаунта. И потому в последнем фигурирует, собственно, не имя (например, users), а соответствующий

группе численный идентификатор. Опять же аналогично идентификаторам пользователей, это обычно порядковый номер вновь создаваемой группы. И начальной точкой отсчета выступает какое-либо число. Например, во FreeBSD нумерация пользовательских групп начинается с 1001: все, что меньше, резервируется для системных псевдопользователей (мы же помним, что в отношении их не допускается никакой дискриминации, и они тоже являются членами всяких разных групп). А в Linux группа реальных пользователей users обычно имеет идентификатор 100.

Я перечислил атрибуты учетной записи пользователя, общие для всех POSIX-систем. И те, которые, как правило, заполняются при использовании всяких программ управления акаунтами. Однако их бывает и больше. Так, практически всегда имеется атрибут комментария, содержание которого произвольно (например, настоящие имя и фамилия пользователя). А в BSD-системах имеется еще и атрибут `класс пользователя` (class), как станет ясным в дальнейшем, очень полезный.

Доступ к атрибутам

Каким образом пользователь может ознакомиться с атрибутами своей учетной записи? Ответ, как обычно в POSIX-системах, будет таким: разными способами. Правда, для этого ему нужно не только авторизоваться в системе (чему мы как будто бы уже научились), но и суметь дать несколько команд. Собственно, команды — тема одной из следующих глав (конкретно, двенадцатой), и потому пока я дам несколько примеров на уровне заклинаний (надеюсь, что скоро они перестанут казаться чем-то сверхъестественным).

Команда, позволяющая пользователю получить информацию о самом себе, это `echo`. Она просто выводит на экран данные ввода (например, с клавиатуры).

Раз уж речь, опережая события, зашла о командах, начинающему пользователю (а читают эти строки, скорее всего, именно совсем начинающие — прочим уже давно надоело пережевывание известных им вещей) очень рекомендуется понемногу запоминать их: все они потребуются в ходе дальнейшего изложения.

Итак, первый резонный вопрос: кто я? Для ответа на него командуем:

```
$ echo $USER,
```

где символ `$` после команды `echo` и пробела обозначает, что все следующие символы являются переменной, а `USER` — имя данной конкретной переменной.

Так вот, на эту команду мы незамедлительно получим ответ:

`alv`,

свидетельствующий, что давший ее пользователь носит имя (login) alv. Ответом на запрос о программе, играющей роль пользовательской среды:

`$ echo $SHELL`,

будет ее имя, вместе с путем к исполняемому файлу:

`/bin/tcsh`.

А если пользователя заинтересует место хранения собственных данных, следует спросить так:

```
$ echo $HOME
```

`/home/alv`,

где /home/alv и есть домашний каталог пользователя alv (совпадение имени пользователя и его домашнего каталога не обязательно, но, как правило, имеет место быть).

Однако наиболее полную информацию о себе пользователь может получить прямым просмотром базы пользовательских акаунтов. База данных хранится в файле passwd каталога /etc (вопросы, что такое файл, каталог, и почему его имя должно предваряться символом / пока замнем для ясности, речь об этом будет в *главе 8*). Пока для наших целей достаточно знать, что /etc/passwd — простой текстовый файл, доступный для чтения кому угодно, был бы инструмент для чтения. Забегая вперед, скажу, что, как и следовало ожидать, такой инструмент в POSIX-системах предоставляется и даже во множестве видов. Хотя сейчас нам много не нужно, достаточно одного, наиболее удобного. Таковым в Linux выступает обычно команда `less`, а в BSD — еще и команда more (правда, как выяснится в *главе 8*, это одно и то же). И та, и другая требуют указания еще и имени просматриваемого файла. То есть в нашем случае это будет выглядеть так:

```
$ less /etc/passwd
```

или

`$ more /etc/passwd`.

Вывод данной команды может выглядеть таким образом (на примере Archlinux, где этот файл по умолчанию устроен проще всего):

```
root:x:0:0:root:/root:/bin/zsh

bin:x:1:1:bin:/bin:

daemon:x:2:2:daemon:/sbin:

mail:x:8:12:mail:/var/spool/mail:
```

```
ftp:x:14:11:ftp:/home/ftp:
nobody:x:99:99:nobody:/:
alv:x:1000:100::/home/alv:/bin/zsh
lis:x:1001:100::/home/lis:/bin/zsh
```

Из чего можно видеть, что мы действительно имеем дело с базой данных, хотя и очень простой по структуре. Любая же база данных, как известно, состоит из уникальных записей (или строк), относящихся к одному объекту базы и составляющих запись полей, характеризующих свойства (атрибуты) этого объекта. Поля записи обязательно разделяются чем-либо: в зависимости от формата базы это могут быть пробелы, символы табуляции, двоеточия (colon), точки с запятой (semicolon) и что-то еще.

Так и здесь, каждая строка представляет запись одного пользовательского акаунта: первым идет запись для суперпользователя, как ему по должности положено, следующие — это те самые псевдопользователи, которых мы договорись пока не трогать, а вот последние две записи — это уже акаунты реальных пользователей.

Наборы символов, разделенные символом двоеточия, — поля этой записи, описывающие каждый атрибут учетной записи (то есть colon — разделитель полей в нашей базе). Считаем, атрибутов этих оказывается 7 (хотя и не все поля, как скоро станет ясно, обязательны к заполнению, напротив, некоторых атрибутов пользовательского акаунта в этой базе мы не увидим). Некоторые атрибуты нам уже знакомы, некоторые — нет. Тем не менее, пройдемся по ним по всем, ведь повторенье мать ученья.

На первом месте каждой записи, в соответствии с гуманистической ориентацией POSIX-систем, стоит имя пользователя, реального или виртуального — не важно. Важно, что одинаковых мы среди них не найдем (в принципе можно создать пользователей с одинаковыми именами и разными численными идентификаторами, но ничего, кроме путаницы, это не даст).

Второе поле отведено под пароль пользователя. В данном случае оно заполнено символом `x` (или `*`, что почти равноценно). Это не значит, что нас посылают по известному на Руси адресу. Просто в файле /etc/passwd, вопреки его названию, реальных паролей не содержится (к этому вопросу мы еще вернемся).

Итак, с паролями пока проехали. Третье поле — самое важное, это и есть числовой идентификатор пользователя, служащий для однозначного его опознавания в системе. Аналогичен и смысл четвертого поля, только здесь стоит численный идентификатор основной группы, к которой пользователь приписан.

Пятое поле записи — так называемый комментарий. Для root и псевдопользователей здесь стоят некие условные значения, совпадающие с их именами. Что на деле отнюдь не обязательно, форма заполнения этого поля свободная. Однако некоторые утилиты создания учетных записей пользователей обязательно потребуют придать этому полю какое-либо значение.

Шестое поле — домашний каталог пользователя. Для псевдопользователей тут опять же стоят некие непонятные имена, в некоторых случаях оно может быть и пустым. А вот для пользователей реальных здесь указываются каталоги, куда они и в самом деле пишут свои данные. Суперпользователь в этом отношении похож на обычных пользователей, его домашний каталог /root также предназначен для собственных данных администратора. Как будет ясно в дальнейшем, выполнять от лица root какие-либо пользовательские задачи — занятие крайне нездоровое, и потому домашний каталог его заполняется (обычно) исключительно его личными конфигурационными файлами (до которых тоже со временем дойдет дело).

И, наконец, последнее поле записи — это имя среды обитания пользователя, запускающейся первой сразу же по его авторизации (в общем случае это login shell). Можно видеть, что у псевдопользователей это поле пусто — они в такой среде не нуждаются. А вот root и тут проявляет свою человеческую сущность, и у него login shell имеется.

Вообще говоря, и для обычных пользователей заполнять седьмое поле записи не обязательно: в этом случае в качестве login shell для них будет запущена некоторая предопределенная по умолчанию среда (в POSIX-системах она всегда носит имя /bin/sh, хотя в реальности это могут быть разные программы).

В принципе в качестве login shell может использоваться не обязательно какая-либо командная оболочка, но чуть не любая произвольная программа, например, Midnight Commander, интерпретатор языка программирования и даже текстовый редактор. Однако это имеет смысл только в специальных случаях, которые здесь не рассматриваются. И потому далее под login shell будет всегда пониматься шелл в собственном смысле слова.

В общем, из рассмотрения содержимого файла /etc/passwd мы узнали о пользователях, имеющих право на вхождение в данную систему, почти все. Однако слабо освещенным остался вопрос о группах, ведь вместо человечьего имени их в соответствующем поле стоит лишь нечленораздельный идентификатор. Что ж, дело легко поправимо, сведения о группах также хранятся в специально предназначенном для этого файле, который называется (кто бы мог подумать) group и находится в том же каталоге /etc, где мы на него и посмотрим:

```
$ less /etc/group
```

Он выглядит примерно следующим образом (многоточиями я заменил группы псевдопользователей, которые в данный момент нас не волнуют, — на самом деле здесь полторы дюжины записей):

```
root::0:root
...
wheel::10:root,alv,lis
...
users::100:alv,lis
sound:x:101:alv,lis
```

Можно видеть, что структура этого файла аналогична базе данных пользователей, но еще более проста: в каждой записи лишь четыре поля. Первое, как и следовало ожидать, — имя группы пользователей. Второе поле пустое. В принципе, оно предназначалось когда-то для пароля группы, однако по своему назначению ни в одной POSIX-системе, насколько мне известно, не используется. Третье поле — числовой идентификатор группы, тот самый, который мы уже видели в четвертом поле файла /etc/passwd.

Ну, а здесь четвертое поле — это список пользователей, входящих в каждую группу. Как уже говорилось, каждый пользователь является членом минимум одной группы, которая называется основной. Однако это не мешает ему состоять, при необходимости, и в каких-либо иных группах. И это очень эффективный механизм разграничения доступа к данным (или, напротив, организации их совместной обработки разными пользователями).

Легко видеть, что root-оператор является единственным членом своей собственной группы (это характерно для большинства псевдопользовательских групп). А группа users объединяет двух пользователей, которые в моей системе являются реальными, и это их основная группа. Кроме того, оба они состоят еще и членами группы sound: без этого, как будет показано в свое время, эти юзеры были бы лишены счастья слушать мою коллекцию авторской песни (и любые другие звуки тоже).

Правда, в данном случае и тот, и другой пользователь представляют собой мою скромную персону. Только не подумайте, что я страдаю раздвоением личности: просто alv — это тот пользователь, от лица которого я выполняю свою работу (например, пишу эти строки), а акаунт lis предназначен для всяких экспериментов с системой, более или менее нездоровых (зачем — станет ясным из одного из последующих разделов).

Особого внимания заслуживает группа wheel. Во многих Linux-дистрибутивах (и, добавлю, во всех BSD-системах) временно получить права администратора могут только члены этой группы (как — скажу чуть позже). В других системах это не так, впрочем, положение легко изменить как в ту, так и

в другую сторону. Потому что особый статус членов группы wheel зависит не от ОС или дистрибутива, а устанавливается в файле /etc/login.access, где и может быть отменен при желании.

К группе wheel в BSD-системах приписано большинство исполнимых файлов базовой системы, расположенных в каталогах /{bin,sbin} и /usr/{bin,sbin} (забегая вперед, отмечу, что фигурные скобки символизируют группировку имен, то есть /{bin,sbin} эквивалентно /bin и /sbin).

Некоторые файлы из каталогов /{bin,sbin} и /usr/{bin,sbin} (напомню, что владельцем всех их является пользователь root) имеют, однако, иную групповую принадлежность. Возникает вопрос, для чего они предназначены? Это станет понятным после рассмотрения понятия файла и его атрибутов. Пока же замечу, что, например, членство в группах dialer и network необходимо для использования модемного и сетевого соединения, а членство в группе operator дает пользователю возможность завершать работу машины соответствующими командами, в иных случаях требующими полномочий администратора. Напомню, что это относится к BSD-системам — в различных дистрибутивах Linux и группы другие, и назначение их может быть разным.

Осталось прояснить вопрос с паролями. Мы уже видели, что в файле /etc/passwd никаких паролей на самом деле нет. На вопрос "почему" легко ответить словами Балбеса из "Операции Ы": чтоб никто не догадался. Ведь если мы, будучи простыми пользователями, легко получили доступ к содержимому столь, казалось бы, важного файла, то и для злоумышленника это труда не составит. Ибо право чтения файла /etc/passwd имеет любой пользователь (хотя вносить изменения в содержимое записей и полей, за единичными исключениями, может только root). И потому для хранения паролей в большинстве дистрибутивов Linux предусмотрено специальное потайное место — файл /etc/shadow.

Впрочем, просмотреть его сразу нам не удастся, ведь мы вошли в систему в качестве обычного пользователя, а ему доступ к /etc/shadow запрещен под любым соусом (в том числе и просто для чтения). Что делать?

Парой абзацев раньше я обмолвился о том, что есть возможность временно получить права администратора. Для этого предназначена специальная команда `su` (что иногда трактуют как аббревиатуру от Super User, но на самом деле означает Set UID, потому что она позволяет получить права не только администратора, но и любого другого пользователя, достаточно указать его имя в качестве аргумента). Данная же без всякого аргумента, в самой простой форме как

```
$ su,
```

она, однако, предоставляет доступ именно к суперпользовательскому акаунту, для чего сначала будет запрошен соответствующий пароль. Если мы его знаем и введем правильно — казалось бы, ничего не произойдет, никаких сообщений не последует. Правда, если присмотреться, окажется, что изменился вид приглашения командной строки (как — зависит от настроек командной оболочки, на которых пока задерживаться не будем). И это должно послужить сигналом к повышенному вниманию: с этого момента наш обычный пользователь наделен полномочиями root.

Почему требуется повышенное внимание? Да потому, что полномочия root, как будет рассказано в разделе о файлах и их атрибутах, практически ничем не ограничены. И он легко может по ошибке совершить какие-нибудь непоправимые действия, вплоть до полного разрушения системы.

Мы однако не собираемся делать ничего потенциально опасного. Нам бы только на файл с паролями полюбоваться, ведь теперь мы имеем на это право. Так что

```
$ less /etc/shadow
```

и вперед, удовлетворять свое любопытство (к тому же не без пользы).

Что же мы видим в нашем /etc/shadow? Да примерно такую же базу данных, что и раньше, о восьми полях. Первое поле — имя пользователя, реального или виртуального. Второе у псевдопользователей пусто. А у реальных (в том числе и у root) заполнено нечленораздельным набором символов. Это и есть пароль. Его представление в базе не имеет ничего общего с тем, что вводит пользователь при авторизации. Ибо, как уже говорилось, пароль помещается в базу в односторонне зашифрованном виде. То есть по его представлению восстановить последовательность символов для ввода невозможно (как говорят эксперты по безопасности, невозможно в принципе; глядя на свой /etc/shadow, охотно им верю). И односторонняя шифровка пароля — это первый эшелон обороны в Linux.

Если пароль нельзя расшифровать, то его можно подобрать. И представление зашифрованного пароля может помочь в этом черном деле. А потому доступ к файлу /etc/shadow закрыт для всех, кроме root — даже для просмотра. Что являет второй эшелон обороны Linux-системы.

Я не случайно в последних абзацах подчеркиваю, что речь здесь идет именно о Linux. Потому что такой механизм защиты (он называется механизмом теневых паролей — shadow passwords) принят именно в этой ОС (по крайней мере, в большинстве ее дистрибутивов). А вот в BSD-системах механизм теневых паролей не используется, а эшелонированная оборона достигается другим способом. Там в качестве дополнительного кольца защиты используется пара файлов: /etc/master.passwd и /etc/passwd. Первый из них, доступ-

ный только для root, содержит все данные о пользователе, включая его пароль, во втором, открытом для всеобщего обозрения, есть все то же самое — но без пароля.

К слову сказать, во FreeBSD оба эти файла существуют в двух вариантах: текстовом (собственно /etc/master.passwd и /etc/passwd), доступном для просмотра, и неудобочитаемом бинарном (/etc/master.spwd.db и /etc/spwd.db). Именно последние используются для идентификации пользователей: изменения, внесенные (скажем, в обычном редакторе) в текстовые варианты баз пользовательских акаунтов, должны быть соответствующим образом преобразованы. Что, кроме всего прочего, может рассматриваться как еще один эшелон обороны.

Завершая разговор о паролях, еще раз обратимся к файлу /etc/master.passwd из FreeBSD. Просмотр его показывает, что кроме полей, общих с файлом /etc/passwd, в нем можно видеть еще три. Они хранят информацию о времени последнего изменения пароля, а также могут определять всякие дополнительные сроки действия пароля. Например, здесь можно задать дату, когда пароль должен быть изменен, или дату, после которой вход по этому паролю в систему невозможен. Наконец, очень важное поле `class` (например, `russian`) позволяет определить некоторые свойства, общие для ряда пользователей.

Создание и изменение акаунтов

Таким образом мы плавно подошли к вопросу о том, как создаются и изменяются пользовательские акаунты. Или, иными словами, кто определяет пользователей системы и каким образом.

В большинстве дистрибутивов Linux в ходе инсталляции создаются минимум две учетные записи: администратора (поскольку имя его фиксировано — root, то при создании запрашивается только пароль) и обычного пользователя, с определением его имени и пароля (в некоторых случаях для такого пользователя можно задать еще и основную группу). И потому может показаться, что пользователи возникают в системе сами собой, по принципу непорочного зачатия. Однако это не так.

Правда, пользователь root действительно существует в системе. Вернее, пользователь с UID 0. Как уже говорилось, сама по себе система об именах пользователей не имеет никакого понятия — за их соответствие идентификаторам отвечает главная системная библиотека glibc. И в процессе установки системы возможна ситуация (а иногда она и действительно возникает, например, в ходе самостоятельной сборки Linux from Scratch, то есть с

нуля), когда система еще не догадывается о существовании пользователя с именем root. И что же? Она прекрасно справляется с распознаванием администратора по его нулевому идентификатору, наделяя соответствующими полномочиями.

Все же прочие пользователи являются порождением нашего бога из машины — root-оператора. Причем косвенно это относится даже к псевдопользователям-демонам (почему — станет ясно со временем). А уж обычные реальные и виртуальные (почтовые и FTP-клиенты) пользователи порождаются с помощью root непосредственно.

Впрочем, сейчас нас волнуют только реальные пользователи. Для их создания (то есть учреждения соответствующих учетных записей) в POSIX-системах существует ряд инструментов (вплоть до обычного текстового редактора), о которых речь пойдет в ближайшей интермедии. Пока же замечу только, что в процессе создания акаунта обязательно задается только его имя и (или) идентификатор. Хотя дополнительно можно (а обычно и нужно) определить также его домашний каталог, пользовательскую среду, основную и дополнительные группы, а также пароль.

Интермедия: средства управления акаунтами

Разговор о пользовательских акаунтах логично завершить описанием средств управления, позволяющих создавать учетные записи, объединять их в группы, изменять атрибуты и т. д. Однако разговор на эту тему требует знаний об атрибутах процессов, описанных в *главе 7*, атрибутах файлов, рассмотренных в *главе 8*, а также представления о принципах работы в командной строке, составляющих предмет *главы 12*. Которые и рекомендуется предварительно прочитать совсем начинающему пользователю.

Средства для управления акаунтами несколько различаются в Linux и в BSD-системах. Причем BSD-утилиты администрирования пользователей и групп являются существенно более мощными, разнообразными, удобными и универсальными, почему и сконцентрируем на них основное внимание, делая соответствующие оговорки относительно аналогичных по назначению средств Linux.

Начнем с создания пользователей. Самое простое на сей предмет средство — команда `adduser`, тем более, что утилита с этим именем имеется во всех POSIX-системах, по крайней мере, открытых. Для создания единичного акаунта ее можно запустить без опций:

```
$ adduser/.
```

В ответ она в диалоговом режиме последовательно запросит следующую информацию:

- **Username** — имя пользователя, требуется задать обязательно;
- **Full name** — теоретически имя и фамилия пользователя, его можно оставить пустым или ввести произвольную информацию, которая составит значение поля комментария в базе данных пользователей;
- **Uid** — идентификатор пользователя; если это поле оставить пустым, то он автоматически примет ближайшее свободное значение (для первого пользователя в системе — 1001);
- **Login group** — основная группа пользователя, по умолчанию — собственная, совпадающая с его именем;

- **Login group is group_name. Invite exp into other groups?** [] — включить ли пользователя в другие группы; если предполагается его доступ к полномочиям администратора, то здесь следует указать группу wheel;
- **Login class [default]** — очень важный атрибут (*см. следующую интермедию*); в наших условиях лучше всего задать класс `russian`;
- **Shell (sh csh tcsh) [sh]** — пользовательская оболочка; предлагаемый по умолчанию вариант (`sh`) — отнюдь не лучший, о чем подробнее я скажу чуть позже;
- **Home directory [/home/username]** — имя домашнего каталога пользователя;
- **Use password-based authentication? [yes]** — использовать ли авторизацию по паролю;
- **Use an empty password? (yes/no) [no]** — использование пустого пароля (по нажатию клавиши <Enter>);
- **Use a random password? (yes/no) [no]** — генерация произвольного пароля;
- **Enter password** — ввод пароля;
- **Enter password again** — и его повторение;
- **Lock out the account after creation? [no]** — закрыть ли доступ к акаунту после его создания.

После честного и откровенного ответа на все поставленные вопросы будет выведено сообщение такого рода:

```
Username   : exp
Password   : *****
Full Name  :
Uid        : 1002
Class      : russian
Groups     : exp
Home       : /home/exp
Shell      : /bin/sh
Locked     : no
OK? (yes/no):
```

Если с этим согласиться, последует сообщение об успехе процедуры и предложение создать еще один акаунт:

```
adduser: INFO: Successfully added (exp) to the user database.
Add another user? (yes/no):
```

Однако если планируется создание серии однотипных учетных записей, целесообразно сначала выполнить настройку команды `adduser`, для чего ее следует запустить таким образом:

```
$ adduser -C
```

После чего последуют запросы о некоторых общих атрибутах, которые будут в дальнейшем распространяться по умолчанию на все создаваемые учетные записи. А именно:

- **Login group** — здесь можно задать общую группу для всех пользователей, например, users (сама группа должна быть создана предварительно, об этом далее);
- **Enter additional groups** — список дополнительных групп, к которым будут приписаны все вновь создаваемые пользователи (например, wheel operator);
- **Login class** — класс пользователей по умолчанию, взамен `defaults`; как я уже говорил, в качестве такового в наших условиях целесообразно определить `russian`;
- **Shell (sh csh tcsh)** — командная оболочка (login shell) по умолчанию; если вы предпочитаете пользоваться каким-либо шеллом, здесь не указанным (например, zsh), его также можно указать, внеся предварительно имя и полный путь к нему в файл /etc/shells;
- **Home directory** — если планируется использование домашних каталогов, отличных от /home/username;
- **Use password-based authentication?** — общие условия парольного доступа;
- согласие с введенными данными и (или) их правка.

Определенные таким образом общие атрибуты сохраняются в файле /etc/adduser.conf, и введенные нами ответы на их запрос в дальнейшем будут предлагаться по умолчанию.

К сказанному остается добавить только, что команда `adduser` автоматически сама проверяет имя и идентификаторы пользователя на предмет уникальности, предлагая для последних подходящие значения (обычно — следующие по порядку за ID предыдущего зарегистрированного пользователя), создает домашний каталог пользователя и копирует туда "скелеты" конфигурационных файлов (они берутся из файлов каталога /etc/skel, по умолчанию он пуст, и созданием соответствующих "скелетов" следует озаботиться самому), после чего делает соответствующие записи в базах данных пользователей — файлах /etc/passwd, /etc/master.passwd, /etc/group и /etc/master.group.

Кроме диалогового режима, команда `adduser` имеет и режим чисто командный, когда все пользовательские атрибуты задаются опциями командной строки. Ознакомиться с ними можно на странице документации:

```
$ man 8 adduser
```

Команда `adduser` (или `useradd`) имеется и в Linux, однако там возможности ее несколько меньше. В частности, она не создает автоматически домашний каталог пользователя, это следует сделать вручную командой

```
$ mkdir /home/username
```

Да еще и позаботиться об установке его принадлежности пользователю и группе:

```
$ chown -R username /home/username && \
        chgroup -R users /home/username.
```

Впрочем, это можно сделать и в один прием:

```
$ chown -R username:users /home/username
```

Использование `adduser` — наиболее автоматизированный, но и наименее гибкий способ управления учетными записями. Основное его назначение — создание либо единичного акаунта, либо многочисленных учетных записей со сходными атрибутами и профильными файлами пользователей, "скелеты" которых заблаговременно создаются в каталоге /etc/skel.

Другая крайность — внесение всех данных пользователя в файл /etc/master.passwd вручную (в текстовом редакторе) и дальнейший запуск команды `pwd_mkdb`, которая в форме

```
$ pwd_mkdb -p /etc/master.passwd
```

внесет соответствующие изменения в остальные три файла, имеющие отношение к акаунтами. При этом поле пароля следует оставить пустым, поскольку любой набор символов в нем развертыванию в реальный пароль не поддается. И в дальнейшем не забыть определить пароль специальной командой `passwd`. Кроме того, в реально многопользовательской (то есть не настольной) системе следует быть уверенным, что никто другой в этот же момент времени не занимается изменением учетных записей, иначе трудно предсказать, чьи модификации возымеют силу. А заниматься этим может не только другой администратор системы, но и пользователь, изменяющий свой пароль или командную оболочку.

Далее, существует программа vipw, совмещающая оба процесса: вызов текстового редактора для внесения вручную изменений в файл /etc/master.passwd, а по выходе из него — модификацию баз пользовательских

записей командой `pwd_mkdb`. В качестве редактора используется значение переменной `EDITOR` в профильном файле администратора. Если такового нет — вызывается классический редактор всех UNIX-систем — vi (откуда и название программы).

Наконец, самое мощное средство для управления учетными записями пользователей вообще — команда `pw`. Это — специфичная для BSD-систем, универсальная утилита администрирования учетных записей пользователей и групп — их создания, изменения и удаления. Для создания нового пользователя ее проще всего запустить в автоматическом режиме — в форме

```
$ pw useradd username
```

после чего она сама подберет подходящие (то есть ближайшие по порядку свободные) идентификаторы пользователя и группы, поля общих сведений, домашнего каталога и командной оболочки заполнит некими значениями по умолчанию (User ID, /home/username, /bin/sh соответственно), прочие же оставит пустыми. Сам домашний каталог она тоже не создаст — для этого она потребует формы

```
$ pw useradd username -m
```

во исполнение которой заодно будут скопированы и "скелетные" файлы.

Кроме этого, команда `pw` может создавать учетную запись пользователя с заранее предопределенными атрибутами, описанными в файле /etc/pw.conf. Правда, для этого такой файл необходимо предварительно создать командой

```
$ pw useradd -D
```

В первозданном виде он будет содержать все те же сведения по умолчанию. Однако его легко отредактировать в текстовом редакторе для внесения всех необходимых данных, которые отныне и будут выступать в качестве умолчания для команды `pw`. К слову сказать (и это очень ценное свойство данной команды), учетные записи ранее созданных пользователей могут быть приведены в соответствие с новой их "скелетной" схемой с помощью команды

```
$ pw useradd -D username
```

Ничто не мешает и задать несколько таких конфигурационных файлов (например, /etc/pw1.conf, /etc/pw2.conf и т. д.) для создания учетных записей пользователей разных категорий (например, реальных, виртуальных или пользователей с различными значениями атрибута `class`). Приведение конкретной учетной записи к какой-либо из принятых схем после этого можно выполнить командой

```
$ pw adduser username -C /etc/pw1.conf.
```

Она создает учетную запись нового пользователя с именем newuser и его домашним каталогом /home/newuser, в который копируются конфигурационные файлы командной оболочки. Файлы, подлежащие копированию, определяются содержимым каталога /etc/skel и могут быть отредактированы администратором в текстовом редакторе.

Некоторые атрибуты учетной записи пользователя, будучи раз созданными, могут быть изменены в дальнейшем, причем некоторые — и самим пользователем (не только root-оператором). Наиболее часто такая необходимость возникает в отношении пароля. Во-первых, как уже было сказано, не всегда он задается при создании учетной записи. Во-вторых, менять пароль время от времени рекомендуется и из соображений безопасности.

Для смены пароля предназначена команда `passwd`, общая для всех POSIX-систем. Запущенная от лица обычного пользователя без параметров и аргументов, она запросит для начала старый его пароль (если такового не было, этот шаг, естественно, пропускается), затем — новый пароль и его повторение (во избежание ошибки ввода).

Администратор системы с помощью команды

```
$ passwd username
```

может поменять пароль любого пользователя. Старый пароль его при этом не запрашивается, в общем случае администратор его и не знает (и восстановить из базы пользовательских записей простым способом не может).

Далее, пользователь может изменить еще два атрибута — общие сведения о себе (то есть поле комментария) и командную оболочку. Для этого он вправе использовать команду `chpass`, которая вызовет в текстовом редакторе (определенном переменной `EDITOR` профильного файла пользователя, за отсутствием таковой — в редакторе vi) следующие сведения:

```
#Changing user database information for alv.

Shell: /bin/csh

Full Name: Alex Fedorchuk

Office Location:

Office Phone:

Home Phone:

Other information:
```

По завершении их модификации и выходе из редактора внесенные изменения будут зафиксированы в базах данных автоматически. Нетрудно дога-

даться, что для этого команда `chpass` должна иметь упоминающийся в *главе 8* атрибут суидности, в чем можно убедиться посредством:

```
$ ls -l /usr/bin/chpass

-r-sr-xr-x  6 root  wheel  32324 Jan 28 16:12 /usr/bin/chpass.
```

Кроме этого, для изменения командной оболочки пользователя во всех POSIX-системах служит команда `chsh`, требующая опции `-s`, значением которой служит полный путь к исполняемому файлу нового шелла, и аргумента — имени пользователя. Впрочем, если шелл изменяет сам пользователь, в последнем нет необходимости.

Другие изменения учетной записи требуют вмешательства администратора, который может обратиться к той же команде `chpass` с указанием имени пользователя как аргумента:

```
$ chpass username

Login: alv

Password: lhBESi9NHbsrg

Uid [#]: 1000

Gid [# or name]: 1000

Change [month day year]:

Expire [month day year]:

Class: russian

Home directory: /home/alv

Shell: /bin/csh

Full Name: Alex Fedorchuk

Office Location:

Office Phone:

Home Phone:

Other information:
```

Можно видеть, что root-оператору предоставлено гораздо больше возможностей для изменения атрибутов акаунта. Что, впрочем, не значит, что он может использовать их все. Так, изменение идентификатора пользователя эквивалентно созданию пользователя нового, а изменение имени просто поставит ему в соответствие тот же идентификатор. Ни то, ни другое не вызовет ничего, кроме путаницы. Идентификатор основной группы, вероятно, изменить можно, хотя трудно представить необходимость этого. Ну, а о пароле уже говорилось достаточно.

Так что фактически из недоступных пользователю полей изменению подлежат только класс и временные атрибуты акаунта. Почему последние подведомственны администратору — понятно. Отношение же к классу выведено из сферы компетенции пользователя, так как через его определение возможен очень широкий круг настроек, в том числе и связанных с разделением доступа, степенью безопасности и т. д.

Теоретически для изменения учетных записей в разных POSIX-системах применяются также команды `chfn` и `chsh`, но в BSD эти файлы — лишь иные имена для команды `chpass`, и потому вызов любой из них автоматически влечет запуск последней.

Внести изменение в учетную запись можно и командой `adduser` с соответствующими опциями, например:

```
$ adduser -shell shell
```

для изменения командной оболочки,

```
$ adduser -class login_class
```

для изменения класса пользователя — список возможностей можно получить по команде

```
$ adduser --help
```

Кроме этого, для изменения учетных записей может применяться и универсальная команда `pw` с параметром `usermod`, именем пользователя и соответствующими (весьма многочисленными) опциями, с которыми можно ознакомиться посредством

```
$ man 8 pw
```

Удаление учетной записи пользователя, на первый взгляд, задача очень простая: достаточно удалить соответствующую строку из файла /etc/master.passwd и запустить команду `pwd_mkdb` для синхронизации изменений, точно так же, как это делалось при создании акаунта. Или прибегнуть к программе vipw, которая сделает это сама. Однако и в том, и в другом случае будут удалены только сами учетные записи, а пользовательские каталоги, почтовые ящики, временные файлы, результаты всякого рода протоколирования действий (log-файлы) благополучно сохранятся, причем будут рассеяны по ряду каталогов. Кроме того, упоминания имени удаленного пользователя останутся в списках принадлежности к группам, конфигурационных файлах авторизации, систем доставки почты, почтовых и FTP-клиентов (и в иных местах). Что может создать проблемы, если в дальнейшем будет создан пользователь с тем же идентификатором и (или) именем.

Поэтому для бесследного истребления следов жизнедеятельности удаленного пользователя их следует сначала отыскать. Все принадлежащие ему файлы могут быть найдены командой `find`:

```
$ find / -user username
```

А добавив в конец строки опцию `-delete` (или перенаправив вывод по конвейеру команде `rm`), можно немедленно уничтожить их. Для поиска упоминаний имени в конфигурационных файлах можно прибегнуть к команде `grep`:

```
$ grep -e username /*
```

Однако ради чистоты исполнения в обоих случаях поиск должен проводиться, начиная с корневого каталога (о чем сигнализирует символ / в аргументах обоих команд). И это может занять немало времени.

Поэтому может оказаться целесообразным прибегнуть к команде `rmuser` для автоматизации процесса. В форме

```
$ rmuser username
```

она удалит не только учетную запись из базы данных, но также и домашний каталог, почтовый ящик, имя в списках принадлежности групп (/etc/group) и еще некоторых конфигурационных файлах.

И конечно, ту же работу в состоянии сделать и палочка-выручалочка администратора — команда `pw` в форме

```
$ pw userdel username -r
```

Следует обратить внимание на опцию `-r`, без нее `pw` удалит только пользовательскую запись из базы акаунтов.

Разумеется, существуют и средства для управления записями групп. Как уже говорилось, новая группа автоматически создается при заведении нового пользовательского акаунта. По умолчанию ее имя и численное значение идентификатора совпадают с именем и идентификатором пользователя. И для последнего эта группа в дальнейшем остается основной (то есть только о ней содержится информация в учетной записи пользователя).

Однако пользователь может принадлежать и к иным, дополнительным, группам: для этого его имя или идентификатор должен быть приписан к списку таких групп в файле /etc/group. Это может сделать администратор в любом текстовом редакторе (никаких дальнейших действий по синхронизации изменений не потребуется).

В полномочиях root-оператора также и создание новых дополнительных групп: просто добавить строку по образу и подобию любой существующей,

записав в последнее ее поле имена тех пользователей, которые должны быть ее членами. Тем же способом группа может быть и удалена.

И конечно, для управления группами можно применить универсальное средство `pw`. В форме

```
$ pw group add groupname
```

эта команда создаст новую группу,

```
$ pw group del groupname
```

удалит существующую, а посредством

```
$ pw group mod groupname -m username1 username2
```

или

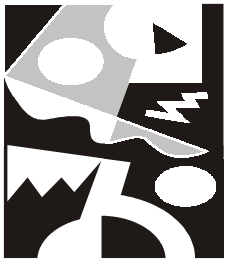
```
$pw group mod groupname -M username1 username2
```

дополнит или полностью перепишет членов группы соответственно. Наконец, с помощью

```
$ pw usermod  username -G groupname
```

можно приписать пользователя к указанной группе в индивидуальном порядке.

Глава 7



Процесс пошел

- Понятие процесса.
- Разновидности процессов.
- Атрибуты процесса.
- Жизнь и смерть процесса.
- Управление процессами.

Понятие процесса

Понятие процесса принадлежит к тем сущностям, кажущимся интуитивно ясными, которым, тем не менее, нелегко дать строгое (и при этом удобопонятное) определение. Наиболее распространенное из них — это представление процесса как программы в стадии ее выполнения.

Вместе с тем необходимо подчеркнуть, что процесс и программа — не идентичные понятия. Во-первых, существуют процессы, которым не соответствует никакой программы (то есть никакого исполняемого файла). Во-вторых, программа в ходе своего выполнения может породить более одного процесса. В-третьих, в рамках одного процесса может происходить замещение одной программы другой. И наконец, в-четвертых (и это, пожалуй, главное), при запуске программы порождение нового процесса предшествует собственно запуску программы.

Все сказанное может показаться настолько кучерявым, что вызывает естественный вопрос: а зачем обычному (конечному) пользователю вообще нужно знать о каких-то там процессах? Ведь пользователь Mac OS (BSD-системы в своей основе, тоже имеющей понятие процесса) спокойно обходится без этих знаний. Да и в Windows процессы, скорее всего, протекают, однако побуждений знакомиться с ними у пользователя нет.

Причин к рассмотрению понятия процесса с позиций пользователя несколько. Например, гносеологическая: просто не кажется лишним знать, что там у этого пользователя в системе, происходит (а, как уже говорилось, все, что происходит в системе, суть процессы).

Однако есть и несколько моментов, более важных практически. Во-первых, понимание взаимоотношений между процессом и запущенной программой очень не вредно при всякого рода настроечных мероприятиях, дабы не удивляться, почему командная оболочка bash (для примера) в консоли ведет себя одним образом, запущенная в окне терминала — другим, а в некоторых иных случаях — вообще третьим (например, категорически отказывается находить исполняемые файлы). А во-вторых, представление о процессах и способах влияния на них часто оказывается незаменимым в аварийных ситуациях. Так что давайте уделим этому вопросу некоторое внимание.

Разновидности процессов

Итак, повторюсь: процесс — это все, что происходит в системе. А все происходящее в системе либо порождено ее ядром (то есть как бы самой системой), либо одним из ее пользователей. Соответственно этому, процессы бывают системные и пользовательские. На первых мы здесь задерживаться не будем. Скажу только, что запуск системных процессов осуществляется ядром ОС, и их задача — управление ресурсами машины (доступом к процессору, памяти, свопингом и т. д.). Важно только помнить, что с системными процессами не коррелируют никакие программы в обычном понимании этого слова (кроме ядра, конечно, которое само по себе программа; но порождает она далеко не один процесс). То есть в системе не существует отдельного исполняемого файла, ответственного за запуск программы, выполняемой системным процессом (внятно выразился?): все они запускаются ядром, которое тоже являет собой просто исполняемый файл, хотя и не совсем обычный по своим функциям.

Из сказанного существует единственное исключение, и это процесс `init`, прародитель всех прочих процессов, почему его и относят к системным процессам. Хотя он и имеет ассоциированный с ним исполнимый файл (имя его всегда и везде в POSIX-системах — /sbin/init, хотя реально это могут быть разные программы), который вызывается по окончании загрузки системы и порождает все остальные пользовательские процессы. В том числе — процессы `getty`, открывающие терминал (за этот процесс в разных системах могут отвечать разные программы), и `login`, ответственный за авторизацию пользователя на этом терминале — ту самую программу, которая, как мы видели в прошлой главе, предлагает ввести пользователю свое имя и пароль.

Процессы `getty` и `login` относятся уже к пользовательским процессам. Которые, в отличие от системных, всегда ассоциированы с неким исполняемым файлом. Для процесса `getty` это будет, например, /usr/libexec/getty, а для процесса `login` — нечто вроде /usr/bin/login.

Процессы типа `getty` и `login` относятся к категории неинтерактивных, то есть тех, которые не привязаны к какому-либо терминалу, и на которые пользователь не может влиять непосредственно (общение с ними возможно только посредством сигналов, о которых будет говориться несколько позже). Неинтерактивные процессы именуются иногда демонами (daemon). Термин этот не несет в себе мистического смысла, представляя простую аббревиатуру (Disk And Execution MONitor). Демоны инициируются самой системой при загрузке обычным образом (запуском соответствующих программ) и функционируют в фоновом режиме, активизируясь при необходимости совершения некоего действия (печати, доставки почты и т. д.). Или, как в случае с `login`, авторизации пользователя.

Есть еще и процессы интерактивные, отличающиеся тем, что привязаны к некоторому терминалу, через который пользователь может с ними взаимодействовать. Так, авторизовавшись в системе, пользователь запускает тем самым свой первый процесс — свою командную оболочку (login shell; в Linux это скорее всего bash, в BSD-системах — /bin/sh или /bin/tcsh). Это процесс интерактивный, он привязан к терминалу, на котором авторизовался пользователь. И последний может с ним взаимодействовать определенным образом (вводить команды, например).

Атрибуты процесса

Как же процесс соотносится с запустившим его пользователем? Для ответа нужно ознакомиться с атрибутами процесса. Сразу следует отметить, что любой процесс, системный ли, пользовательский, интерактивный или неинтерактивный, имеет следующие атрибуты:

- идентификатор процесса (PID);
- идентификатор родительского процесса (PPID);
- имя хозяина процесса;
- идентификатор хозяина процесса (UID);
- идентификатор группы хозяев (GUID);
- приоритет;
- терминал.

Идентификатор процесса — это просто номер его в порядке запуска, от нуля до максимального значения, возможного в данной системе (количество одновременно запущенных процессов — величина конечная). Минимальный (первый) номер получает процесс `init` — первопредок всех остальных процессов в системе.

Идентификатор родительского процесса — это номер процесса, от которого произошел (посредством системного вызова `fork`) процесс данный, о чем будет сказано в следующем разделе.

Для рассмотрения смысла остальных атрибутов нам придется обратиться к команде `ps`, которая и выводит информацию о процессе в виде, доступном пониманию пользователя.

Команда `ps` имеет множество опций, из которых нам сейчас потребуется только `-u` (впрочем, символ дефиса в этой команде можно опустить): она выведет весьма подробную информацию о процессах, запущенных данным пользователем на текущей виртуальной консоли (с помощью иных опций можно получить сведения о чужих процессах и о процессах на иных консолях, но для нас пока это не важно). Опять же, далеко не все из этих сведений нам сейчас интересны, поэтому задержим внимание только на существенных в контексте данной главы.

Итак, сразу после авторизации пользователя в ответ на команду

```
ps -u
```

будут выведены две строки, состоящие из нескольких колонок, в числе которых — следующие:

```
USER       PID          COMMAND
alv       5408           -zsh
alv       5598           ps -u
```

Первая колонка указывает на имя хозяина процесса: как правило (хотя и не обязательно, и позднее можно будет видеть, что это важно), таковым выступает пользователь, этот процесс запустивший. Вторая колонка — идентификатор процесса, третья — имя программы, породившей процесс; очевидно, что в момент сразу после авторизации это будет только командная оболочка `-zsh` (символ дефиса перед именем показывает, что это не просто некая программа, а именно login shell, завершение ее эквивалентно окончанию сеанса работы данного пользователя). Ну и, конечно, сама команда `ps`, ей ведь тоже соответствует свой процесс, хотя и завершившийся. Так вот, именем хозяина определяются отношения процесса к файлам и иным процессам. То есть процесс получает те же привилегии, которые имеет его хозяин.

Сказанное о принадлежности процессов относится не только к интерактивным пользовательским процессам, но и к демонам. Хотя их, казалось бы, никто не запускает, но и они имеют своих хозяев. Это те самые псевдопользователи, о которых я упоминал в предыдущей главе.

Однако не обязательно хозяином процесса является запустивший его пользователь (или псевдопользователь). Это легко проверить, дав от лица обычного пользователя команду `passwd` для смены его пароля и посмотрев атрибуты соответствующего процесса:

```
root      5617          passwd
```

Как могло получиться, что обычный непривилегированный пользователь запустил процесс, хозяином которого оказался root? Это станет понятно из дальнейшего. Пока же скажу, для чего это нужно. Ясно, что любой пользователь должен иметь возможность сменить свой пароль. Однако для этого ему придется внести изменения в базу пользовательских акаунтов, а файл /etc/passwd, как мы помним, открыт для изменения только администратору. И потому пользовательский процесс, соответствующий команде `passwd`, наделяется правами, в обычной жизни этому пользователю недоступными.

Более строго это можно описать так. Как было сказано, система работает не с именем пользователя (по большому счету оно ей до лампочки), а с его идентификатором (UID). Именно он обычно наследуется процессом в качестве идентификатора хозяина (и потому называется реальным UID). Однако у процесса есть еще и так называемый эффективный идентификатор хозяина (EUID), который собственно определяет привилегии данного процесса. Обычно UID и EUID совпадают, но бывают и исключения из этого правила. Если посмотреть их с помощью команды

```
ps -l
```

для процесса `zsh`, в поле UID мы увидим цифру 1000 (в данном случае это UID пользователя `alv`). Однако для процесса `passwd` поле это будет содержать значение 0, то есть эффективный идентификатор его унаследован не от пользователя, а от root-оператора.

Кроме реального и эффективного идентификатора пользователя, для процесса устанавливаются еще два атрибута принадлежности — реальный и эффективный идентификаторы группы (GUID и EGUID соответственно). Смысл их аналогичен, но они наследуют права доступа процесса для группы, к которой принадлежит запустивший процесс пользователь. Именно это дает возможность пользователю слушать музыку или выключать машину, о чем говорилось в предыдущей главе.

Следующий атрибут процесса — его относительный приоритет (NICE), обозначаемый по-английски словосочетанием nice value (что интерпретируется обычно как степень дружелюбия, тактичности или даже альтруизма по от-

ношению к другим процессам). Он варьирует в диапазоне от −20 (минимальное дружелюбие, то есть высший приоритет) до +20 (максимальное дружелюбие, соответствующее низшему приоритету). Прикладные процессы в момент своего рождения получают приоритет 0 (то есть некую среднюю степень дружелюбия).

Относительный приоритет процесса лишь косвенно связан собственно с приоритетами выполнения процессов, которые перераспределяются системой динамически, в зависимости от ряда факторов (и относительный приоритет — лишь один из них). Относительный приоритет сам по себе остается неизменным на протяжении всего времени существования процесса, хотя, как будет показано далее, может быть изменен принудительно.

Наконец, последний из упоминавшихся выше атрибутов процесса — терминал, с которым он связан. Атрибут этот имеет смысл только для прикладных процессов, запущенных пользователем с определенной консоли, реальной или виртуальной, от которой он и наследуется (хотя результаты выполнения процесса могут быть перенаправлены на другую консоль).

Жизнь и смерть процесса

Каждый пользовательский процесс порождается каким-либо другим процессом — в основе их всех лежит процесс `init`. Как нетрудно догадаться, порождающий процесс называется также родительским, или материнским, а порожденный — дочерним (по-русски получается несоответствие грамматических родов, но "сыновний" звучит не очень привычно). И идентификатор родительского процесса (PPID) — также один из важных атрибутов пользовательского процесса, в чем мы немедленно и убедимся.

Каждый процесс в своем существовании проходит стадии зарождения, исполнения (часто с порождением новых процессов) и завершения (отмирания). Как все это выглядит в реальности, можно рассмотреть на примере самого первого пользовательского процесса, запускающего его командную оболочку по умолчанию (login shell). Для этого опять обратимся к выводу команды `ps` в форме

```
$ps l
```

для того, чтобы видеть идентификаторы родительских процессов (за их показ отвечает опция `l`). Дополнительно отфильтруем (что это такое, будет объясняться в *главе 12*) процессы, относящиеся только к одной из виртуальных консолей, на которой в данный момент никто не авторизован — для определенности, второй по счету (устройство `v1`):

```
$ ps l | grep v1
```

Для пустой, то есть не несущей пользовательского сеанса консоли вывод последней команды будет примерно таким (я опускаю столбцы, содержание которых в данный момент для нас не существенно):

```
UID     PID     PPID    COMMAND
0       491     1       /usr/libexec/getty
```

Мы имеем в этой консоли единственный процесс `getty`, принадлежащий суперпользователю (UID 0), имеющий идентификатор 491 и порожденный процессом `init` (пример приведен для DragonFlyBSD, и потому идентификатор его равен единице).

После же авторизации вывод той же команды приобретет такой вид:

```
UID     PID     PPID    COMMAND
0       491     1       login
1001    1949    491     -zsh
```

Он свидетельствует, что в нашей экспериментальной консоли запущено уже два процесса — `login`, показывающий, что на ней зарегистрирован некий пользователь, и командная оболочка последнего zsh.

И еще прошу обратить внимание: PID процесса, ассоциированного с командой `login`, тот же, что и для процесса, исполнявшего перед этим команду `getty`. Это свидетельствует, что авторизация пользователя не привела к запуску нового процесса; напротив, замещение одной программы другою произошло в рамках одного и того же процесса (одна из причин, почему не следует отождествлять процесс и программу). А вот идентификатор для `zsh` — другой: командная оболочка исполняется в порожденном (смотрим значение PPID) процессе.

Вытеснение одной программы другой в рамках единого процесса — типичный способ запуска новой программы. Так, если из командной строки `zsh` мы запустим, например, текстовый редактор `joe`, то в ответ на

```
$ ps l | grep v1
```

увидим следующую картину:

```
UID     PID     PPID    COMMAND
0       491     1       login
1001    1949    491     -zsh
1001    2208    1949    joe
```

Может создаться впечатление, будто бы процесс `zsh` непосредственно породил процесс `joe`. Однако это не так. К сожалению, проиллюстрировать динамику зарождения процесса не представляется возможным (по крайней мере, я не знаю, как), поэтому прошу поверить на слово — не мне даже, а авторам фундаментальных руководств по UNIX.

На самом деле родительский процесс перво-наперво порождает свою собственную копию — в данном случае еще один экземпляр командной оболочки zsh (не потому ли первый shell Борна и получил свое имя, что вызов программы из оболочки напоминает последовательную серию скорлупок?). И различаются в момент зарождения родительский и дочерний процессы фактически только своими PPID (ну и собственными идентификаторами, разумеется). И лишь затем в рамках дочернего процесса осуществляет запуск на исполнение собственно нашей новой программы.

В свою очередь, новый процесс также может выступать в качестве родительского. Так, редактор joe обладает способностью запуска из своей среды программ оболочки. В образовавшемся имитаторе командной строки можно запустить какую-либо другую программу, скажем, поиск файла командой `find`. Если в процессе поиска посмотреть распределение процессов (например, с другой виртуальной консоли), можно будет увидеть, что процесс `joe` породил как бы свою копию — но уже с иными идентификаторами (при этом PPID копии, как и следовало ожидать, равен PID оригинала).

Как станет ясным из дальнейшего, понимание механизма создания процессов и запуска программ очень важно. В одной из следующих глав речь пойдет о командных оболочках и вызываемых из них командах. Так вот, большинство команд запускается именно таким образом, как описано. Запущенная программа наследует свойства (программное окружение) не той командной оболочки, которая выступает в качестве пользовательской, а ее копии, которую команда заместила в рамках нового процесса. И вполне возможно, что программное окружение материнского и дочернего шеллов окажется разным, что не может не сказаться на выполнении команды.

Управление процессами

Это одна из причин, почему я в своем пользовательском введении вообще заговорил о процессах. Вторая же причина — то, что необходимость управления процессами время от времени возникает перед пользователем с неотвратимостью рока.

Управлять процессами можно двояко. Первый способ — это отдача прямой команды. Например, выход из редактора joe автоматически приводит к отмиранию соответствующего ему процесса, комбинация клавиш <Ctrl>+ +<Z> в большинстве случаев переводит процесс в фоновый режим и т. д.

Очевидно, что прямое управление возможно только в отношении интерактивных пользовательских процессов, имеющих реально запустившего их хозяина и управляющий терминал, с которого этот процесс запущен. С про-

цессами-демонами же пользователь может взаимодействовать только посредством посылки им неких сигналов. Сигналы — это также единственный способ ликвидировать безнадежно зависший процесс, который невозможно остановить штатными средствами.

Посылка сигналов осуществляется командой `kill`. Она требует указания имени сигнала (или соответствующего ему численного значения) и номера процесса, на который нужно оказать воздействие. Так, безнадежно зависший процесс можно попробовать убить следующим образом:

```
$ kill -15 ###
```

где `###` — номер (PID) убиваемого процесса, а `15` — численное значение для сигнала `TERM` (от terminated). Соответственно, команда эта может быть дана и в форме

```
$ kill -TERM ###
```

По получении сигнала `TERM` программа, ассоциированная с процессом `###`, пытается по возможности корректно завершить свою работу с записью несохраненных данных.

Однако это ей не всегда удается — в этом случае процесс, соответствующий зависшей программе, продолжает существовать. И тут приходится прибегать к более жесткой форме:

```
$ kill -9 ###
```

или

```
$ kill -KILL ###
```

Сигнал `KILL` не может быть проигнорирован никаким процессом — его получение знаменуется немедленным и неизбежным завершением работы зависшей программы. Разумеется, ни о каком сохранении данных при этом речи идти не может, поэтому применение этого сигнала — крайняя мера.

Есть и еще один способ управления процессами — изменение их приоритета с целью перераспределения ресурсов машины. Однако на практике пользователю с этим почти не приходится сталкиваться, и потому эту тему я затрону лишь вкратце.

Как уже говорилось, все пользовательские процессы (и большинство системных) по умолчанию запускаются с равным, как бы промежуточным, относительным приоритетом 0. И, соответственно, при многих запущенных задачах ресурсы компьютера (процессорное время и объем оперативной памяти) распределяются между ними (более или менее) равномерно.

При необходимости перераспределения ресурсов между программами можно изменить приоритет выполнения какой-либо из них. При этом пользователь в состоянии только уменьшить приоритет одного или нескольких про-

цессов, хозяином которых он является, администратор же имеет возможность и повысить приоритет любого процесса.

Делается это двояко. Если требуется запустить программу с приоритетом, отличным от обычного, используется команда `nice` с величиной изменения (nice value) в качестве опции (предваряемой дефисом) и именем программы в качестве опции. Так, команда

```
$ nice -5 joe
```

запустит редактор joe с приоритетом, уменьшенным на пять единиц. Если же опустить опцию, приоритет уменьшится на десять единиц. Для увеличения приоритета администратор (и только он) может дать команду

```
$ nice --7 joe
```

что приведет к росту приоритета (то есть уменьшению дружелюбия) на семь единиц.

Кроме того, приоритет может быть изменен для уже запущенных процессов с помощью команды `renice`, параметром которой является новое значение приоритета, а аргументом идентификатор (PID) процесса. Например, команда

```
$ renice 7 735
```

данная от лица пользователя — владельца процесса с PID 735, приведет к установке для него nice value, равного 7 (при условии, что прежний приоритет этого процесса был выше, например, 3 или 0). Администратор же может понизить приоритет того же процесса до значения nice value, равного 2, с помощью команды

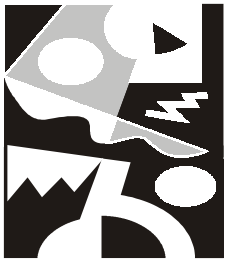
```
$ renice 2 735
```

или присвоить ему максимальный приоритет:

```
$ renice -20 735
```

В завершение разговора о процессах повторю, что пользователь может управлять только теми процессами, хозяином которых он является, — то есть собственноручно запущенными интерактивными. Администратор системы же располагает правами на управление всеми запущенными процессами, в том числе и неинтерактивными демонами.

Глава 8



Файл как он есть

- Что такое файл.
- Классификация файлов.
- Каталоги.
- Символические ссылки.
- Файлы устройств.
- Каналы и сокеты.
- Обычные файлы.
- Еще раз об именах.
- Право на файл
- О времени и о файле.

Все сказанное ранее может показаться пока не совсем понятным. Однако надеюсь, эта глава внесет ясность в вопрос о китах POSIX'ивизма (а равно баранах и прочих кашалотах). Нам только потребуется замкнуть разорванный вначале круг и поговорить о файлах. Тема эта очень многогранная и будет продолжена в следующих главах.

Как уже говорилось, все, что существует в POSIX-системе в статическом виде, являет собою файлы. Собственно, использование файловой системы как универсального интерфейса доступа ко всему, чему угодно — от устройств, физически подсоединенных к машине, до процессов, в системе протекающих, и есть один из критериев UNIX-подобия ОС (и ее соответствия стандартам POSIX).

Что такое файл

Помнится, на заре моего приобщения к компьютерам этот вопрос доставил мне немало пищи для размышлений. Встреченное где-то (не у Фигурнова ли?) определение файла как именованной области на диске для хранения

данных казалось интуитивно непонятным и не соответствующим собственным впечатлениям. И, по мере освоения POSIX-систем (сначала Linux, а потом и BSD), оказалось, что интуиция меня не подвела. Ибо файл — это отнюдь не область, расположен он не всегда на диске и уж ни в коей мере не привязан к какому-то имени.

В самом общем виде файл можно определить (хотя это лишь мое скромное мнение) как некую последовательность байтов, идентифицируемую тем или иным образом. Причем имя файла, как скоро станет ясным, отнюдь не единственный и даже не главный способ его идентификации. А то, где хранится эта самая байт-последовательность и какая именно информация передается теми самыми байтами, — дело уже десятое.

Предложенное определение может быть применено к любой операционной системе. Конкретно же в UNIX всякого рода к этому следует прибавить, что файл состоит как бы из двух частей, не обязательно находящихся в одном месте (а в общем случае как раз разобщенных).

Первая часть файла — так называемая область метаданных, именуемая по-английски inode (index node или information node — мне встречались обе расшифровки). Нормального русского эквивалента для этого термина не придумано (выражения типа "информационный узел" вряд ли прояснят существо дела, а "индексный дескриптор" назвать русским выражением трудно), и потому далее он будет использоваться без перевода. В inode содержатся некоторые сведения о файле (именуемые атрибутами файла), как-то:

- идентификаторы файла и устройства, на котором он расположен;
- тип файла;
- счетчик ссылок;
- его размер;
- атрибуты принадлежности, доступа, режима и времени.

Все эти характеристики файла будут рассмотрены в этом разделе.

Область данных файла заключает в себе собственно ту информацию, для хранения которой файл и предназначен. Она может быть самой различной: текстовые символы, двоичные коды и многое, многое другое. Однако характер этой информации для операционной системы POSIX-типа безразличен (чего нельзя сказать о пользовательских приложениях), она воспринимает их просто как последовательность байтов.

Область данных не является непременной принадлежностью файла. Существуют файловые системы, которые умудряются разместить данные маленьких файлов в области метаданных, чем достигается как быстрота доступа к хранимой информации, так и экономия дискового пространства. Потому что, несмотря на приведенную оговорку, файлы в основном все же хранятся на дисковых устройствах.

Внимательный читатель, привыкший под влиянием DOS/Windows к трактовке файлов в качестве именованных документов, обратил внимание на то, что среди атрибутов файлов нет одного, казалось бы, самого важного — имени файла. Как же они различаются?

Цели идентификации файлов служат числа, которые, как это ни странно, так и называются — идентификаторы. Таковых два: идентификатор устройства и собственно идентификатор файла. Со вторым все более-менее понятно — это просто порядковый номер файла (с некоторыми оговорками). И номер этот уникален для файловой системы. Именно по нему ОС считывает информацию из конкретного файла, то есть из области его данных, поставленных в однозначное соответствие с файловым ID из inode.

А понятие идентификатора устройства требует некоторого пояснения. Когда речь шла об установке системы (*см. главу 5*), вскользь было упомянуто о том, что Linux или BSD могут быть установлены в несколько дисковых разделов. И каждый из этих разделов будет нести свою файловую систему, образующую отдельные ветви единого файлового древа.

Так вот, нумерация файлов в каждом таком разделе не зависит от других, начинаясь с 2 (идентификатор "локального" корня). И потому в каждой ветви файловой системы могут быть (и неизбежно будут) файлы с одинаковыми идентификаторами. Чтобы отличать их друг от друга, введен такой атрибут — идентификатор устройства.

Файлы существуют, в том числе, и для того, чтобы пользователь с ними работал: открывал, просматривал, редактировал и т. д. А из предыдущей главы мы уже знаем, что все эти действия осуществляются процессами. То есть каждый процесс имеет некий набор открытых им файлов. Однако легко представить себе ситуацию, при которой два и более процессов открывают один и тот же файл. Как они различаются? Ведь inode открытого файла по определению один, а имя файла в контексте процесса не фигурирует.

Для этой цели предназначены файловые дескрипторы, которые не следует путать с дескрипторами индексными (inodes). Файловые дескрипторы (в дальнейшем условимся называть их дескрипторами просто) — это также числовые идентификаторы, которые присваиваются файлам по мере открытия их данным процессом. Причем нумерация их для каждого процесса отдельная. То есть файл с inode `xxx`, открытый процессом `A`, получит дескриптор `M`, а в процессе `B` ему же будет присвоен дескриптор `N`.

Однако одинаковые дескрипторы для одних и тех же файлов также не запрещаются. Примером этому — имеющие особое значение дескрипторы `0`, `1` и `2`. В любом процессе они присваиваются фиксированным файлам устройств — стандартного ввода, стандартного вывода и стандартного потока сообщений (/dev/stdin, /dev/stdout, /dev/stderr, соответственно). К ним мы

еще вернемся, а пока достаточно запомнить, что в случае настольной персоналки для ввода данных в процесс стандартно служит устройство под названием "клавиатура", а вывод процесса осуществляется на экран монитора, куда по умолчанию валится и весь поток сообщений (в том числе и об ошибках).

Классификация файлов

Старый пользователь DOS/Windows, не погрязший окончательно в метафоре файлов-документов, возникшей в Mac OS (где, надо отдать ей должное, метафора эта вполне оправдана) и внедренной в Windows, начиная с версии 95, помнит о существовании типов файлов, отличающихся своими расширениями: исполняемых (EXE, COM), пакетных (bat), текстовых (TXT), изображений в различных форматах (TIF там или GIF) и т. д. И потому будет весьма удивлен, когда узнает, что в UNIX все они не типизированы никак. И действительно: как ОС может отличить их друг от друга, если имени файла (и, следовательно, расширения) в атрибутах его нет и в помине? В этом смысле говорят, что в UNIX нет понятия типизации файла.

На самом деле средства для различения файлов с контентом разного рода существуют, но это прерогатива приложений, а не самой ОС, и о них сейчас речи не будет.

Тем не менее, под понятие файла в POSIX-системах подпадают столь разнородные объекты, что они неизбежно требуют классификации. Другое дело, что классификация эта основана на совершенно иных принципах. Выделяются следующие типы файлов (и тип в данном случае — это атрибут, описанный в соответствующем поле inode каждого файла):

- каталоги;
- символические ссылки;
- специальные файлы устройств;
- именованные каналы и сокеты;
- обычные, или регулярные файлы.

Пользователю постоянно приходится иметь дело с обычными файлами и каталогами, довольно часто — с символическими ссылками, время от времени — с файлами устройств. А вот с каналами и сокетами он практически не сталкивается (по крайней мере, у меня за все годы карьеры POSIX-пользователя такой необходимости ни разу не возникло). Пропорционально этому мы и уделим внимание перечисленным типам файлов. Начнем мы, однако, не с обычных файлов, с коими пользователь общается наиболее тесно, а с каталогов, и сейчас же станет ясно, почему.

Каталоги

Итак, каталоги (directory) в некотором смысле противопоставляются всем другим типам. Это специальные файлы, объединяющие другие файлы (и подкаталоги, называемые также вложенными каталогами) в обособленные друг от друга группы (по крайней мере, так они видятся пользователю в специальных программах управления файлами — файловых менеджерах).

В русскоязычной литературе по DOS/Windows каталоги обычно именуются директориями, а в последнее время в обиход вошел термин "папка" (folder). Однако для UNIX-систем термин "каталог" предпочтительнее, и я сейчас постараюсь объяснить, почему.

Принятая в Windows (и идущая исторически от Mac OS) метафора каталога как папки с документами способна создать впечатление (и подчас его создает), что каталог — это некое физическое вместилище включенных в него файлов. Но это не так. Каталоги суть не более чем списки имен входящих в них файлов и тех самых числовых идентификаторов, о которых речь шла в прошлом разделе (и по которым собственно данные и находятся системой). И потому метафора библиотечного каталога (или книжного оглавления) как нельзя лучше отражает физический смысл соответствующего понятия.

Роль каталогов трудно переоценить. Потому что имя файла в файловой системе UNIX существует только в составе каталога и никак иначе. Собственно же содержимое файла ставится в соответствие имени через идентификатор его inode, подобно тому, как это осуществляется в базах данных. Механизм соотнесения имени файла, его метаданных и данных носит название жесткой ссылки (hardlink), чтобы отличить его от ссылок символических — файлов особого типа, о которых речь пойдет в следующем разделе. Количество жестких ссылок и есть значение поля счетчика в inode файла. Очевидно, что возможный его минимум для большинства типов файлов — единица.

Из сказанного следует несколько важных свойств имен файлов. Идентификатора устройства, несущего данный файл, в каталоге нет. И потому имя файла (то есть элемент данных включающего его каталога) может существовать только в той же физической файловой системе (дисковом разделе), что и inode, с которым он связан жесткой ссылкой (и, разумеется, в той же, что и собственно область данных файла).

Во-вторых, на один inode с его данными может быть создано произвольное количество жестких ссылок (вспомним о соответствующем атрибуте — счетчике ссылок). То есть файл с одним и тем же физическим содержанием может выступать под рядом имен, и все они будут равноправны между собой. Или, напротив, под одним и тем же именем входить в состав разных каталогов.

Редактирование файла, вызванного под любым из его имен или из разных каталогов (например, текста в текстовом редакторе), будет приводить к изменению области данных файла, не зависящих от имени, и, следовательно, содержимое любого из таких файлов-двойников будет оставаться одинаковым. Более того, любое из имен такого файла можно удалить (то есть изменить содержимое включающего его каталога), что не окажет никакого влияния на остальные имена файла (и, тем более, на его реальное содержимое).

Подчеркну, что два имени файла, связанных жесткой ссылкой с одним и тем же inode, не являются копией один другого, поскольку при копировании файла создаются новые области метаданных и данных, которые в дальнейшем будут существовать абсолютно независимо друг от друга, то есть изменение содержимого копии или ее атрибутов никак не затронет исходный файл.

В-третьих, удаление файлов в UNIX происходит совершенно иначе, чем в DOS/Windows. А именно: файл считается удаленным, когда уничтожены все имена, ссылающиеся на идентификатор данного inode (то есть файл исключен из файловой системы), и закрыта последняя программа, к нему обращающаяся (завершен процесс, загрузивший данные файла в память, и уничтожен индексный дескриптор файла в этом процессе). В описании атрибутов файла это выражается в том, что счетчик ссылок его inode обнуляется. Разумеется, сами по себе данные, составляющие содержание файла, физически могут продолжать существовать на диске, но для системы они уже недоступны. А поскольку содержание файла оторвано от его имени, восстановление случайно удаленного файла по фрагменту имени оказывается невозможным.

Пока любой файл открыт, то есть существует ссылающийся на него процесс, он продолжает существовать, даже если имя его исключено из всех каталогов, и может быть записан, скопирован, переименован и т. д. То есть открытый каким-либо процессом дескриптор данного файла — гарантия его существования, по крайней мере, до завершения процесса. Именно поэтому я ранее сказал, что файл не обязательно имеет имя: в случае удаления открытого файла из каталога он некоторое время существует как бы безымянным, для поддержания его бытия достаточно открытого дескриптора, ассоциированного с inode.

Однако вернемся к каталогам. Как мы только что увидели, любой файл доступен для системы в том только случае, если он включен в файловую систему. То есть имя его приписано к определенному каталогу. Но ведь каталог — это тоже файл, и он тоже должен быть приписан к некоему каталогу более высокого уровня, который носит название родительского. И этот родительский каталог (символически он обозначается как "..") — один из элементов любого, даже не содержащего как бы файлов, каталога. А второй его непременный элемент — это он сам, то есть текущий каталог

(что символизируется как "."). То есть для каталога минимум возможных связей (значение поля счетчика в inode), в отличие от всех прочих типов файлов, равно 2.

Итак, каждый файл входит в какой-либо каталог, который, в свою очередь, включен в каталог более высокого уровня, и т. д. Вплоть до самого высокого каталога, не вполне логично именуемого корневым, за которым зарезервирован символ /. Вся эта конструкция называется деревом файловой системы (хотя на иллюстрациях в большинстве книжек это дерево растет обычно сверху вниз), файловой системой просто (как скоро станет ясным, это один из самых многозначных терминов) или, наконец, файловой иерархией. На последнем термине мы и остановимся, а к разъяснению его сути обратимся в *главе 10*.

Корневому каталогу файловой системы всегда присваивается идентификатор 2. Очевидно, что он не имеет родительского каталога, выше него в файловой иерархии не лежит ничего. Тем не менее, счетчик его ссылок также имеет два значения, ибо он сам выступает в качестве своего родителя. То есть ссылка на родительский каталог есть ссылка на него же.

Отдельные ветви файловой системы, расположенные в самостоятельных дисковых разделах, имеют свои корневые каталоги, также имеющие в качестве идентификатора 2. Однако, поскольку они различаются идентификаторами устройств (то есть разделов), путницы между ними и "главным" корневым каталогом не происходит (подробнее об этом речь пойдет в *главе 10*, рассказывающей о файловой иерархии).

Символические ссылки

Как уже было сказано, жесткая ссылка на файл может существовать только в той же файловой системе, в которой размещен его inode. Однако подчас возникает необходимость поместить имена одних и тех же файлов в каталогах, лежащих в разных файловых системах (и из главы о файловой иерархии станет ясно, зачем). Конечно, такие файлы можно просто скопировать. Однако это, помимо напрасной траты дискового пространства, влечет за собой и другие неудобства. В частности, нельзя гарантировать идентичности содержимого таких файлов-копий, а в ряде случаев именно идентичности от них и требуется.

Кроме требования нахождения в одной файловой системе, на жесткие ссылки накладывается еще одно ограничение: они могут быть созданы только на обычные или специальные файлы (например, файлы устройств), но не на каталоги. И потому не подходят для воспроизведения файловой иерархии, в чем подчас также возникает необходимость.

Для разрешения этих проблем был придуман особый тип файлов — символические ссылки (symlinks), часто называемые просто ссылками (links). Будем так поступать и мы. А для отличия от них жестких ссылок последние будут именоваться полностью.

В отличие от жесткой ссылки, представляющей собой просто еще одно имя для одного и того же набора метаданных и данных, символическая ссылка — это именно самостоятельный файл со своим идентификатором и прочими атрибутами, записанными в области метаданных, и со своими данными. Другое дело, что последние предоставляют очень ограниченную информацию: просто указание пути к собственно файлу или каталогу, на который эта ссылка указывает. Причем файл этот может лежать не только на иной файловой системе (дисковом разделе), но даже находиться на иной машине в локальной или даже глобальной сети. Не возбраняются и символические ссылки на каталоги, в этом случае содержимое каталога-ссылки будет точно воспроизводить внутреннюю иерархию исходного каталога.

Собственно говоря, на локализацию файла, исходного для символической ссылки, не накладывается практически никаких ограничений. Потому что при обращении к файлу-симлинку (например, при вызове его для редактирования в текстовом редакторе) происходит отслеживание пути до исходного файла, который и загружается данной программой.

Очевидно, что если исходный файл по каким-либо причинам недоступен (например, лежит на удаленной машине, связь с которой в данный момент отсутствует), обращение к нему окажется невозможным, последует просто сообщение об ошибке. Аналогично будет и при обращении к симлинку, исходный файл которого был удален — такие ссылки называются мертвыми.

Роль символических ссылок в UNIX очень велика. Файловые иерархии этого класса складывались исторически и, несмотря на ряд общих черт, со временем стали весьма различаться в отдельных клонах. В то же время мы помним, что все POSIX-совместимые системы должны уметь исполнять одни и те же программы, удовлетворяющие соответствующим стандартам. А программы эти могут искать требующиеся им для работы компоненты (в частности, системные библиотеки) в совершенно определенных каталогах, которые в разных UNIX-клонах вполне могут оказаться не только в разных местах, но и на разных файловых системах. И тут символические ссылки оказываются незаменимыми, достаточно продублировать ими требуемые ветви файловой системы в соответствующих местах, чтобы данное приложение спокойно находило все, что ему нужно.

Есть и другое применение символических ссылок. Некоторые программы требуют для своей работы не просто определенные компоненты, чаще всего — библиотеки, но и фиксированные их версии. В то же время установленная в данной системе версия может быть иной, более старой или более новой. В первом случае все ясно: программа, запрашивающая библиотеку

имя_рек.5 с версией имя_рек.4, скорее всего, работать не будет (хотя бывают и исключения).

А что делать, если версия библиотеки более новая? Ведь большинство разработчиков программ для POSIX-систем обычно свято придерживаются принципа обратной совместимости. То есть функциональность библиотеки имя_рек.6 будет, как правило, заведомо достаточной для работы программы, требующей версии имя_рек.5 (исключения и здесь бывают, но они не так уж часты). Однако просит-то наша программа именно версию имя_рек.5 и, не обнаружив ее, выдает сообщение об ошибке.

Конечно, теоретически правильно написанная программа должна требовать версии не ниже некоторой определенной, однако на практике такая ситуация встречается сплошь и рядом. И разрешается она часто достаточно легко — посредством механизма тех же символических ссылок. То есть обычно достаточно создать симлинк библиотеки имя_рек.5 на имя_рек.6, чтобы обмануть наше приложение и заставить его работать правильно.

Сказанным не исчерпывается сфера применения символических ссылок. В частности, они часто требуются для корректного доступа рядя программ к определенным файлам устройств. Если внимательно просмотреть каталог /dev любой POSIX-системы, там почти наверняка обнаружатся такие файлы, как cdrom, mouse, modem, audio, и еще несколько им подобных. Так вот, это обычно не более, чем символические ссылки на файлы реальных устройств, созданные для единообразия доступа к ним, позволяющие абстрагироваться от конкретного его интерфейса (положения на IDE-контроллере, номера последовательного порта). Впрочем, это уже тема следующего раздела.

Файлы устройств

Файлы устройств соответствуют различным присутствующим в системе устройствам. Устройства эти могут быть реальными (жесткие диски, принтеры и т. д.), а могут — псевдоустройствами, с которыми не ассоциировано никакого "железа" (например, знаменитое устройство /dev/null, не содержащее ничего, или /dev/zero, содержимое которого — сплошные нули).

С другой стороны, выделяются файлы символьных устройств и устройств блочных. К первым возможен только последовательный (посимвольный) доступ. Примером их являются последовательные и параллельные порты, а также терминалы, реальные и виртуальные. К блочным устройствам можно осуществлять произвольный доступ, обмен информацией с ними осуществляется блоками фиксированной (реже переменной) длины. Блочные устройства представлены, в частности, жесткими дисками и другими накопителями.

Нужно заметить, что в BSD-системах (FreeBSD, DragonFlyBSD) в последнее время отмечается отказ от поддержки устройств блочного типа: диски и

прочие накопители предстают в них как устройства символьные. Что, впрочем, не отменяет блочной организации лежащих на них файловых систем, которые будут предметом рассмотрения в следующей главе.

Для файлов устройств, кроме обычной идентификации, есть и собственная система опознавания: по номерам — основному, или старшему (major), определяющему класс устройств (например, 12 — старший номер устройств, относимых к классу виртуальных консолей; это для DragonFlyBSD, в иных системах старший номер того же устройства будет иным), и дополнительному, или младшему (minor). Последний является просто порядковым номером данного устройства в своем классе (например, для первой виртуальной консоли он будет равен 0, для второй — 1 и т. д.).

Файлы устройств могут быть фиксированными и динамическими. В первом случае они создаются (как правило, при инсталляции системы) в комплекте на все случаи жизни и включают в себя почти все, что теоретически может быть подсоединено к машине. Например, на машине с обычным IDE-контроллером, имеющим два канала на два устройства каждое, можно обнаружить четыре файла, соответствующие жестким дискам, для каждого по четыре файла, отвечающих теоретически возможным первичным разделам, большое количество файлов терминальных и псевдотерминальных устройств, и т. д. Хотя большая часть из этих файлов не соответствует никаким реально присутствующим устройствам.

В то же время некоторые на самом деле задействованные устройства могут не иметь ассоциированных с ними файлов. В этом случае пользователь вынужден пополнять их список, для чего предназначены специальная команда (`mknode`) и построенный на ее основе сценарий /dev/MAKEDEV.

Динамическое создание файлов устройств применяется в тех системах, которые используют файловую систему устройств (devfs) или механизм udev. Оно происходит каждый раз заново в процессе старта машины на основе тестирования наличного оборудования. И тут уже файлы создаются только для тех устройств, которые присутствуют на самом деле (и, что немаловажно, поддерживаются текущей конфигурацией ядра, в противном случае устройства все равно что нет).

Понятие файла устройств в ОС UNIX-семейства позволяет пользователю (и программам) общаться с железом, не вдаваясь в подробности того, как оно устроено. В разделе о командах будет показано, что вывод программ может быть перенаправлен в произвольный файл. Или, напротив, содержимое файла может быть считано в качестве входных данных. А поскольку устройства — это такие же файлы, то понятие перенаправления распространяется и на них. То есть для распечатки некоего текста его достаточно перенаправить в файл печатающего устройства (например, /dev/lpr — принтер на параллельном порту).

Каналы и сокеты

Специальные виды файлов — именованные каналы (named pipe) и сокеты (socket) — предназначены для обмена данными между процессами. Они важны для разработчиков ПО, пользователь с ними, как правило, напрямую не общается. Однако знать о факте их существования не вредно.

Обычные файлы

Наконец-то мы подобрались к типу, наиболее важному для пользователей, — обычным, или регулярным, файлам. Рассмотрение их я отложил напоследок по одной причине — дать определение обычного файла можно только методом исключения. К типу обычных файлов отнесено все, что не является каталогами, символическими ссылками, файлами устройств, каналами и сокетами.

Легко понять, что множество обычных файлов охватывает все то, что в DOS/Windows рассматривалось бы как самостоятельные типы: исполняемые двоичные, файлы сценариев оболочки, обычные тексты, файлы изображений, файлы в форматах специальных программ (например, документы word-процессоров или электронных таблиц). Общее между ними одно: содержимое всех их может быть непосредственно просмотрено либо стандартными командами типа `cat`, `less`, `more`, либо программами, в которых они были созданы. Тогда как непосредственный просмотр содержимого файлов прочих типов, как правило, невозможен: доступ к ним можно получить только косвенными путями.

Однако под понятие регулярных файлов подпадают и весьма неожиданные вещи. Выше я упоминал, что в UNIX в качестве файлов предстают не только устройства (сущности статические), но даже протекающие в системе процессы. Для представления последних задействуется файловая система процессов — procfs. Так вот, как это ни странно, файлы процессов суть также обычные файлы, многие из которых можно легко просмотреть с помощью обычных средств доступа к файлам — командами типа `cat`, `more`, `less`.

К слову сказать, существование файлов процессов опровергает еще один момент из приведенного в начале определения файла — будто бы это обязательно область на диске. Файловая система процессов со всем своим содержимым никакого места на диске не занимает, являясь лишь отображением происходящего в системе в виде, пригодном для восприятия универсальным интерфейсом файловой системы.

Еще раз об именах

На самом деле в UNIX на длину и формат имени файла накладывается весьма мало ограничений. Длина имени определяется форматом каталога, принятым в данной файловой системе. И ныне большинство их реализаций поддерживают имена длиной до 255 символов.

В именах файлов абсолютно запрещенными к использованию символами, с точки зрения ОС, являются лишь прямой слэш (/), зарезервированный как имя собственное корневого каталога и разделитель каталогов, и `NULL`, то есть отсутствие всякого символа. Все прочие символы алфавитные, цифровые и специальные символы теоретически могут входить в имена.

Важно помнить, что имена файлов в POSIX-системах чувствительны к регистру образующих их алфавитных символов. То есть README и readme всегда будут восприниматься как имена разных файлов.

Однако практически на имена файлов накладывают свои ограничения еще и некоторые прикладные программы. В первую очередь это — командные оболочки, которые целый ряд символов, не принадлежащих к числу алфавитно-цифровых, интерпретируют по-своему — как коды управления, регулярные выражения и т. д. В *главе 12*, повествующей о командах, будет показано, что специальное значение символов можно отменить. Тем не менее, такие знаки, как *, !, #, @ и прочие из верхнего ряда клавиатуры, за исключением подчеркивания (_), не рекомендуется включать в имена файлов. К слову, сам символ отмены специального значения (обратный слэш \) также в именах лучше не использовать...

Зато точек имя UNIX-файла может содержать сколько угодно, ведь мистического значения (отделения собственно имени от расширения) в этот символ не вкладывается. И потому здесь обычными являются имена типа archive.tar.gz или archive.tar.bz2 — файлы архивов, образованных утилитой tar и компрессированных программами gzip или bzip2 соответственно (в UNIX понятия архивации и компрессии суть вещи обычно разные).

Особую роль играет точка в начале имени файла — например, ".cshrc". Файлы такого вида так и называются — dot-файлами. Имена их недоступны при просмотре содержимого каталогов без специальных средств (о чем речь пойдет в ближайшей интермедии). Обычно в качестве dot-файлов выступают конфигурационные файлы пользователей.

Поскольку одноименные файлы (с разным или одинаковым содержанием) вполне могут существовать в разных каталогах, имя файла само по себе не позволяет однозначно идентифицировать его положение в файловой системе. Отсюда возникает понятие пути к имени, по-английски — pathname, что иногда переводится как полное, или составное имя. Оно, помимо собственно именования файла, указывает всю цепочку подкаталогов, отделяющих

его либо от корневого каталога (абсолютный путь к имени), либо от каталога текущего (относительный путь).

Абсолютный путь отсчитывается от "главного" корневого каталога файловой системы вообще и имеет следующий вид:

/dirname/filename,

где dirname — подкаталог корневого каталога (первый /). Относительный путь отсчитывается от каталога, выступающего текущим для данного процесса. В предположении, что текущим является каталог /dirname1, относительный путь к файлу /dirname2/filename будет выглядеть так:

../dirname2/filename

Элементы пути к имени (промежуточные подкаталоги) в любом случае разделяются символами прямого слэша, именно поэтому он и является абсолютно запрещенным к употреблению собственно в именах.

На полную длину пути также накладываются системные ограничения. Впрочем, они весьма мягкие — верхний лимит pathname в Linux, например, составляет 4096 символов. Так что путь к имени файла может быть сколь угодно длинным — в разумных пределах, разумеется. А вполне возможная в Windows (точнее, в одной из ее файловых систем — vfat) ситуация, когда создается файл, доступ к которому в дальнейшем оказывается невозможным вследствие превышения лимита длины пути, в UNIX практически исключена.

Вспомним о символических ссылках: путь к именам файлов любого другого типа и составляет содержимое файлов этого типа. Соответственно, ссылки могут быть абсолютными, определенными от корневого каталога, и относительными, задаваемыми от каталога, текущего в настоящий момент. Абсолютный симлинк всегда однозначно определяет файл, на который он ссылается. Симлинк же относительный зависит от того, какой каталог выступал в качестве текущего каталога в момент его создания. И при перемещении относительной ссылки в иной каталог она станет неработоспособной.

И еще о текущем каталоге. Для пользователя, заставшего времена DOS, представляется очевидным, что любой файл (например, исполняемый exe) ищется в первую очередь в текущем каталоге. В UNIX это не так: в большинстве систем поиск в текущем каталоге или не производится вообще, или осуществляется в последнюю очередь. И потому, если требуется вызвать исполняемый файл из текущего каталога, его положение должно быть указано в явном виде:

```
$ ./filename
```

Это имеет глубокое внутреннее обоснование, в том числе и с точки зрения безопасности, и к этому вопросу мы еще будем возвращаться.

Право на файл

В начале этого параграфа были вскользь упомянуты прочие атрибуты файла: принадлежности, доступа и режима. Это очень важные понятия любой UNIX-системы. И одно из тех понятий, которые (знаю по собственному опыту) психологически трудны для понимания пользователя, пришедшего из мира DOS.

Впрочем, смысл атрибутов принадлежности интуитивно понятен — каждый файл в системе: а) имеет своего хозяина; б) приписан к какой-либо группе пользователей, и в) находится в неких отношениях со всеми прочими. В качестве хозяина файла обычно выступает его создатель. Точнее, хозяин процесса, этот файл создавшего. А еще точнее — пользователь, идентификатор которого был унаследован процессом, создавшим файл, в качестве эффективного UID. То есть если процесс этот по каким-либо причинам получил привилегии root-оператора, то и созданный им файл будет иметь своим хозяином его, а не пользователя, на самом деле процесс запустившего.

С групповой принадлежностью также не все однозначно. В большинстве дистрибутивов Linux каждый файл приписывается по умолчанию к основной группе пользователя — хозяина файла. Однако такое положение не является непременным. Существует еще и наследование групповой принадлежности в BSD-стиле, когда новообразованный файл оказывается принадлежащим той же группе, что и его родительский каталог.

Именно отнесение файла к некоей группе широко используется для разграничения доступа, организации коллективной работы и прочих задач администрирования. Ну и все "прочие" также имеют некие права на файл, и не обязательно более узкие, чем члены его группы (или даже хозяин).

С атрибутами доступа дело несколько сложнее. Основных из них также три: атрибут чтения, изменения, исполнения, причем смысл их различается для каталогов и всех прочих файлов. Начнем со вторых.

Право на чтение (read) файла — это возможность просмотреть его командой `less` (естественно, только для обычного файла, файлы устройств, сокеты или каналы просмотреть таким образом нельзя, а попытка просмотра символической ссылки вызовет ее файл-источник), открыть в текстовом редакторе или какой-либо прикладной программе. Атрибут изменения (write) также понятен, он дает право изменить содержимое файла вплоть до полного удаления (содержимого, но не файла!). А атрибут исполнения (execute) означает возможность запустить скрипт: именно присвоение этого атрибута волшебным образом превращает набранную в текстовом редакторе последовательность команд в сценарий оболочки.

Для каталога же атрибут чтения позволяет просмотреть командой `ls` (или каким бы то ни было иным способом) его содержимое. А по предыдущему

разделу мы помним, что все содержимое каталога — это просто список имен входящих в него файлов. И потому право на изменение каталога — это возможность этот самый список модифицировать. Например, удалить из него какое-либо имя или создать новое (путем перемещения, копирования файла или записи некоторых данных из программы). А поскольку больше имя файла нигде не фигурирует, лишившись его, файл становится недоступным для системы, то есть удаленным. При этом прошу обратить внимание: никаких особенных прав на сам файл не требуется, в общем случае можно удалить файл, не имея на него прав не только изменения, но даже чтения, достаточно обладать правом на изменение каталога. При этом, зная точно имя удаляемого файла и полный путь к нему, можно не иметь и права на чтение каталога, к которому файл приписан.

Наконец, атрибут исполнения для каталога дает право входить в него командой `cd`. Всего-то, но без этого права какие-либо операции внутри каталога становятся затруднительными. Хотя и не невозможными: по аналогии с правом чтения, легко догадаться, что файл из каталога можно удалить, не входя в него — достаточно знать имя файла и путь к нему.

Атрибуты доступа сцеплены с атрибутами принадлежности. Для каждой категории обладателей файла (даже прочих, они тоже в определенной мере обладатели файла) может быть установлено свое (и теоретически любое) сочетание прав на доступ к нему. Так, пользователь может иметь право на чтение, изменение и (если это оправдано по смыслу) исполнение файла или каталога (очевидно, что право исполнения TIFF-файла ни малейшего смысла не имеет), группа и прочие — иметь права чтения и, если нужно, исполнения: это схема обычного назначения по умолчанию атрибутов вновь создаваемого файла.

Можно запретить любой доступ к своим файлам для прочих, оставив их только для группы особо избранных. А можно, напротив, запретить право чтения и исполнения для группы, сохранив их для прочих. И потому, если чтение файла запрещено для членов группы, к которой файл приписан, попытки ознакомиться с ним будут отвергнуты, тогда как все прочие прочтут файл беспрепятственно...

Кроме атрибутов принадлежности и доступа, которые присваиваются файлу (или, напротив, отнимаются у него) в обязательном порядке, он может иметь и три дополнительных атрибута, называемых иногда атрибутами режима. Для названия их удобочитаемого русского перевода не существует. Это атрибуты:

- SUID (Set User IDentificator), иногда именуемый битом суидности (не путать с суицидом);
- SGID (Set Group IDentificator);
- sticky (можно трактовать как атрибут "липкости" или "сохранности").

Первые два имеют смысл только для исполняемых обычных файлов, последний — в основном для каталогов. Атрибут суидности обеспечивает механизм, благодаря которому пользователю удается изменить свой пароль без помощи администратора. А именно: файл, которому он присвоен, при запуске на выполнение (именно потому он имеет смысл только для файлов с атрибутом исполнения) порождает процесс, наследующий эффективный UID не пользователя, его запустившего, а хозяина файла, правами доступа которого и определяются привилегии процесса. А поскольку хозяином подавляющего большинства файлов за пределами домашних каталогов является root, именно его-то права процессу обычно и достаются. Смысл атрибута SGID аналогичен, только тут процессом наследуется не эффективный UID пользователя, а эффективный GID группы, к которой приписан помеченный файл.

Атрибут sticky присваивается обычно каталогам и влечет за собой невозможность удаления из него файла кем бы то ни было, за исключением владельца файла, ведь в обычном случае для этого достаточно иметь права доступа не к файлу, а к каталогу. Установка его целесообразна для каталогов, хранящих всякого рода временные данные (типа /tmp и некоторых подкаталогов ветви /var), права доступа к которым по умолчанию (и по смыслу) допускают модификацию всеми пользователями. Однако если право на запись временных файлов пользователь вряд ли сможет использовать во вред, то удаление из таких каталогов чужих файлов вряд ли будет приветствоваться их хозяином. И именно для предотвращения такой ситуации предназначен атрибут sticky.

Сведения обо всех атрибутах файла можно получить посредством команды `ls` в "длинном" формате (с опцией `-l`). Или от какого-либо файлового менеджера. Хотя, на мой взгляд, команда `ls` ничуть не менее информативна и выразительна. Что я и попытаюсь продемонстрировать в ближайшей интермедии.

В обычном выводе команды

```
ls -l
```

имена пользователя и группы находятся в третьем и четвертом полях, а атрибуты доступа объединены в первом. Он имеет вид

```
-rwxrwxrwx
```

и расшифровывается следующим образом:

- первая позиция последовательности — тип файла (символ дефиса в примере означает, что мы имеем дело с обычным файлом, для каталога там был бы символ `d` — от directory, для символьного устройства — символ `c`, для блочного — символ `b`);
- следующие три символа определяют атрибуты доступа для хозяина файла: `r` — чтение, `w` — изменение, `x` — исполнение;

- две последние тройки символов суть то же самое, но для группы и прочих соответственно.

То есть в данном примере фигурирует исполняемый файл, открытый на чтение, запись и запуск для всех. Если же у кого-либо какое-то право отнято, в соответствующей позиции мы увидим символ дефиса. Так, атрибуты нового текстового файла по описанной схеме будут выглядеть как

```
-rw-r--r--
```

Если файлу приписан атрибут суидности, в тройке владельца место символа `x` займет символ `s`, а при наличии атрибута SGID то же `s` окажется на месте `x` в тройке группы. Атрибут "липкости" маркируется символом `t` в последней позиции (вместо `x` для "прочих"). То есть вывод команды

```
ls -l /usr/bin/passwd
```

будет выглядеть так:

```
-rwsr-xr-x
```

а прочие варианты предлагается домыслить самостоятельно.

При описании прав доступа использована символьная нотация, простая и мнемонически понятная (`r` — от read, `o` — от other и т. д.). Однако наряду с ней существует (и активно используется) нотация цифровая, где права доступа обозначаются числами типа `644`. Поначалу она кажется загадочной. Однако при ближайшем рассмотрении — ничего подобного. Первая цифра соответствует правам хозяина файла, вторая — правам членов группы, третья — правам разных там прочих, как и при символьной нотации. А сама цифра представляет собой простую арифметическую сумму прав, также выраженных численно, только в двоичной системе счисления, трансформированной в восьмеричную для компактности. Наличию любого права соответствует двоичная единица, отсутствию — двоичный ноль. То есть символьной форме

```
rwxrwxrwx
```

будет соответствовать двоичная

```
111 111 111
```

что в восьмеричном пересчете и даст число `777` (полный доступ для всех атрибутов принадлежности). Образуется оно из суммы прав чтения (восьмеричное 4), изменения (восьмеричное 2) и исполнения (восьмеричная 1) в трех позициях. Из чего можно догадаться, что число `000`, напротив, означает отсутствие прав на любые действия у кого бы то ни было.

Арифметические вычисления того, какие числа соответствуют каким соотношениям прав, я предоставляю читателю. Скажу только, что атрибутика новорожденного файла

```
rw-r--r-
```

в численной нотации будет выглядеть как

```
644
```

В численной нотации определяются обычно и права, даруемые файлу при рождении. Для этого существует команда `umask`, и ее аргумент в численной форме показывает, какие права из их суммы должны быть отняты (на этот раз именно отняты) от совокупности исходных прав у файла. То есть для получения стандартных (в большинстве систем) прав `644` этот аргумент должен быть равен `022`.

Догадались, почему в первой позиции стандартной совокупности прав фигурирует `6`, хотя в аргументе команды `umask` видим `0`? Правильно, потому, что ни один создаваемый непосредственно пользователем файл не рождается как исполняемый: этот атрибут должен быть присвоен ему принудительно. А от рождения право на исполнение обретают только двоичные файлы, создаваемые компилятором (например, gcc).

О времени и о файле

В заключение несколько слов о временных атрибутах файла, которые, как станет ясным из главы про управление пакетами, в некоторых случаях оказываются очень важными. Их опять же три:

- время доступа (`atime` — от Access Time);
- время модификации (`mtime` — от Modification Time);
- время изменения (`ctime` — от Change Time).

Первый атрибут фиксирует время последнего обращения к файлу — любого, например, просмотра его командой `less`. Атрибут этот на практике используется редко, а пересчет времени доступа для изобилия файлов на древе UNIX-системы отъедает ресурсы. И потому часто он (пересчет, конечно, а не атрибут) отменяется, о чем будет разговор в *главе 12*.

Атрибут модификации `mtime` устанавливает время последнего изменения файла, вернее, изменения области его данных (например, правки текста в редакторе). А атрибут изменения `ctime` фиксирует время изменения метаданных файла, например, прав доступа, владельца, группы и т. д.

Характерно, что ни один из временных атрибутов файла не отражает непосредственно времени его создания. В качестве такового можно рассматривать атрибут `ctime`, но только в том случае, если в дальнейшем статус файла не изменялся (например, если файлу текста скрипта не был присвоен атрибут исполнения). В противном случае определить истинное время создания файла невозможно.

Интермедия: управление файлами

- Создание.
- Атрибуция.
- Навигация по файловой системе.
- Получение информации о файлах.
- Манипулирование файлами.
- Архивация и компрессия.
- Резервное копирование.
- Венец универсализма: утилита find.

Эта интермедия посвящена средствам для операций с файлами как сущностями, вне зависимости от их содержимого. Она требует знакомства не только с предыдущими главами, но и с *главой 12*, посвященной описанию принципов командного интерфейса.

Средства управления файлами в Base POSIX включают в себя множество отдельных утилит для создания файлов различных типов, установки и изменения их атрибутов, копирования, перемещения, переименования и удаления файлов, а также получения информации о файлах. Кроме того, к сфере файловых операций следует отнести и средства их архивирования, компрессии, да и вообще резервного копирования. Наконец, к этому же кругу относится утилита find — практически универсальное средство не только для поиска файлов, но и для массовой их обработки.

Далее я рассмотрю основные команды, предназначенные для файловых операций, вместе с их наиболее используемыми опциями. Эти команды имеются в любом дистрибутиве Linux (где они объединены в несколько отдельных пакетов, таких как coreutils, findutils и т. д.) и в любой BSD-системе, хотя конкретные их реализации могут несколько различаться. Здесь будет говориться главным образом о самостоятельных командах и утилитах, которым соответствует собственный исполняемый файл (обычно в каталогах /bin, /sbin, /usr/bin или /usr/sbin). Случаи применения встроенных команд оболочки (а они подчас совпадают с внешними утилитами по назначению и имени) будут оговариваться специально.

Чтобы не повторяться, напомню, что почти все описанные команды имеют три стандартные опции (GNU Standard Options):

- `--help` (иногда также `-h` или `-?`) для получения помощи;
- `--verbose` (иногда также `-v`) для вывода информации о текущей версии;
- `--` символизирует окончание перечня опций (любой символ или их последовательность после нее интерпретируются как аргумент).

Так что далее эти опции в описаниях команд фигурировать не будут.

Создание

Работа с файлами начинается с их создания. Конечно, в большинстве случаев файлы создаются соответствующими приложениями (текстовыми редакторами, word-процессорами и т. д.). Однако есть несколько команд, специально предназначенных для создания файлов:

- `touch`;
- `mkdir`;
- `ln`;
- `mknod`;
- `mkfifo`.

Кроме того, с этой же целью могут быть использованы команды `cat` и `tee`.

Команды *touch*, *cat*, *tee*

Первая из указанных команд в форме

```
$ touch filename
```

просто создает обычный (регулярный) файл с именем filename и без всякого содержимого. Кроме того, с помощью специальных опций она позволяет устанавливать временные атрибуты файла, о чем я скажу чуть позже.

Для чего может потребоваться пустой файл? Например, для создания скелета Web-сайта с целью проверки целостности ссылок. Поскольку число аргументов команды `touch` не ограничено ничем (вернее, ограничено только максимальным количеством символов в командной строке), это можно сделать одной командой:

```
$ touch index.html about.html content.html [...]
```

Можно, воспользовавшись приемом группировки аргументов (об этом говорится в *главе 12*), заполнить файлами все подкаталоги текущего каталога:

```
$ touch dirname1/{filename1,filename2} \
        dirname2/{filename3,filename4}
```

и т. д. Правда, сама команда `touch` создавать подкаталоги не способна, это следует сделать предварительно командой `mkdir`.

Для создания пустого регулярного файла может быть также команда `cat` (хотя основное ее назначение — слияние нескольких файлов). Для этого нужно просто перенаправить ее вывод в файл:

```
$ cat > filename
```

создать новую строку (нажатием клавиши <Enter>) и ввести символ конца файла (комбинацией клавиш <Ctrl>+<Z>). Разумеется, предварительно в этот файл можно и ввести какой-нибудь текст.

Интересно создание файлов с помощью команды `tee`. Смысл ее в раздвоении выходного потока, выводимого одновременно и на стандартный вывод, и в файл, указанный в качестве ее аргумента. Если использовать ее для создания файла с клавиатуры, это выглядит так, будто строки удваиваются на экране. Но это не соответствует действительности: просто весь вводимый текст копируется одновременно и на экран, и в файл. И потому ее удобно применять в командных конструкциях, когда требуется одновременно и просмотреть результаты исполнения какой-либо команды, и запечатлеть их в файле:

```
$ ls dir | tee filename
```

По умолчанию команда `tee` создает новый файл с указанным именем или перезаписывает одноименный, если он существовал ранее. Однако с опцией `-a` она добавляет новые данные в конец существующего файла.

Команда *mkdir*

Команда `mkdir` создает файл особого типа — каталог, содержимым которого является список входящих в него файлов. Очевидно, что список этот в момент создания каталога должен быть пуст, однако это не совсем так: любой, даже пустой, каталог содержит две ссылки: на каталог текущий, обозначаемый как ./ (сам на себя), и на каталог родительский ../ (тот, в список файлов которого он включается в момент создания).

Команда `mkdir` требует обязательного аргумента — имени создаваемого каталога. Аргументов может быть больше одного, в этом случае будет создано два или больше поименованных каталогов. По умолчанию они создаются как подкаталоги каталога текущего. Можно создать также подкаталог в существующем подкаталоге:

```
$ mkdir parentdir/newdir.
```

Если же требуется создать подкаталог в каталоге, отличном от текущего, путь к нему требуется указать в явном виде, в относительной форме

```
$ mkdir ../dirname1/dirname2
```

или в форме абсолютной

```
$ mkdir /home/username/dirname1/dirname2
```

В отличном от текущего каталоге можно одной командой создать несколько подкаталогов, для чего нужно прибегнуть к группировке аргументов:

```
$ mkdir ../parentdir/{dirname1,dirname2,...,dirname#}
```

Такой прием позволяет одной командой создать дерево каталогов проекта. Например, скелет Web-сайта, который потом можно наполнить пустыми файлами с помощью команды `touch`.

А опций у команды `mkdir` всего две (за исключением стандартных опций GNU):

- `--mode` (или `-m`) для установки атрибутов доступа;
- `--parents` (или `-p`) для создания как требуемого каталога, так и родительского по отношению к нему (если таковой ранее не существовал).

Первая опция используется в форме

```
$ mkdir --mode=### dirname
```

или

```
$ mkdir -m ### dirname
```

Здесь под `###` понимаются атрибуты доступа для владельца файла, группы и прочих, заданные в численной нотации (например, `777` — полный доступ на чтение, изменение и исполнение для всех). Не возбраняется и использование символьной нотации: команда

```
$ mkdir -m a+rwx dirname
```

создаст каталог с теми же атрибутами полного доступа для всех.

Опция `--parents` (она же `-p`) позволяет создавать иерархическую цепочку подкаталогов любого уровня вложенности. Например,

```
$ mkdir -p dirlevel1/dirlevel2/dirlevel3
```

в один заход создаст в текущем каталоге цепочку вложенных друг друга подкаталогов. Разумеется, и здесь с помощью группировки аргументов можно создать несколько одноранговых подкаталогов:

```
$ mkdir -p dirlevel1/dirlevel2/{dirlevel31,...,dirlevel3#}
```

Команда ln

Командой `ln` создаются ссылки обоих видов — жесткие и символические. Первые — просто иное имя (то есть иная запись в каком-либо каталоге) для того же набора данных. И создается столь же просто

```
$ ln filename linkname,
```

где первый аргумент (`filename`) — имя существующего файла, а второй (`linkname`) — имя создаваемой ссылки. Как известно, жесткая ссылка может быть создана только на обычный (регулярный) или специальный файл (например, файл устройства), но не на каталог. Кроме того, она не может пересекать границ физической файловой системы.

Символическая ссылка создается той же командой `ln` и с теми же аргументами, но со специальной опцией `-s`:

```
$ ln -s filename linkname
```

Символическая ссылка — файл особого типа и таких ограничений не имеет: она может указывать на любой файл или каталог, в том числе расположенный в другой файловой системе (и даже на другой машине). При создании символической ссылки на каталог автоматически создаются символические ссылки на все входящие в его состав файлы и вложенные подкаталоги. Изменение содержимого исходного каталога (пополнение его файлами или их удаление) столь же автоматически влечет за собой изменение каталога-ссылки.

В качестве источника символической ссылки может выступать другая символическая ссылка. То есть команда

```
$ ln -s linkname linkname2
```

создаст символическую ссылку `linkname2`, ссылающуюся на ссылку же `linkname`, и только последняя будет указывать на обычный файл или каталог `filename`. Однако если дать команду

```
$ ln -n linkname linkname2
```

то новообразованная ссылка `linkname2` будет указывать не на ссылку `linkname`, а на исходный для последней файл.

Если имя второго аргумента команды `ls -s` совпадает с именем существующего файла (регулярного, каталога, или символической ссылки на файл, отличный от первого аргумента команды), то новая символическая ссылка создана не будет. Однако такую ссылку, совпадающую с существующим именем, можно принудительно создать посредством опции `-f` (или `--force`).

При этом, разумеется, содержание оригинального файла (будь то другая символическая ссылка, регулярный файл или каталог), одноименного создаваемой символической ссылке, будет утеряно безвозвратно, в случае каталога — вместе со всеми его файлами и подкаталогами. Поэтому при форсированном создании символических ссылок на каталоги, содержащие большое количество разноименных файлов, существует вероятность случайной (при простом совпадении имен) утери важных данных. Чтобы предотвратить это, используется опция `--interactive` (`-i`): благодаря ей команда

`ln` будет выдавать запрос на подтверждение действий, если обнаружит совпадения имен создаваемых ссылок и существующих файлов.

Есть и другие способы сохранить исходный файл при совпадении его имени с именем создаваемой ссылки. Так, опция `-b` (`--backup`), прежде чем переписать замещаемый файл, создаст его резервную копию вида `filename~`. А опция `-S` (`--suffix`) не только создаст такую копию, но и припишет к ее имени какой-либо осмысленный суффикс. Так, в результате команды

```
$ ln -sf --S=.old filename1 filename2
```

существовавший ранее регулярный файл `filename2` будет замещен символической ссылкой на файл `filename1`, однако содержимое оригинала будет сохранено в файле `filename2.old`. Опция же `-V METHOD` (`--version-control=METHOD`) позволяет последовательно нумеровать такие копии замещаемых симлинками файлов. Значения `METHOD` могут быть:

- `t`, `numbered` — всегда создавать нумерованную копию;
- `nil`, `existing` — создавать нумерованную копию только в том случае, если хоть одна таковая уже существует, если же нет — создавать обычную копию;
- `never`, `simple` — всегда создавать обычную копию;
- `none`, `off` — не создавать копию замещаемого файла вообще.

Нумерация копий с помощью команды `ln` позволяет создавать своего рода систему контроля версий проекта, когда в каждый момент времени символическая ссылка указывает на каталог текущей версии, а все предыдущие сохраняются в самостоятельных каталогах.

Команда *mknod*

Команда `mknod` предназначается для создания файлов специального типа — файлов устройств. Она требует в качестве аргументов указать:

- имя устройства в форме `/dev/dev_name`;
- тип:
  - символьный — `c` (например, виртуальные терминалы);
  - блочный — `b` (например, дисковые накопители и их разделы);
- старший номер (major) — уникальный идентификатор, характеризующий родовую группу устройств (например, `4` — идентификатор виртуальных терминалов);
- младший номер (minor), являющийся идентификатором конкретного устройства в своей группе (например, младший номер `0` в группе старшего номера `4` — идентификатор первой, системной, виртуальной консоли, а `63` — идентификатор последней теоретически возможной из них).

Кроме стандартных, команда `mknod` имеет только одну опцию `-m` (`--mode`), с помощью которой устанавливаются атрибуты доступа к создаваемому файлу устройства (точно также, как это было описано для команды `mkdir`). Таким образом, команда

```
$ mknod --mode=200 /dev/tty63 c 4 63
```

создаст файл устройства для последней теоретически возможной виртуальной консоли. Что, впрочем, не значит, будто бы сразу после этого в нее можно переключаться, предварительно следует внести должные изменения в скрипты инициализации.

На практике команда `mknod` часто используется в опосредованном виде, в составе сценария /dev/MAKEDEV, автоматизирующего процесс создания файлов устройств. В частности, он делает ненужным знание старшего и младшего номеров устройств. Правда, только для тех из них, которые были предусмотрены при разработке сценария. Если же требуемое устройство не входит в число охваченных данным вариантом /dev/MAKEDEV, использования команды `mknod` в явном виде не избежать. Правда, все более широкое внедрение файловой системы devfs или (пока только в Linux) механизма udev, позволяющих создавать файлы устройств при загрузке или "на лету", при "горячем" подключении, скоро сделают команду `mknod` анахронизмом.

Команда *mkfifo*

Если создавать файлы устройств приходится достаточно редко, то необходимости применения команды `mkfifo` у пользователя может не возникнуть никогда (у меня, например, такой необходимости не возникало ни разу за все время знакомства с POSIX-системами). Тем не менее, отметим для полноты картины, что такая команда существует и создает именованные каналы (named pipes) — специальные файлы, предназначенные для обмена данными между процессами. Вообще-то, именованные каналы играют большую роль во всех UNIX-системах, но роль эта от пользователя хорошо замаскирована. Так что просто приведу формат команды без комментариев:

```
$ mkfifo [options] filename
```

А опция здесь всего одна `-m` (`--mode`), и как нетрудно догадаться по аналогии с командой `mknod`, устанавливает она атрибуты доступа.

Атрибуция

К слову сказать об атрибутах, следующая группа команд предназначена именно для атрибуции файлов. В ней — `chmod`, `chown`, `chgrp`, `umask`, а также уже затронутая ранее команда `touch`.

Команды *chown* и *chgrp*

Команды `chown` и `chgrp` служат для изменения атрибутов принадлежности файла — хозяину и группе: очевидно, что все, не являющиеся хозяином файла, и не входящие в группу, к которой файл приписан, автоматически попадают в категорию прочих (`other`).

Формат команды `chown` — следующий:

```
$ chown newowner filename
```

По соображениям безопасности изменить хозяина файла может только суперпользователь. Пользователь обычный в подавляющем большинстве случаев автоматически становится хозяином всех им созданных (и скопированных) файлов, и избавиться от этого бремени, как и от родительского долга, не в состоянии.

А вот изменить групповую принадлежность своих файлов (в атрибутах принадлежности он прописан как хозяин) пользователь вполне может. Команда:

```
$ chgrp newgroup filename
```

от его лица припишет файл `filename` к группе `newgroup`. Однако и здесь есть ограничение: результат будет достигнут, только если хозяин файла является членом группы `newgroup`, иначе опять придется прибегнуть к полномочиям администратора.

Можно также одной командой сменить (только суперпользователю, конечно) и хозяина файла, и группу, к которой он приписан. Делается это так:

```
$ chown newowner:newgroup filename
```

или так:

```
$ chown newowner.newgroup filename
```

Где, понятное дело, под именем `newowner` выступает новый хозяин файла, а под именем `newgroup` — новая группа, к которой он приписан.

В обеих командах вместо имени хозяина и группы могут фигурировать их численные идентификаторы (UID и GID соответственно). Это имеет смысл, например, при совместном использовании файлов в разных операционных системах. Так, даже единственный пользователь в каком-либо варианте Linux и в BSD по умолчанию имеет разные идентификаторы, и чтобы сделать его владельцем неких файлов и там, и там, именно численный идентификатор должен фигурировать в качестве параметра команды `chown`.

Для команд `chown` и `chgrp` поддерживается один и тот же набор опций. Наиболее интересны (и важны) две из них:

- опция `--reference` позволяет определить хозяина файла и его принадлежность к группе не явным образом, а по образу и подобию файла, имя которого выступает в качестве значения опции.

Так, команда

```
$ chown --reference=ref_filename filename
```

установит для файла `filename` те же атрибуты принадлежности (хозяина и группу), что были ранее у файла `ref_filename`. Это весьма полезно при массовой реатрибуции файлов, полученных из разных источников;

- опция `-R` (или `--recursive`) распространяет действие обеих команд не только на файлы текущего каталога, но и на все вложенные подкаталоги, вместе с входящими в них файлами. То есть пользователь может поменять групповую принадлежность всех файлов в своем домашнем каталоге одной командой:

  ```
  $ chgrp -R newgroup ~/*
  ```

  А суперпользователь тем же способом может установить единообразные атрибуты принадлежности "по образцу" для всех компонентов любого каталога:

  ```
  $ chown -R --reference=ref_filename \
      /somepath/somecat/*
  ```

Команды *chmod* и *umask*

Как и следует из ее имени, команда `chmod` предназначена для смены атрибутов доступа: чтения, изменения и исполнения. В отношении единичного файла делается это просто:

```
$ chmod [атрибуты] filename
```

Атрибуты доступа могу устанавливаться с использованием как символьной, так и цифровой нотации. Первый способ — указание, для каких атрибутов принадлежности (хозяина, группы и всех остальных) какие атрибуты доступа задействованы. Атрибуты принадлежности обозначаются символами `u` (от user) для хозяина файла, `g` (от group) — для группы, `o` (от other) для прочих и `a` (от all) — для всех категорий принадлежности вообще. Атрибуты доступа символизируются литерами `r` (от read), дающей право чтения, `w` (от write) — право изменения и `x` (от execute) — право исполнения.

Атрибуты принадлежности соединяются с атрибутами доступа:

- `+` (присвоение атрибута доступа);
- `-` (отнятие атрибута);
- `=` (присвоение только данного атрибута доступа с одновременным отнятием всех остальных).

Одновременно в строке можно указать (подряд, без пробелов) более чем один из атрибутов принадлежности и несколько (или все) атрибуты доступа.

Для пояснения сказанного приведу несколько примеров. Так, команда

```
$ chmod u+w filename
```

установит для хозяина (`u`) право изменения (`+w`) файла `filename`, а команда

```
$ chmod a-x filename
```

отнимет у всех пользователей вообще (`a`) право его исполнения (`-x`). В случае, если некоторый атрибут доступа присваивается всем категориям принадлежности, символ `a` можно опустить. Так, команда

```
$ chmod +x filename
```

присвоит атрибут исполнения файла `filename` всем категориям принадлежности (и хозяину, и группе, и прочим).

С помощью команды

```
$ chmod go=rx filename
```

можно присвоить группе принадлежности файла `filename` и всем прочим (не хозяину и не группе) право на его чтение и исполнение с одновременным отнятием права изменения.

Наконец, команда `chmod` в состоянии установить и дополнительные атрибуты режима для файлов, такие как биты SUID и GUID, или, скажем, атрибут sticky. Так, в некоторых системах (например, во FreeBSD — в Linux-дистрибутивах я с таким не встречался) XFree86 подчас по умолчанию устанавливается без атрибута суидности на исполнимом файле X-сервера. Это влечет за собой невозможность запуска Иксов от лица обычного пользователя. Один из способов борьбы с этим — просто присвоить этому файлу бит суидности:

```
$ chmod u+s /usr/X11R6/bin/XFree86
```

Это, правда, не рекомендуется из соображений безопасности: штатным средством для преодоления этой коллизии является использование специальной программы — `xwrapper`.

Приведенные примеры можно многократно умножить, но, думается, их достаточно для понимания принципов работы команды `chmod` с символьной нотацией атрибутов.

Цифровая нотация еще проще. При ней достаточно указать сумму присваиваемых атрибутов в восьмеричном исчислении (`4` — атрибут чтения, `2` — атрибут изменения и `1` — атрибут исполнения; `0` символизирует отсутствие любых атрибутов доступа) для хозяина (первая позиция), группы (вторая позиция) и прочих (третья позиция). Все атрибуты доступа, оставшиеся вне этой суммы, автоматически отнимаются у данного файла. То есть команда

```
$ chmod 000 filename
```

означает снятие с файла `filename` всех атрибутов доступа для всех категорий принадлежности (в том числе и хозяина) и эквивалентна команде

```
$ chmod =rwx filename
```

в символьной нотации. А команда

```
$ chmod 777 filename
```

напротив, устанавливает для всех полный доступ к файлу `filename`. Для установки дополнительных атрибутов доступа в численной нотации потребуется указать значение четвертого, старшего, регистра. Так, команда для рассмотренного примера присвоения атрибута суидности исполнимому файлу X-сервера в численной нотации будет выглядеть как

```
$ chmod 4711 /usr/X11R6/bin/XFree86
```

Как и для команд `chown` и `chgrp`, наиболее значимые опции `chmod`:

- `--reference`;
- `-R`.

И смысл их тот же самый. Первая устанавливает для файла (файлов) атрибуты доступа «по образцу», вторая распространяет действие команды на все вложенные подкаталоги и входящие в них файлы.

Рекурсивное присвоение атрибутов доступа «по образцу» требует внимания. Так, если рекурсивно отнять для всего содержимого домашнего каталога атрибут исполнения (а он без соблюдения некоторых условий монтирования автоматом присваивается любым файлам, скопированным с носителей файловой структуры FAT или ISO9660 без расширения RockRidge, что подчас мешает), то тем самым станет невозможным вход в любой из вложенных подкаталогов. Впрочем, в параграфе про утилиту `find` будет показан один из способов борьбы с таким безобразием.

Для всех вновь создаваемых данным пользователем файлов можно установить атрибуты доступа по умолчанию. Этой цели служит команда `umask`: в отличие от прочих, не самостоятельная утилита, а встроенная команда оболочки. Данная без аргумента, она выведет текущее значение субтрактивных (то есть отнимаемых от суммы) прав доступа для новообразуемых файлов:

```
$ umask
22
```

Вывод прав дается в символьной нотации, нули игнорируются. Если же в качестве аргумента указать "отнимаемые" права, все вновь создаваемые файлы будут иметь новые атрибуты доступа. Например, команда

```
$ umask 000
```

приведет к тому, что новые файлы будут иметь всю совокупность атрибутов доступа (чтения, изменения и исполнения) для хозяина, группы и прочих.

Действие команды `umask` таким образом распространяется только на текущий сеанс работы. Поэтому обычно она включается в профильный файл пользовательской командной оболочки, определяя права доступа по умолчанию на вечные времена.

Команда *touch* для атрибуции

Кроме атрибутов принадлежности и доступа, файлам свойственны еще и атрибуты времени — времени доступа (`atime`), времени изменения метаданных (`ctime`) и времени изменения данных (`mtime`) файла. Они устанавливаются автоматически, в силу самого факта открытия файла (`atime`), смены любых атрибутов, например, доступа (`ctime`) или редактирования содержимого файла (`mtime`).

Однако бывают ситуации, когда автоматически установленные временные атрибуты требуется изменить. Случай продления жизни trial-версии программы не рассматриваем: настоящий POSIX'ивист до такого не опускается, не так ли? А вот сбой системных часов, в результате которого временные атрибуты создаваемых и модифицируемых файлов перестанут соответствовать действительности, — штука вполне вероятная.

Казалось бы, чего страшного? А нет, фактор времени играет в UNIX-системах очень существенную роль. Во-первых, команда `make` (а под ее управлением компилируются программы из исходников) проверяет временные атрибуты файлов (в первую очередь, атрибут `mtime`) и при их несоответствии может работать с ошибками. Ну и более обычная ситуация: на основе временных меток файлов можно эффективно осуществлять, скажем, резервное копирование. И потому желательно, чтобы они отражали реальное время создания и модификации файла.

Так вот, для изменения временных атрибутов файлов и предназначена в первую очередь команда `touch`, которую ранее мы использовали просто для создания пустого файла. С именем существующего файла в качестве аргумента

```
$ touch exist_file
```

она присвоит всем его временным атрибутам (`atime`, `ctime`, `mtime`) значения текущего момента времени. Изменение временных атрибутов можно варьировать с помощью опций. Так, если указать только одну из опций `-a`, `-c`, или `-m`, то текущее значение времени будет присвоено только атрибуту `atime`, `ctime` или `mtime` соответственно. Если при этом использовать еще и опцию `-d` со значением, то любому из указанных атрибутов (или им всем) можно присвоить любую временную метку, в том числе и из далекого будущего. А посредством опции `-r filename` файл-аргумент получит временные атрибуты.

Навигация по файловой системе

Для начала неплохо определиться со своим текущим положением в файловой системе. Этому послужит команда `pwd`. В ответ на нее выводится полный путь к текущему каталогу. Например, если текущим является домашний каталог пользователя, в ответ на

```
$ pwd
```

последует

```
/home/username.
```

Команда `pwd` имеет всего две опции: `-L` и `-P`. Первая выводит логический путь к текущему каталогу. Таковым является, скажем, каталог /usr/src/linux, являющий собой символическую ссылку на каталог /usr/src/linux-номер_версии, то в ответ на

```
$ pwd -L
```

так и будет выведено

```
/usr/src/linux.
```

Впрочем, тот же ответ последует и на команду `pwd` без опций вообще. Если же дать эту команду в форме

```
$ pwd -P
```

то будет выведен путь к физическому каталогу, на который ссылается текущий, например:

```
/usr/src/linux-2.4.19-gentoo-r9
```

Далее, по каталогам неплохо как-то перемещаться. Это делается командой `cd`. В отличие от большинства прочих команд, рассматриваемых в этом разделе, это внутренняя команда, встроенная во все командные оболочки, бесполезно было бы искать соответствующий ей исполняемый файл. Однако это не уменьшает ее важности. Использование ее очень просто:

```
$ cd pathname
```

где `pathname` — путь к искомому каталогу в абсолютной (относительно корня) или относительной (относительно текущего каталога) форме.

Определить местоположение команды (и вообще исполняемых файлов) в структуре файловой системы можно с помощью команды `which` (это также встроенная команда оболочки). В качестве аргумента ее можно указать одно или несколько имен файлов, в ответ на что будет выведен полный путь к каждому из них:

```
$ which tcsh zsh bash
/bin/tcsh
/bin/zsh
/bin/bash
```

При наличии одноименных исполняемых файлов в разных каталогах по умолчанию будет выведен путь только к первому из них: для вывода всех файлов-"тезок" можно прибегнуть к опции `-a`. При этом не важно, будут это жесткие или символические ссылки.

Более широкие возможности поиска у команды `whereis`. По умолчанию без опций она для заданного в качестве аргумента имени выводит список бинарных файлов, man-страниц и каталогов с исходными текстами:

```
$ whereis zsh
zsh: /bin/zsh /etc/zsh /usr/lib/zsh /usr/share/zsh \
/usr/man/man1/zsh.1.gz /usr/share/man/man1/zsh.1.gz
```

Соответствующими опциями можно задать поиск файлов одного из этих типов:

- `-b` — бинарных;
- `-m` — страниц руководств;
- `-s` — каталогов с исходниками.

Дополнительные опции `-B`, `-M`, `-S` (в сочетании с опцией `-f`) позволяют определить исходные каталоги для их поиска.

Наконец, команда `locate` осуществляет поиск всех файлов и каталогов, содержащих компонент имени, указанный в качестве аргумента и осуществляет вывод содержимого найденных каталогов. Так, в ответ на команду

```
$ locate zsh
```

будет выведен список вроде следующего:

```
/bin/zsh
/bin/zsh-4.0.6
/etc/zsh
/etc/zsh/zlogin
/etc/zsh/zshenv
/etc/zsh/zshrc
```

и т. д. Команда `locate` при этом обращается к базе данных, расположенной в каталоге вроде /var/spool/locate/locatedb или /var/db/locate.database. По умолчанию эта база данных пуста и перед использованием команды locate должна быть наполнена содержанием. Для этого предназначен сценарий /usr/bin/updatedb или, в иных системах, /usr/libexec/locate.updatedb (обращаем внимание на полный путь во втором случае, поскольку каталоги типа /usr/libexec/ в переменной `PATH` обычно не указывается, именно таким способом и следует его запускать). Сценарий этот извлекает сведения из базы данных установленных пакетов, например, в BSD-системах, /var/db/pkg. База данных для команды `locate` нуждается в периодическом обновлении (за

обновление базы данных установленных пакетов обычно отвечает система пакетного менеджмента данного дистрибутива или ОС).

Приведенные команды относятся к поиску исполняемых файлов и программных компонентов. Однако чаще перед пользователем возникает необходимость поиска неких произвольных файлов. На сей предмет существует команда `find`. Однако возможности ее не сводятся к поиску, это практически универсальный инструмент для файловых операций. И потому она будет подробно рассмотрена отдельно, в последнем разделе этой главы.

Получение информации о файлах

Наиболее универсальным средством получения практически исчерпывающей информации о файлах является команда `ls`. Общая форма ее запуска

```
$ ls [options] names
```

где в качестве аргумента `names` могут выступать имена файлов или каталогов в любом количестве. Команда эта имеет многочисленные опции, основные из которых мы и рассмотрим.

Начать с того, что команда `ls`, данная без всяких опций, по умолчанию выводит только имена файлов, причем опуская dot-файлы, имена которых начинаются с точки (это некие аналоги скрытых файлов в MS-DOS и Windows). Кроме того, если в качестве аргумента указано имя каталога (или аргумент не указан вообще, что подразумевает текущий каталог), из списка имен его файлов не выводятся текущий (.) и родительский (..) каталоги.

Для вывода всех без исключения имен файлов (в том числе и скрытых) предназначена опция `-a`. Смысл опции `-A` близок, она выводит список имен всех файлов, за исключением символов текущего (.) и родительского (..) каталога.

Кроме имени, любой файл идентифицируется своим номером `inode`. Для его вывода используется опция `-i`:

```
$ ls -i
12144 content.html
```

и т. д. Как и многие другие, команда `ls` обладает способностью рекурсивной обработки аргументов, для чего предназначена опция `-R`, выводящая список имен файлов не только текущего каталога, но и всех вложенных подкаталогов:

```
$ ls -R
unixforall:
about/  apps/     diffimages/  distro/  signature.html  sys/
anons/  content/  difftext/    gentoo/  statistics/     u4articles/
```

```
unixforall/about:
about_lol.html  about_lol.txt  index.html

unixforall/anons:
anons_dc.html
```

Опция же `-d`, напротив, запрещает вывод содержимого вложенных подкаталогов.

В выводе команды `ls` по умолчанию имена файлов разных типов даются абсолютно одинаково. Для их визуального различия используется опция `-F`, завершающая имена каталогов символом `/`, исполнимых файлов — `*`, символических ссылок — `@`; имена регулярных файлов, не имеющих атрибута исполнения, никакого символа не включают.

```
$ ls -F
dir1/   dir2/   dir3@   file1   file2*  file3@
```

Другое средство для визуального различия типов файлов — колоризация, для чего применяется опция `-G`. Цвет шрифта, воспроизводящего имена, по умолчанию синий для каталогов, лиловый (magenta) для символических ссылок, красный — исполнимых файлов. Для файлов устройств, исполнимых файлов с атрибутом суидности, каталогов, имеющих атрибут `sticky`, дополнительно колоризуется и фон, на котором выводится шрифт, воспроизводящий их имена. Подробности можно посмотреть в секции ENVIRONMENT man-страницы для команды `ls`.

По умолчанию команда `ls` выводит список файлов в порядке ASCII-кодов первого символа имени. Однако есть возможность его сортировки в порядке времени модификации (`-t`), изменения статуса (`-c`) или времени доступа (`-tu`), а также в порядке, обратном любому из перечисленных (`-r`). Кроме того, опция `-f` отменяет какую-либо сортировку списка вообще.

Информацию об объеме файлов можно получить, используя опцию `-s`, выводящую для имени каждого файла его размер в блоках, а также суммарный объем всех выведенных файлов.

```
$ ls -s ../book
total 822
656 book.html
4 content1.html
86 var_part2.html
24 command.html
38 part2.html
6 command.txt
8 shell_tmp.html
```

Добавление к опции `-s` еще и опции `-k` (то есть `ls -sk`) выведет всю ту же информацию в килобайтах.

Как можно видеть из всех приведенных примеров, списки файлов по команде `ls` выводятся в многоколоночном виде (чему соответствует опция `-C`, однако указывать ее нет необходимости — многоколоночный вид принят для краткого формата по умолчанию). Но посредством опции `-1` можно задать представление списка в одну колонку:

```
$ ls -1
dir1
dir2
dir3
file1
file2
file3
```

До сих пор речь шла о кратком формате вывода команды `ls`. Однако более информативным является длинный ее формат (с опцией `-l`):

```
$ ls -l
total 8
drwxr-xr-x  2 alv  alv  512  8 май 18:04 dir1
drwxr-xr-x  3 alv  alv  512  8 май 17:43 dir2
lrwxr-xr-x  1 alv  alv    4  9 май 07:59 dir3 -> dir2
-rw-r--r--  1 alv  alv   14  8 май 10:39 file1
-rwxr-xr-x  1 alv  alv   30  9 май 08:02 file2
lrwxr-xr-x  1 alv  alv    2  8 май 10:57 file3 -> f1
```

Можно видеть, что по умолчанию в длинном формате выводятся:

- сведения о типе файла:
  - `-` — регулярный файл;
  - `d` — каталог;
  - `l` — символическая ссылка;
  - `c` — файл символьного устройства;
  - `b` — файл блочного устройства;
- количество жестких ссылок на данный идентификатор inode;
- имя владельца файла и группы пользователей, которой файл принадлежит;
- размер файла в блоках;
- время модификации файла с точностью до месяца, дня, часа и минуты;
- имя файла и (для символических ссылок) имя файла-источника.

Однако это еще не все. Добавив к команде `ls -l` еще и опцию `-i`, можно дополнительно получить идентификатор inode каждого файла, опция `-n` заменит имя владельца и группу на их численные идентификаторы (UID и GUID, соответственно), а опция `-T` выведет в поле времени модификации еще и годы, и секунды:

```
$ ls -linT
total 8
694402 drwxr-xr-x  2 1000  1000  512  8 май 18:04:56 2002 dir1
694404 drwxr-xr-x  3 1000  1000  512  8 май 17:43:31 2002 dir2
673058 lrwxr-xr-x  1 1000  1000    4  9 май 07:59:08 2002 dir3 -> dir2
673099 -rw-r--r--  1 1000  1000   14  8 май 10:39:38 2002 file1
673059 -rwxr-xr-x  1 1000  1000   30  9 май 08:02:23 2002 file2
673057 lrwxr-xr-x  1 1000  1000    2  8 май 10:57:07 2002 file3 -> f1
```

Разумеется, никто не запрещает использовать в длинном формате и опции визуализации (`-F` и `-G`), и опции сортировки (`-r`, `t`, `tu`), и любые другие, за исключением опции `-C` — указание ее ведет к принудительному выводу списка в многоколоночной форме, что естественным образом подавляет длинный формат представления.

Я столь подробно остановился на описании команды `ls` потому, что это основное средство визуализации файловых систем любого UNIX, при умелом использовании ничуть не уступающее развитым файловым менеджерам (типа Midnight Commander или Konqueror) по своей выразительности и информативности.

Существуют и другие команды для получения информации о файлах. Например, команда с характерным именем `file` и аргументом в виде имени файла в состоянии определить тип его, а также характер содержания с большой детальностью. Так, для регулярных файлов она распознает:

- исполняемые бинарные файлы с указанием их формата (например, ELF), архитектуры процессора, для которых они скомпилированы, характер связи с разделяемыми библиотеками (статический или динамический);
- исполняемые сценарии с указанием оболочки, для которой они созданы;
- текстовые и HTML-документы, часто с указанием используемого набора символов.

Последнему, впрочем, для русскоязычных документов доверять особо не следует: кодировка KOI8-R в них вполне может быть обозвана ISO-8859.

Определяет она также каталоги, символические ссылки, специальные файлы устройств, указывая для последних старшие и младшие номера устройств.

Наконец, команда `stat` (и это встроенная команда оболочки) с именем файла в качестве аргумента выводит большую часть существенных сведений

о файле в удобном для восприятия виде, например, включая идентификатор `inode`, режим доступа (в символьной форме), идентификаторы владельца и группы, временные атрибуты, количество жестких и символических ссылок.

Приведенных способов получения информации о файле, как кажется, пользователю должно быть достаточно. Перейдем к манипуляциям с существующими файлами: копированию, перемещению, переименованию, удалению.

Манипулирование файлами

Начнем с копирования, оно выполняется очень простой командой, `cp`, имеющей, однако, весьма разнообразные аспекты применения. В самом простом виде она требует всего двух аргументов: имени файла-источника на первом месте и имени целевого файла — на втором.

```
$ cp file_source file_target
```

Этим в текущем каталоге создается новый файл (`file_target`), идентичный копируемому (`file_source`). То есть область данных первого будет дублировать таковую последнего. Однако области метаданных у них будут различны изначально. Целевой файл — это именно новый файл, со своим идентификатором `inode`, заведомо иными временными атрибутами; его атрибуты доступа и принадлежности в общем случае также не обязаны совпадать с таковыми файла-источника.

Новый файл может быть создан и в произвольном каталоге, к которому пользователь имеет соответствующий доступ: для этого следует только указать полный путь к нему.

```
$ cp file_source dir/subdir/file_target
```

Если в качестве второго аргумента команды указано просто имя каталога, то новый файл будет создан в нем с именем, идентичным имени файла-источника. Однако подчеркну, что в любом случае копирования создается именно новый файл, никак после этого не связанный с файлом исходным.

Если в качестве последнего аргумента выступает имя каталога, он может предваряться любым количеством аргументов — имен файлов.

```
$ cp file1 file2 ... file3 dir/
```

В этом случае в целевом каталоге `dir/` будут созданы новые файлы, идентичные по содержанию файлам `file1`, `file2` и т. д.

Если в целевом (или текущем) каталоге уже имеется файл с именем, совпадающим с именем вновь создаваемого файла, он в общем случае будет без предупреждения заменен новым файлом. Единственное средство предот-

вращения этого — задание опции `-i` (от interactive), при ее наличии последует запрос на перезапись существующего файла.

```
$ cp -i file1 file2
overwrite file2? (y/n [n])
```

Удаление большого количества заведомо ненужных файлов можно выполнить с помощью опции `-f` (от force).

Имя каталога может выступать и в качестве первого аргумента команды `cp`. Однако это потребует опции `-R` (иногда допустима и опция `-r` — в обоих случаях от recursive). В этом случае второй аргумент также будет воспринят как имя каталога, в нем также будет рекурсивно воспроизведено содержимое каталога источника (включая и вложенные подкаталоги).

При копировании файлов, представляющих собой символические ссылки, они будут преобразованы в регулярные файлы, копирующие содержимое файлов — источников ссылки. При рекурсивном копировании каталогов, содержащих символические ссылки, возможно их воспроизведение в первозданном виде — как симлинков. Для этого вместе с опцией `-R` должна быть указана одна из опций `-H` или `-L`. Обе они при отсутствии `-R` игнорируются.

Как уже было сказано, создаваемые при копировании целевые файлы по умолчанию получают атрибуты доступа и времени, не зависящие от таковых файла-источника. Обычно они определяются значением аргумента команды `umask`, заданной глобально, в профильном файле командной оболочки пользователя (по умолчанию значение `umask`, как правило — `022`). Однако при желании атрибуты исходного файла можно сохранить в файле целевом, для этого предназначена опция `-p`. Разумеется, атрибуты эти будут сохранены только в том случае, если это допустимо целевой файловой системой: не следует ожидать, что атрибуты доступа и принадлежности будут сохранены при копировании на носитель с файловой системой FAT.

При операции копирования файла он должен иметь атрибут чтения для пользователя, выполняющего копирование; кроме того, последний должен обладать правом на изменение каталога, в который производится копирование.

Кроме простого копирования файлов, существует команда для копирования с преобразованием — `dd`. Обобщенный ее формат весьма прост:

```
$ dd [options]
```

то есть она просто копирует файл стандартного ввода в файл стандартного вывода, а опции описывают условия преобразования входного потока данных в выходной. Реально основными опциями являются `if=file1`, подменяющая стандартный ввод указанным файлом, и `of=file2`, проделывающая ту же операцию со стандартным выводом.

А далее — прочие условия преобразования, весьма обильные. Большинство из них принимают численные значения в блоках:

- опции `ibs=n` и `obs=n` устанавливают размер блока для входного и выходного потоков, `bs=n` — для обоих сразу;
- опция `skip=n` указывает, сколько блоков нужно пропустить перед записью входного потока;
- опция `count=n` предписывает скопировать из входного потока лишь указанное количество блоков, отсчитываемых с начала файла-источника.

Имеется и опция `conv=value`, которая преобразует входной поток в соответствии с принятыми значениями, например, из формата ASCII в формат EBCDIC, рекомендуемый для использования в ОС на базе UNIX System V.

Сфера применения команды `dd` далеко выходит за рамки простого копирования файлов. Например, именно с ее помощью изготавливаются загрузочные дискеты, точные копии диска в файловой системе на винчестере, преобразуются шрифтовые файлы из одного формата в другой, и еще многое другое. В частности, эта же команда может применяться для резервного копирования данных, в том числе и на компакт-диск. Наконец, в BSD-клане существует еще и команда `cpdup`, призванная копировать не просто файлы, а целые файловые системы. В отличие от команды `cp`, с ее помощью создается полное зеркало файловой системы или отдельных ее ветвей, с сохранением жестких и символических ссылок, файлов устройств, временных атрибутов и атрибутов доступа файлов и подкаталогов. Именно команда `cpdup` используется при установке DragonFlyBSD для переноса структуры установочного диска на носитель, выбранный в качестве цели инсталляции. Другое применение команды — резервное копирование.

Следующие две часто используемые файловые операции: переименование и перемещение — выполняются одной командой `mv`. Она требует минимум двух аргументов — имени источника и целевого имени. Если оба они имена файлов, происходит переименование первого файла во второй. Если последним аргументом выступает имя уже существующего каталога, то файл или каталог, указанные в качестве первого аргумента, перемещается в каталог назначения. Причем если первый аргумент — файл, между первым и последним аргументами может быть сколько угодно аргументов — имен файлов (но не каталогов).

Как и при копировании, при перемещении и переименовании одноименные файлы, ранее существовавшие в целевом каталоге, затираются, замещаясь файлами-источниками без предупреждения. Чтобы этого не случилось, используется опция `-i`, требующая запрос на подтверждение действия. Напротив, опция `-f` в принудительном порядке перезаписывает существующий файл.

Операции копирования и перемещения/переименования выглядят сходными, однако по сути своей глубоко различны. Начать с того, что команда `mv` не совершает никаких действий с перемещаемыми или переименовываемыми файлами, она модифицирует каталоги, к которым приписаны имена этих файлов. Это имеет два важных следствия. Во-первых, при перемещении/переименовании файлы сохраняют первозданными атрибуты доступа, принадлежности и даже времени изменения метаданных (`ctime`) и модификации данных (`mtime`): ведь ни те, ни другие при перемещении/переименовании файла не изменяются.

Во-вторых, для выполнения этих действий можно не иметь никаких вообще прав доступа к файлам, достаточно иметь право на изменение каталогов, в которых они переименовываются или перемещаются: ведь имя файла фигурирует только в составе каталога и нигде более.

Аналогичный смысл имеет и удаление файлов, выполняемое командой

```
$ rm filename
```

в которой аргументов, означающих имена подлежащих удалению файлов, может быть произвольное количество. Как и при перемещении, при этом не затрагиваются ни метаданные, ни данные файлов, а только удаляются их имена из родительских каталогов. И потому для удаления файлов опять же не обязательно иметь какие-либо права в их отношении, достаточно прав на изменение содержащих их каталогов.

Командой `rm` файлы-аргументы будут удалены в общем случае без предупреждения. Подобно командам `cp` и `mv`, для команды `rm` предусмотрены опции `-i` (запрос на подтверждение) и `-f` (принудительное удаление вне зависимости от настроек оболочки).

Интересный момент — удаление случайно созданных файлов с именами, "неправильными" с точки зрения системы или командной оболочки. Примером этого могут быть имена, начинающиеся с символа дефиса. Если попробовать сделать это обычным образом

```
$ rm -file
```

в ответ последует сообщение об ошибке типа

```
rm: illegal option -- l
```

то есть имя файла будет воспринято как опция. Для предотвращения этого такое "неправильное" имя следует предварить символом двойного дефиса и пробелом, означающими конец списка опций:

```
$ rm -- -file
```

В принципе, команда `rm` ориентирована на удаление обычных и прочих файлов, но не каталогов. Однако с опцией `-d` она в состоянии справиться и

с этой задачей в случае, если удаляемый каталог пуст. Наконец, опция `-R` (или `-r`) производит рекурсивное удаление каталогов со всеми их файлами и вложенными подкаталогами.

Это делает использование опции `-R` весьма опасным: возможно, набивший оскомину пример

```
$ rm -R /
```

когда при наличии прав суперпользователя уничтожается вся файловая система, и утрирован, но в локальном масштабе такая операция более чем реальна.

Специально для удаления каталогов предназначена команда

```
$ rmdir
```

которая способна удалить только пустой каталог. Кроме того, с опцией `-p` она может сделать это и в отношении каталогов родительских, но также только в том случае, если они не содержат файлов.

Архивация и компрессия

Архивация и компрессия — это уже не только манипулирование файлами, но и изменение их контента. Тем не менее, рассмотрим их в этом разделе, ведь с позиций пользователя их смысл близок копированию файлов. Архивация — это сборка группы файлов или каталогов в единый файл, содержащий не только данные файлов-источников, но и информацию о них: имена файлов и каталогов, к которым они приписаны, атрибуты принадлежности, доступа и времени, что позволяет восстановить как данные, так и их структуру из архива в первозданном виде. Компрессия же предназначена исключительно для уменьшения объема, занимаемого файлами на диске (или ином носителе).

Для архивации и компрессии предназначены самостоятельные команды. Хотя архивацию и компрессию можно объединить в одной конструкции или представить так, будто они выполняются как бы в едином процессе.

Традиционные средства архивации UNIX-систем — команды `cpio` и `tar`. Суть первой, как можно понять их названия, — копирование файлов в файл архива и из файла архива. Используется она в трех режимах.

- Первый режим copy-out, определяемый опцией `-o` (или `--create`), предусматривает считывание списка файлов (name list) со стандартного ввода и объединяет их в архив, который может быть направлен в архивный файл или на устройство для записи резервных копий. Список файлов для архивирования может представлять собой вывод какой-либо иной команды.

Так, в примере

```
$ find ./* | cpio -o > arch.cpio
```

файлы текущего каталога, найденные командой `find`, при посредстве команды `cpio` будут направлены в архивный файл `arch.cpio`.

- Второй режим (copy-in, опция `-i`, или `--extract`) осуществляет обратную процедуру: развертывание ранее созданного архива в текущем каталоге

  ```
  $ cpio -i < arch.cpio
  ```

  Здесь нужно заметить, что если разворачиваемый архив включает подкаталоги, автоматически они созданы не будут, и последует сообщение об ошибке. Для создания промежуточных каталогов команда `cpio` должна использоваться с опцией `-d` (`--make-directories`).

- В третьем режиме (copy-pass, опция `-p`, или `--pass-through`) команда `cpio` выполняет копирование файлов из одного дерева каталогов в другой, комбинируя режимы copy-out и copy-in, но без образования промежуточного архива. Список файлов для копирования считывается со стандартного ввода, а каталог назначения указывается в качестве аргумента:

  ```
  $ cpio -p dir2 < name_list
  ```

Команда `cpio` имеет множество опций, позволяющих создавать, в частности, архивы в различных форматах (для межплатформенной переносимости). Однако я на них останавливаться не буду, отсылая заинтересованных к соответствующей man-странице: команда эта не кажется мне удобной в применении. И упомянута здесь, во-первых, для полноты картины, во-вторых — универсальности ради (архивы `cpio` понимаются абсолютно всеми UNIX), в-третьих — как одно из средств преобразования пакетов, используемых в различных дистрибутивах Linux, друг в друга. Например, утилита `rpm2cpio` преобразует широко распространенный формат пакетов `rpm` в еще более универсальный `cpio`.

Основным же средством архивирования во всех UNIX-системах является команда `tar`. Обобщенный формат ее

```
$ tar [options] archiv_name [arguments]
```

где `archiv_name` — обязательный аргумент, указывающий на имя архивного файла, с которым производятся действия, определяемые главными опциями. Формы указания опций для команды `tar` очень разнообразны. Исторически первой была краткая форма без предваряющего дефиса, что поддерживается и поныне. Однако в текущих версиях команды в целях единообразия утверждена краткая форма с предваряющим дефисом или дублирующая ее полная форма, предваряемая двумя дефисами. Некоторые опции (например, `--help`, получение справки об использовании команды) предусмотрены только в полной форме.

Главные опции и указывают на то, какие действия следует выполнить над архивом в целом:

- создание архива (опция `c`, `-c` или `--create`);
- просмотр содержимого существующего архива (опция `t`, `-t` или `--list`);
- распаковка архива (опция `x`, `-x`, `--extract` или `--get`).

Легко понять, что при работе с архивом как целым одна из этих главных (т. н. функциональных) опций обязательна. При манипулировании же фрагментами архива они могут подменяться другими функциональными опциями, как то:

- `r` (или `--append`) — добавление новых файлов в конец архива;
- `u` (или `--update`) — обновление архива с добавлением не только новых, но и модифицированных (с меньшим значением атрибута `mtime`) файлов;
- `-A` (`--catenate` или `--concatenate`) — присоединение одного архива к другому;
- `--delete` — удаление именованных файлов из архива;
- `--compare` — сравнение архива с его источниками в файловой системе.

Прочие (очень многочисленные) опции можно отнести в разряд дополнительных, они определяют условия выполнения основных функций команды. Однако одна из таких дополнительных опций `f` (`-f` или `--file`), значение которой — имя файла (в том числе файла устройства, и не обязательно на локальной машине), также является практически обязательной. Дело в том, что команда `tar` (от tape archiv) изначально создавалась для прямого резервного копирования на ленту стриммера, и именно это устройство подразумевается в качестве целевого по умолчанию. Так что в нынешних условиях имя архивного файла в качестве значения опции `f` следует указывать явно. Причем некоторые реализации команды `tar` требуют, чтобы в списке опций она стояла последней.

Проиллюстрируем сказанное несколькими примерами. Так, архив из нескольких файлов текущего каталога создается следующим образом:

```
$ tar cf arch_name.tar file1 ... file#
```

Если задать дополнительную опцию `v`, ход процесса будет отображаться на экране, это целесообразно, и в дальнейших примерах эта опция будет использоваться постоянно.

С помощью команды `tar` можно заархивировать и целый каталог, включая его подкаталоги любого уровня вложенности, причем — двояким образом. Так, если дать команду

```
$ tar cvf arch_name.tar *
```

файлы текущего каталога (включая подкаталоги) будут собраны в единый архив, но без указания имени каталога родительского. А командой

```
$ tar cvf arch_name.tar dir
```

каталог `dir` будет упакован с полным сохранением его структуры.

С помощью команды

```
$ tar xvf arch_name.tar
```

будет выполнена обратная процедура — распаковка заархивированных файлов в текущий каталог. Если при архивировании в качестве аргумента было указано имя каталога, а не набора файлов (пусть даже в виде шаблона), этот каталог будет восстановлен в виде корневого для всех разархивируемых файлов.

При извлечении файлов из архива никто не обязывает нас распаковывать весь архив, при необходимости это можно сделать для одного нужного файла, следует только указать его имя в качестве аргумента.

```
$ tar xvf arch_name.tar filename
```

Правда, если искомый файл находился до архивации во вложенном подкаталоге, потребуется указать и путь к нему, от корневого для архива каталога, который будет различным для двух указанных схем архивации. Ну, а для просмотра того, каким образом был собран наш архив, следует воспользоваться командой

```
$ tar tf arch_name.tar
```

Если архив собирался по первой схеме (с именами файлов в качестве аргументов, вывод ее будет примерно следующим:

```
dir2/
dir2/file1
example
new
newfile
tee.png.
```

При втором способе архивации мы увидим на выводе нечто вроде

```
dir1/
dir1/example
dir1/new
dir1/newfile
dir1/tee.png
dir1/dir2/
dir1/dir2/file1.
```

В данном примере опция `v` была опущена. Включение ее приведет к тому, что список файлов будет выведен в длинном формате, подобном выводу команды `ls -l`:

```
drwxr-xr-x alv/alv    0 10 май 11:03 2002 dir2/
-rw-r--r-- alv/alv    0 10 май 11:03 2002 dir2/file1
...
```

Команда `tar` имеет еще множество дополнительных опций, призванных предотвращать перезапись существующих файлов, осуществлять верификацию архивов, учитывать при архивации разного рода временные атрибуты, вызывать для исполнения другие программы. К некоторым опциям я еще вернусь после рассмотрения команд компрессии, другие же предлагается изучить самостоятельно, воспользовавшись страницей экранной документации:

```
man tar
```

Команд для компрессии файлов несколько, но реальный интерес ныне представляют две парные утилиты: `gzip`/`gunzip` и `bz2`/`bunzip2`. Первый член каждой пары, как легко догадаться из названия, отвечает за компрессию, второй — за декомпрессию файлов (хотя посредством должных опций они легко меняются ролями).

Команда `gzip` — это традиционный компрессор UNIX-систем, разработанный в рамках проекта GNU и сменивший в сей роли более старую утилиту `compress`. Простейший способ ее использования

```
$ gzip filename
```

где в качестве аргументов будет выступать имя файла. При этом исходный несжатый файл подменяется своей сжатой копией, которой автоматически присваивается расширение `gz`.

В качестве аргументов может выступать и произвольное количество имен файлов, каждый из них будет заменен сжатым файлом `*.gz`. Более того, с опцией `-r` может быть выполнено рекурсивное сжатие файлов во всех вложенных подкаталогах. Подчеркну, однако, что никакой архивации команда `gzip` не производит, обрабатывая за раз только единичный файл. Фактически форма

```
$ gzip file1 file2 ... file#
```

просто эквивалент последовательности команд

```
$ gzip file1
$ gzip file2
...
$ gzip file#
```

Правда, объединение компрессированных файлов возможно методом конкатенации (с помощью команды `cat`) или посредством архивирования командой `tar`, и о том, и о другом будет сказано чуть позже.

Команда `gzip` имеет и другие опции, указываемые в краткой (однобуквенной) или полной нотации. В отличие от `tar`, знак дефиса (или, соответственно, двойного дефиса) обязателен в обоих случаях. Так, опциями `-1` ... `-9` можно задать степень сжатия и, соответственно, время исполнения процедуры: `-1` соответствует минимальному, но быстрому сжатию, `-9` — максимальному, но медленному. По умолчанию в команде `gzip` используется опция `-6`, обеспечивающая разумный компромисс между скоростью и компрессией.

Благодаря опции `-d` (`--decompress`) команда `gzip` может выполнить развертывание сжатого файла, заменяя его оригиналом без расширения gz. Хотя в принципе для этого предназначена команда `gunzip`:

```
$ gunzip file.gz
```

Использование этой команды настолько прозрачно, что я задерживаться на ней не буду.

В последнее время широкое распространение получил компрессор `bzip2`, обеспечивающий большую (на 10—15%) степень сжатия, хотя и менее быстродействующий. Использование его аналогично `gzip`, с деталями его можно ознакомиться с помощью страницы экранной документации `man bzip2`. Итоговый компрессированный файл получает имя вида `*.bz2` и может быть распакован командой `bunzip2` (или командой `bzip2 -d`). Следует только помнить, что форматы `*.gz` и `*.bz2` не совместимы между собой. Соответственно, первый не может быть распакован программой `bunzip2`, и наоборот.

Поскольку программы `tar` и `gz` обеспечивают каждая свою сторону обработки файлов, возникает резонное желание использовать их совместно. Самый простой способ сделать это — воспользоваться командой `tar` с опцией `z`. Например:

```
$ tar cvzf dir.tar.gz dir/
```

Обратите внимание, что суффикс `gz` в этом случае нужно указывать в явном виде, автоматически он к имени архива не присоединяется, и компрессированный архив будет иметь вид `dir.tar`. Поскольку в UNIX расширения имен файлов не играют той сакральной роли, что в MS-DOS, это не помешает распаковке такого файла командой

```
$ tar xvzf dir.tar
```

Опция `z` сама по себе никакой компрессии не выполняет, она просто вызывает компрессор `gzip` для сжатия каждого из архивируемых файлов. Аналогичный смысл имеет и опция `j`, только ею для этой цели привлекается

команда `bzip2` (в некоторых системах для вызова последней из команды `tar` используется опция `y`).

При использовании команды `tar` с опцией `z` (или `j`) исходные файлы остаются в неприкосновенности. Следует помнить, что архив сжатых файлов не может быть обновлен командой `tar` с параметрами `r` или `u`.

Есть и другой способ совместной архивации и компрессии — просто последовательность команд

```
$ tar cf dir.tar *
$ gzip dir.tar
```

В результате образуется сжатый архив, внешне такой же файл `dir.tar.gz`. Хотя в принципе архив сжатых файлов и сжатый архивный файл — это разные вещи (можно заметить, что они даже различаются по объему, хотя и всего на несколько байтов), сжатый архив также может быть благополучно развернут командой `tar` с опцией `z`. И столь же очевидно, что он не может быть ни пополнен, ни обновлен средствами архиватора `tar`.

Компрессированные архивы, созданные `tar` и `gzip`/`bzip2`, — общепринятый в UNIX-системах метод распространения файлов. Однако иногда для совместимости с ОС, не допускающими двух точек в имени файла (знаете такую ОС?), компрессированным tar-архивам присваивается суффикс `.tgz`. Можно встретить и файлы `.tbz2` (или даже `.tbz` — именно такой вид имеют пакеты в пятой ветке FreeBSD). Нетрудно догадаться, что это те же архивы `*.tar.bz2`.

Резервное копирование

Архивы, как правило, создаются для целей резервного копирования, то есть записи их на какой-либо внешний носитель. В качестве последних в настоящее время практически могут рассматриваться только внешние винчестеры и оптические диски (CD-R/RW и записываемые DVD разных форматов). И потому способы обращения с ними резонно рассмотреть тут же, в интермедии о файловых операциях.

Можно выделить два основных способа резервного копирования: создание точных слепков файловой системы или отдельных ее фрагментов и запись архивов. Первый способ применяется, например, при переносе системы с одного носителя на другой, второй же — для сохранения данных на внешних носителях.

Обычный способ переноса файловых систем — классическая утилита `dd`, описанная в одном из предыдущих разделов. Для использования ее в этом качестве достаточно указать файл устройства-источника и файл целевого устройства.

Например, директива

```
$ dd if=/dev/ad0s1a of=/dev/ar0s1a
```

воспроизведет корневую файловую систему дискового раздела, указанного в качестве первого аргумента, на разделе второго диска. При этом нужно учитывать, что каталоги корневой файловой системы, представляющие точки монтирования самостоятельных файловых систем на отдельных разделах (такими обычно являются `/usr`, `var`, `/home`), затронуты не будут: команду `dd` придется повторить с указанием соответствующих источников и целевых устройств.

Важно также, что команда `dd` не требует ни предварительного создания файловой системы на целевом носителе, ни его монтирования. Ибо механизм ее работы — поблочный перенос всего содержимого устройства-источника на устройство-цель.

В BSD-системах та же задача может быть решена с помощью команды `cpdup`, также упоминавшейся ранее. Правда, она требует предварительного создания разделов на целевом носителе, файловых систем на разделах и их монтирования в структуру текущей корневой файловой системы. Вот как используется `cpdup` при ручной установке ОС DragonFlyBSD (без помощи программы BSD Installer):

```
$ cpdup / /mnt
$ cpdup /var /mnt/var
$ cpdup /etc /mnt/etc
$ cpdup /dev /mnt/dev
$ cpdup /usr /mnt/usr
```

Здесь каталоги `/`, `/var` — точки монтирования корня и отдельных его ветвей файловой системы установочного LiveCD, а `/mnt`, `/mnt/var` — заблаговременно созданные, отформатированные и смонтированные разделы на винчестере, на который инсталлируется DragonFlyBSD.

Если потребность в точном реплицировании файловых систем возникает не так уж и часто, то сохранение архивов данных — процедура достаточно регулярная (по крайней мере, должна ею быть). И наиболее распространенными носителями для архивов ныне являются диски CD-R/RW, процедуру записи которых я и рассмотрю далее. Конечно, записывающие DVD-устройства приобретают все большую популярность, однако этот вопрос я оставляю для изучения любопытными читателями, ибо у меня таких устройств нет, и опыта общения с ними я не имею.

Мне известны два инструмента для записи дисков: кроссплатформенный комплект cdrtools и утилита burncd, используемая только в BSD-системах. Первый к настоящему времени не описал только ленивый. А вот утилита

burncd известна относительно мало, и по причинам, которые станут ясными несколько позже, заслуживает подробного рассмотрения.

Подобно большинству исконных BSD-инструментов, использование burncd для записи дисков — дело не простое, а очень простое. И требует оно только наличия записывающего привода с ATAPI-интерфейсом. При этом нет необходимости ни в каких специфических настройках типа включения эмуляции SCSI через IDE (без чего до недавнего времени было не обойтись в Linux при использовании cdrtools).

Правда, обычно запись CD-диска начинается с создания его образа. Для чего требуется программа mkisofs из все того же пакета cdrtools. Хотя во FreeBSD и DragonFlyBSD она доступна в качестве самостоятельного порта или автономного бинарника, не требующего установки прочих компонентов оригинального пакета. Собственно создание образа происходит так:

```
$ mkisofs -R -J -o name.iso path2data
```

Здесь опция `-R` обеспечивает поддержку расширения стандарта ISO9660 — Rock Ridge для UNIX-систем (длинные имена, множественные точки в именах файлов, сохранение атрибутов доступа и принадлежности файлов и каталогов). Опция `-J` дает поддержку расширения Jouliet для систем семейства Windows (то есть длинные имена файлов будут видны и там). Опция `-o` имеет своим значением имя файла создаваемого iso-образа. Ну, а `path2data` — путь к каталогу, из содержимого которого будет создаваться образ.

Непосредственно запись диска выполняется BSD-утилитой burncd. Например, это можно сделать такой командой:

```
$ burncd -e -s max -f /dev/acd0 data iso_name fixate
```

Значения опций — следующие:

- `-e` — обеспечивает выдвижение лотка после записи;
- `-s` — скорость записи (по умолчанию 4, значение `max` обеспечивает максимально возможную для данного накопителя и заготовки);
- `-f` — имя файла устройства (в примере `/dev/acd0`).

Команда `fixate` указывает на фиксирование сессии (подразумевается односессионная запись). Ну, а `data` предписывает запись диска с данными (а не аудиодиска) с образа `name.iso`.

У burncd есть еще несколько опций, с которыми можно ознакомиться посредством

```
$ man 8 burncd
```

В частности, полезной может быть опция `-v`, выводящая информацию о ходе записи. А опция `-t` осуществит имитацию записи, что позволяет в случае ошибки избежать порчи "болванки".

Для стирания в burncd предусмотрены команды `blank` (быстрая очистка оглавления диска) и `erase` (полная очистка диска):

```
$ burncd -e -f /dev/acd0 blank
```

или

```
$ burncd -e -f /dev/acd0 erase
```

соответственно. Нужно только помнить, что вторая операция может занять много времени столько же, сколько и запись).

Для целей чисто резервного копирования (например, архива вида *.tar.gz) burncd можно использовать и без предварительного создания iso-образа (и, соответственно, без пакета mkisofs). Все, что для этого нужно (помимо заблаговременно созданного архива подходящего размера) — директива примерно такого вида:

```
$ burncd -f /dev/acd1c -s max data archive.tar.gz fixate
```

Правда, записанный таким образом диск не может быть ни прочитан в каких-либо других системах, ни смонтирован как обычный диск — доступ к нему потребует прямого обращения к файлу соответствующего устройства, например:

```
$ tar xzvf /dev/acd1c
```

Однако выполнить запись такого рода гораздо быстрее. Особенно значителен выигрыш во времени при записи очень большого массива данных. В этом случае их можно собрать в единый тарбалл, утилитой `split` разбить на фрагменты подходящего размера:

```
$ split --bytes=700m archive.tar.gz [PREFIX],
```

где в качестве префикса можно указать какое-либо мнемонически полезное значение (дату создания архива, например), после чего последовательно записать кучу образовавшихся файлов (имеющих вид `[PREFIX]aa`, `[PREFIX]ab` и т. д.) почти так же, как было сказано:

```
$ burncd -f /dev/acd1c -s max data [PREFIX]?? fixate
```

Восстановление данных из такого архива выполняется следующим образом. Сначала содержимое полученной стопки дисков последовательно копируется в файлы на винчестере:

```
$ cp /dev/acd1c path2/file#.
```

Затем они сливаются утилитой `cat` в единый архив:

```
$ cat file1 ... file# > archive.tar.gz
```

который и развертывается обычным образом.

Возможность применения burncd для резервного копирования без предварительного создания iso-образов определяет, по моему мнению, ее предпочтительность перед cdrecord. Однако на ее использование накладывается не-

сколько ограничений. Во-первых, как я уже сказал, она существует только в BSD-системах, и это, конечно, главное. Во-вторых, она работает только с ATAPI-приводами. И если вероятностью наличия в пользовательской машине CD-R/RW с SCSI-интерфейсом можно пренебречь, то записывающие USB-устройства получают все большее распространение, а перед POSIX-системой последние предстанут в виде SCSI CD ROM. И, наконец, с некоторыми моделями даже ATAPI CD ROM burncd отказывается работать категорически (правда, мне встречался один-единственный такой привод, и то уже давно).

Тем не менее, любое из приведенных ограничений может быть причиной обращения к утилите cdrecord из пакета cdrtools. Как уже говорилось, это кроссплатформенный инструмент, который может быть собран в любом UNIX.

До недавнего времени cdrecord напрямую мог работать только с SCSI-приводами, практически не встречающимися в пользовательских машинах. Это при наличии обычного ATAPI CD-R/RW требовало некоторых ухищрений — включения эмуляции SCSI через IDE в Linux или использования модуля ATAPI/CAM в BSD. Разумеется, это не возбраняется и поныне, однако проще воспользоваться возможностью последних версий cdrecord общаться с ATAPI-приводами напрямую, что и будет предметом последующего рассказа (тем более что, повторяю, работа с cdrecord при эмуляции SCSI через IDE была многократно описана ранее).

Дополнительный плюс отказа от эмуляции SCSI для CD-привода — возможность включения для него DMA-режима, что существенно ускоряет скорость как записи, так и (особенно) чтения компакт-дисков.

Запись дисков посредством cdrecord, в отличие от burncd, требует обязательного создания iso-образа. Благо, соответствующая утилита mkisofs входит в состав пакета, а сам процесс выполняется точно так же, как было описано.

Далее, программа cdrecord сохранила некоторые реликты своего SCSI-происхождения. И ATAPI-устройство, на которое она записывает, все равно должно иметь номер SCSI-облика (вида `dev=#,#,#`). Он определяется командой

```
$ cdrecord -scanbus dev=ATAPI:
```

вывод которой должен выглядеть примерно так:

```
Cdrecord 2.0 (i686-pc-linux-gnu) Copyright (C) 1995-2002 JЖrg Schilling

scsidev: 'ATAPI:'

devname: 'ATAPI'

scsibus: -1 target: -1 lun: -1

Warning: Using ATA Packet interface.
```

```
Warning: The related libscg interface code is in pre alpha.
Warning: There may be fatal problems.
Using libscg version 'schily-0.7'
scsibus0:
        0,0,0     0) 'TEAC    ' 'CD-W540E       ' '1.0C' Removable CD-ROM
        0,1,0     1) *
        0,2,0     2) *
        0,3,0     3) *
        0,4,0     4) *
        0,5,0     5) *
        0,6,0     6) *
        0,7,0     7) *
```

Обращаем внимание на цифры `0,0,0` в первой строке, именно они будут фигурировать далее в опциях cdrecord наряду с именем протокола (в виде `dev=ATAPI:0,0,0`). Логики в этом немного, особенно если задуматься над смыслом трех сакраментальных цифр, которые определяют положение устройства на SCSI-шине (!). Но пакет cdrtools вообще не в ладах с достижениями мысли товарища Аристотеля, поэтому просто запоминаем их.

Дальше все просто. Легко догадаться, что для записи диска программа cdrecord потребует минимум двух аргументов: имени устройства и имени файла iso-образа:

```
$ cdrecord dev=ATAPI:0,0,0 name.iso
```

Как обычно, в дополнительных опциях нас никто не ограничивает. В частности, можно задать опцию `-speed=#`, где в качестве значения `#` выступит желаемая скорость записи. Но можно этого и не делать: без явного указания скорости cdrecord будет писать максимально быстро, насколько это возможно для данных привода. Так что интересней будет дать опцию `-v` (от обычного verbose), которая обеспечит нас подробной информацией о ходе записи (в том числе о таких параметрах, как максимальная, минимальная и средняя скорость записи, процент заполнения буфера и т. д.).

Еще одна опция `-eject` предписывает выдвигать лоток с болванкой по окончании записи. В финале команда для записи созданного образа приобретает такой вид:

```
$ cdrecord -v -eject [-speed=12] dev=ATAPI:0,0,0 name.iso
```

Раз уж речь зашла о перезаписи, не лишним будет вспомнить, как к ней подготовиться (то есть освободить болванку от записанного ранее). Делается это той же командой `cdrecord`, но с опцией `blank=значение`, каковое может быть `fast` (быстрая очистка, при которой стирается оглавление без физического уничтожения данных) или `all` (полная очистка диска, которая занимает ровно столько же времени, сколько и его запись на всю катушку).

До сих пор речь шла о записи диска "в один присест". Однако никто этого делать не заставляет: мы ведь знаем, что бывает мультисессионая запись, если данных на полный диск сразу не накопилось. В этом случае создаем образ диска, как описано ранее (объемом хоть в 10 Мбайт — только помним, что на каждую сессию, вне зависимости от количества записанной информации, некоторое количество этого самого объема теряется), и записываем точно так же, добавив лишь опцию `-multi`:

```
$ cdrecord -v -eject -multi [-speed=#] dev=ATAPI:0,0,0 name1.iso
```

А вот со следующими сессиями нужно быть повнимательней. Во-первых, команда `mkisofs` в этом случае потребует опции `-C` (вероятно, от continue). Очевидно, что уже на стадии создания образа для второй сессии (и всех последующих) требуется информация, а где же завершилась предыдущая? Правда, получить ее просто, заранее вставив бывшую в употреблении недописанную болванку, даем команду

```
$ cdrecord -msinfo dev=ATAPI:0,0,0
```

которая выдаст нам последний использованный в прошлой сессии трэк — некое число `###`, которое и подставим в командную строку в качестве значения опции `-C`.

Далее, команде `mkisofs` требуется сообщить еще и имя устройства, на котором этот трэк был записан. Для чего предназначена опция `-M`. А вот значением ее будет (внимание!) не трехзначный номер устройства, полученный при выводе команды

```
cdrecord -scanbus
```

как можно было бы ожидать, а просто имя файла устройства типа `/dev/cdroms/cdrom` (или как он точно обозначается в данной системе).

В итоге команда `mkisofs` для создания образа второй и последующих сессий приобретет вид:

```
$ mkisofs -R -J -C ### -M /dev/cdroms/cdrom -o name2.iso /path2data
```

А вот записывается этот образ самым обычным образом:

```
$ cdrecord -v -eject [-multi] [-speed=#] dev=ATAPI:0,0,0 name2.iso
```

К слову сказать, любой образ диска может быть проверен. В Linux для этого он должен быть подмонтирован как loopback-устройство (что требует соответствующей поддержки в ядре, но таковая обычно в виде модуля в прекомпилированных ядрах большинства дистрибутивов имеется):

```
$ mount -o loop name.iso /mnt_point
```

После чего содержимое каталога /mnt_point просматривается обычным образом.

До сих пор речь шла о записи CD-R/RW. Однако все, сказанное в отношении mkisofs и cdrecord, применимо и к DVD-дискам (по крайней мере, к

DVD+R/RW точно). Для них точно так же можно создать образ (разумеется, несколько большего в пределе объема) и перенести его на носитель, в том числе и в несколько сессий. В результате образуется DVD с файловой системой iso9660, отличающийся от обычного компакт-диска только объемом информации. Однако DVD-носители поддерживают и другую файловую систему — UDF, позволяющую обращаться с ними как с очень большими дискетами. Правда, я этого никогда не делал и потому от описания воздержусь в надежде, что кто-нибудь восполнит мой пробел.

Как я говорил, программа cdrecord не позволяет обойтись без создания iso-образа. Однако если нет намерения его проверять, никто не обязывает его записывать на диск: можно перенаправить вывод команды `mkisofs` по конвейеру прямо на ввод `cdrecord`:

```
$ mkisofs -R -J  /path2data | cdrecord  dev=ATAPI:0,1,0 -
```

И, наконец, последнее о cdrtools: этот пакет позволяет использовать многочисленные его front end для графического режима, из которых самым удобным мне представляется пакет k3b для интегрированной среды KDE.

Венец универсализма: утилита *find*

В этом разделе речь пойдет о пакете, известном в проекте GNU как findutils. И в первую голову — о команде `find` (как, впрочем, и о тесно связанной с ней команде `xargs`). Столь высокая честь выпадает им потому, что посредством этих двух команд можно выполнить если не все, то большинство задач, возникающих при работе с файлами.

Кроме `find` и `xargs`, в состав пакета findutils входят команды `locate` и `updatedb`, о которых говорилось ранее, а также команды `bigram`, `code`, `frcode`, реального применения которым я, честно говоря, не знаю (и, соответственно, говорить о них не буду). В BSD-системах те же команды входят в базовый комплект.

Итак, апофеоз командного файлового менеджмента — утилита `find`. Строго говоря, вопреки своему имени, команда эта выполняет не поиск файлов как таковой, но рекурсивный обход дерева каталогов, начиная с заданного в качестве аргумента, отбирает из них файлы в соответствии с некоторыми критериями и выполняет над отбракованным файловым хозяйством некоторые действия. Именно эту ее особенность подчеркивает резюме команды `find`, получаемое (в некоторых системах) посредством

```
$ whatis find
find(1)                    - walk a file hierarchy
```

что применительно случаю можно перевести как "прогулка по файловой системе".

Команда `find` по своему синтаксису существенно отличается от большинства прочих UNIX-команд. В обобщенном виде формат ее можно представить следующим образом:

```
$ find аргумент [опция_поиска] [значение] \
        [опция_действия]
```

В качестве аргумента здесь задается путь поиска, то есть каталог, начиная с которого следует совершать обход файловой системы, например, корень ее:

```
$ find / [опция_поиска] [значение] \
        [опция_действия]
```

или домашний каталог пользователя:

```
$ find ~/ [опция_поиска] [значение] \
        [опция_действия]
```

Опция поиска — критерий, по которому следует отбирать файл (файлы) из определенных в аргументе частей файловой системы. В качестве таковых могут выступать имя файла (`-name`), его тип (`-type`), атрибуты принадлежности, доступа или времени.

Ну, а опция действия определяет, что же надлежит сделать с отобранными файлом или файлами. А сделать с ними, надо заметить, можно немало, начиная с вывода на экран (`-print`, опция действия по умолчанию) и кончая передачей в качестве аргументов любой другой команде (`-exec`).

Как можно видеть из примера, опция поиска и опция действия предваряются знаком дефиса, значение первой отделяется от ее имени пробелом.

Однако начнем по порядку. Опции поиска команды `find` позволяют выполнить отбор файлов по следующим критериям (символ дефиса перед опциями далее опущен, но не следует забывать его ставить):

- `name` — поиск по имени файла или по маске имени; в последнем случае метасимволы маски должны обязательно экранироваться (например, `name \*.tar.gz`) или заключаться в кавычки (одинарные или двойные, в зависимости от ситуации); этот критерий чувствителен к регистру, но близкий по смыслу критерий `iname` позволяет производить поиск по имени без различения строчных и заглавных букв;
- `type` — поиск по типу файла; этот критерий принимает следующие значения:
  - `f` (регулярный файл);
  - `d` (каталог);
  - `s` (символическая ссылка);
  - `b` (файл блочного устройства);
  - `c` (файл символьного устройства);

- `user` и `group` — поиск по имени или идентификатору владельца или группы, выступающим в качестве значения критерия; существуют также критерии `nouser` и `nogroup`, они отыскивают файлы, владельцев и групповой принадлежности не имеющие (то есть тех, учетные записи для которых отсутствую в файлах /etc/passwd и /etc/group); последние два критерия в значениях, разумеется, не нуждаются;
- `size` — поиск по размеру, задаваемому в виде числа в блоках или в байтах в виде числа с последующим символом `c`; возможны значения:
  - `n` (равно `n` блоков);
  - `+n` (более `n` блоков);
  - `-n` (менее `n` блоков);
- `perm` — поиск файлов по значениям их атрибутов доступа, задаваемых в символьной форме;
- `atime`, `ctime`, `mtime` — поиск файлов с указанными временными атрибутами; значения временных атрибутов указываются в сутках (точнее, в периодах, кратных 24 часам); возможны формы значений этих атрибутов:
  - `n` (равно указанному значению `n*24` часа);
  - `+n` (ранее `n*24` часа);
  - `-n` (позднее `n*24 часа`);
- `newer` — поиск файлов, измененных после файла, указанного в качестве значения критерия (то есть имеющего меньшее значение `mtime`);
- `maxdepth` и `mindepth` позволяют конкретизировать глубину поиска во вложенных подкаталогах, меньшую или равную численному значению для первого критерия и большую или равную — для второго;
- `depth` — производит отбор в обратном порядке, то есть не от каталога, указанного в качестве аргумента, а с наиболее глубоко вложенных подкаталогов; смысл этого действия — получить доступ к файлам в каталоге, для которого пользователь не имеет права чтения и исполнения;
- `prune` — позволяет указать подкаталоги внутри пути поиска, в которых отбора файлов производить не следует.

Кроме этого, существует еще одна опция поиска — `fstype`, предписывающая выполнять поиск только в файловой системе указанного типа; очевидно, что она может сочетаться с любыми другими опциями поиска. Например, команда

```
$ find / -fstype ext3 -name zsh*
```

будет искать файлы, имеющие отношение к оболочке Z-Shell, начиная с корня, но только в пределах тех разделов, на которых размещена файловая система Ext3fs.

Критерии отбора файлов могут группироваться практически любым образом. Так, в форме

```
$ find ~/ -name *.tar.gz newer filename
```

она выберет в домашнем каталоге пользователя все компрессированные архивы, созданные после файла с именем `filename`. По умолчанию между критериями отбора предполагается наличие логического оператора `AND` (логическое "И"). То есть будут отыскиваться файлы, удовлетворяющие и маске имени, и соответствующему атрибуту времени. Если требуется использование оператора `OR` (логическое "ИЛИ"), он должен быть явно определен в виде дополнительной опции `-o` между опциями поиска. Так, команда:

```
$ find ~/ -mtime -2 -o newer filename
```

призвана отобрать файлы, созданные менее двух суток назад, или же позднее, чем файл `filename`.

Особенность GNU-реализации команды `find` (как, впрочем, и ее тезки из числа BSD-утилит) — то, что она по умолчанию выводит список отобранных в соответствии с заданными критериями файлов на экран, не требуя дополнительных опций действия. Однако, как говорят, в других UNIX-системах (помнится, даже и в некоторых реализациях Linux мне такое встречалось) указание какой-либо из таких опций обязательно. Так что рассмотрим их по порядку.

Для выведения списка отобранных файлов на экран в общем случае предназначена опция `-print`. Вывод этот имеет примерно следующий вид:

```
$ find . -name f* -print
./file1
./file2
./dir1/file3
```

Сходный смысл имеет и опция `-ls`, однако она выводит более полные сведения о найденных файлах аналогично команде `ls` с опциями `-dgils`:

```
$ find / -fstype ext3 -name zsh -ls
88161  511 -rwxr-xr-x   1 root      root   519320 Ноя 23 15:50 /bin/zsh
```

Важное, как мне кажется, замечание. Если команда указанного вида будет дана от лица обычного пользователя (не root-оператора), кроме приведенной строки вывода последуют многочисленные сообщения вроде

```
find: /root: Permission denied
```

указывающие на каталоги, закрытые для просмотра обычным пользователем, и весьма мешающие восприятию результатов поиска. Чтобы подавить их, следует перенаправить вывод сообщения об ошибках в файл `/dev/null`:

```
$ find / -fstype ext3 -name zsh -ls 2> /dev/null
```

Идем далее. Опция `-delete` уничтожит все файлы, отобранные по указанным критериям. Так, командой

```
$ find ~ -atime +100 -delete
```

будут автоматически стерты все файлы, к которым не было обращения за последние 100 дней (из молчаливого предположения, что раз к ним три месяца не обращались, значит, они и вообще не нужны). Истреблению подвергнутся файлы в подкаталогах любого уровня вложенности, но не включающие их подкаталоги (если, конечно, последние сами не подпадают под критерии отбора).

И, наконец, опция `-exec`: именно ею обусловлено величие утилиты `find`. В качестве значения ее можно указать любую команду с необходимыми опциями, и она будет выполнена над отобранными файлами, которые будут рассматриваться в качестве ее аргументов. Проиллюстрируем это на примере.

Использовать для удаления файлов опцию `-delete`, как мы это только что сделали, — не самое здоровое решение, ибо файлы при этом удаляются без запроса, и можно случайно удалить что-нибудь нужное. И потому достигнем той же цели следующим образом:

```
$ find ~/ -atime +100 -exec rm -i {} \;
```

В этом случае (вне зависимости от настроек псевдонимов командной оболочки) на удаление каждого отобранного файла будет запрашиваться подтверждение.

Обращаю внимание на последовательность символов `{} \;` (с пробелом между закрывающей фигурной скобкой и обратным слэшем) в конце строки. Пара фигурных скобок `{}` символизирует, что свои аргументы исполняемая команда (в примере `rm`) получает от результатов отбора команды `find`, точка с запятой означает завершение команды-значения опции `-exec`, а обратный слэш экранирует ее специальное значение от интерпретации командной оболочкой.

Кроме опции действия `-exec`, у команды `find` есть еще одна, близкая по смыслу, опция `-ok`. Она также вызывает некую произвольную команду, которой в качестве аргументов передаются имена файлов, отобранные по критериям, заданным опцией (опциями) поиска. Однако перед выполнением каждой операции над каждым файлом запрашивается подтверждение.

Приведенный пример, хотя и вполне жизненный, достаточно элементарен. Рассмотрим более сложный случай — собирание в один каталог всех скриншотов в формате PNG, разбросанных по древу домашнего каталога:

```
$ find ~/ -name *.png -exec cp {} imagesdir \;
```

В результате все PNG-файлы будут изысканы и скопированы (или перемещены, если воспользоваться командой `mv` вместо `cp`) в одно место.

А теперь вариант решения задачи, которая казалась мне некогда трудноразрешимой: рекурсивное присвоение необходимых атрибутов доступа в разветвленном дереве каталогов, различных для регулярных файлов и каталогов.

Зачем и отчего это нужно? Поясню на примере. Как-то раз, обзаведясь огромным по тем временам (40 Гбайт) винчестером, я решил собрать на него все нужные мне данные, рассеянные по дискам CD-R/RW и нескольким сменным винчестерам, одни из которых были отформатированы в FAT16, другие — в FAT32, третьи — вообще в ext2fs (к слову сказать, рабочей моей системой в тот момент была FreeBSD). Сгрузив все это богатство в один каталог на новом диске, я создал в нем весьма неприглядную картину.

Ну, во-первых, все файлы, скопированные с FAT-дисков, получили (исключительно из-за неаккуратности монтирования, с помощью должных опций этого можно было бы избежать, но спешка, спешка...) биты исполняемости, хотя были это лишь файлы данных. Казалось бы, мелочь, но иногда очень мешающая; в некоторых случаях это не позволяет, например, просмотреть HTML-файл в Midnight Commander простым нажатием клавиши <Enter>. Во-вторых, для некоторых каталогов, напротив, исполнение не было предусмотрено ни для кого: то есть я же сам перейти в них не мог. В-третьих, каталоги (и файлы) с компакт-диска часто не имели атрибута изменения, а они нужны мне были для работы (и редактирования). Так что ситуация явно требовала исправления, однако проделать вручную такую работу над данными более чем в 20 Гбайт виделось немыслимым.

Да так оно, собственно, и было бы, если б не опция `-exec` утилиты `find`. Каковая позволила изменить права доступа требуемым образом. Итак, сначала отбираем все регулярные файлы и снимаем с них бит исполнения для всех, заодно присваивая атрибут изменения для себя, любимого:

```
$ find ~/dir_data -type f \
        -exec chmod a-x,u+w {} \;
```

Далее поиск каталогов и обратная процедура над итоговой выборкой:

```
$ find ~/dir_data -type d \
        -exec chmod a+xr,u+w {} \;
```

И дело в шляпе, все права доступа стали единообразными (и теми, что мне нужны). Именно после этого случая я проникся величием утилиты `find`. А ведь это еще не предел ее возможностей — последний устанавливается только встающими задачами и собственной фантазией...

Так, с помощью команды `find` легко наладить периодическое архивирование результатов текущей работы. Для этого создаем командой `tar` полный архив результатов своей деятельности:

```
$ tar cvf alldata.tar ~/*
```

А затем время от времени запускаем команду

```
$ find ~/ -newer alldata.tar \
        -exec tar uvf alldata.tar {} \;
```

Еще один практически полезный вариант использования команды `find` в мирных целях — периодическое добавление отдельно написанных фрагментов к итоговому труду жизни (например, собственным мемуарам). Впрочем, чтобы сделать это, необходимо сначала ознакомиться с командами обработки файлов, к которым мы вскоре обратимся.

А пока — об ограничении возможностей столь замечательной сцепки команды `find` с опцией действия `-exec` (распространяющемся и на опцию `-ok`). Оно достаточно очевидно: вызываемая любой из этих опций команда выполняется в рамках самостоятельного процесса, что на слабых машинах, как говорят, приводит к падению производительности (должен заметить, что на машинах современных заметить этого практически невозможно).

Тем не менее, ситуация вполне разрешима. И сделать это призвана команда `xargs`. Она определяется как построитель и исполнитель командной строки со стандартного ввода. А поскольку на стандартный ввод может быть направлен вывод команды `find`, `xargs` воспримет результаты ее работы как аргументы какой-либо команды, которую, в свою очередь, можно рассматривать как аргумент ее самой (по умолчанию такой командой-аргументом является `/bin/echo`).

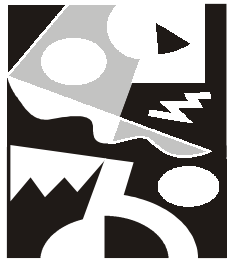
Использование команды `xargs` не связано с созданием изобилия процессов (дополнительный процесс создается только для нее самой). Однако она имеет другое ограничение — лимит на максимальную длину командной строки. Во всех BSD-системах, которые мне довелось видеть, этот лимит составляет 65 536, что определяется командой следующего вида:

```
$ sysctl -a | grep kern.argmax
```

И способы изменить этот лимит мне не известны, был бы благодарен за соответствующую информацию.

Поэтому в реальности у меня не возникало необходимости в команде `xargs`, и, соответственно, я не занимался ее глубоким изучением. Так что заинтересованных отсылаю к соответствующей man-странице.

Глава 9



Физика файловых систем

- Дисковые накопители.
- Немного о геометрии.
- Собственно о разделах.
- Особенности BSD-разметки.
- RAID и LVM.
- Общие черты файловых систем POSIX-семейства.
- Основные типы файловых систем POSIX-мира.
- Виртуальные файловые системы.

Файловая система — один из самых многозначных терминов в околокомпьютерной литературе. Им обозначается и подсистема ядра, отвечающая за управление операциями с файлами, и способ физической организации данных на носителях, и логическая структура файлов и каталогов. В настоящий момент нас будут интересовать два последних понятия. В этой главе мы рассмотрим физику размещения файлов, а логику их организации отложим до главы следующей.

Дисковые накопители

ОС POSIX-семейства поддерживают множество файловых систем. В первом приближении их можно разделить на две группы: реальные (block based файловые системы) и виртуальные, под которыми никаких реальных физических устройств не обнаруживается. Есть и промежуточный вид файловых систем — на виртуальных дисках в оперативной памяти (RAM-дисках). Основным предметом нашего рассмотрения будут файловые системы первой группы, о виртуальных файловых системах я скажу несколько слов под занавес.

Как следует из названия, block based файловые системы размещаются на блочных устройствах, в качестве которых выступают обычно диски и их разделы. И потому рассмотрению этого типа следует предпослать введение о дисковых накопителях.

Дабы не повторять это в дальнейшем, сразу оговорюсь, что почти все сказанное далее относится к дискам с интерфейсом IDE (ATA). SCSI-диски имеют свою специфику, но я с ними дела не имел, и собственных впечатлений на сей предмет у меня нет. Да и для пользователя, с ростом объема и скорости ATA-дисков, падением цены и распространением внешних накопителей с USB- и FireWire-интерфейсами, SCSI становится все менее актуальным. Хотя эту тему и придется затронуть, но не в отношении дисков, а устройств иного типа.

Для начала вспомним, что дисковые накопители, как и любые другие устройства, в любой POSIX-совместимой системе предстают перед пользователем в виде файлов специального типа — файлов устройств. Файлы эти расположены в каталоге /dev, и номенклатура их подчиняется правилам, своим для каждого представителя этого семейства ОС.

В Linux ATA-диски традиционно именовались как /dev/hd?, где hd — общее имя устройств этого класса (рискну предположить, происходящее от hard disk), а символ ? — литера, идентифицирующая конкретный его представитель (табл. 9.1).

Таблица 9.1. *Номенклатура дисковых накопителей в Linux*

Файл	Накопитель
/dev/hda	Master на 1 IDE-канале
/dev/hdb	Slave на 1 IDE-канале
/dev/hdc	Master на 2 IDE-канале
/dev/hdd	Slave на 2 IDE-канале

Номенклатура устройств статична, то есть не зависит от физического их наличия: диск, подключенный как Slave ко второй линии встроенного IDE-контроллера, всегда будет именоваться /dev/hdd, даже являясь единственным накопителем в системе (хотя трудно представить себе реальные основания для конфигурации такого рода).

Отдельно следует оговорить, что тем же правилам подчиняется именование не только дисков, но и других накопителей с интерфейсом ATA — CD ROM/R/RW, DVD любого вида или отходящих в прошлое внутренних Zip-приводов и "супер-флопов".

Одно время большинство дистрибутивов использовали файловую систему устройств (devfs), вносящую свои коррективы в номенклатуру устройств, в том числе и дисковых. Правда, ныне она отходит в прошлое, вытесняясь механизмом udev, однако традиции ее в именовании дисков живы, и о них нужно знать.

При использовании devfs для файлов любых ATA-накопителей предназначен каталог /dev/ide (в некоторых дистрибутивах файловая система устройств монтируется в каталог /devices, а каталог /dev сохраняется для совместимости). Файлы накопителей на встроенном основном IDE-контроллере локализуются в подкаталоге /dev/ide/host0 (если используется еще и дополнительный IDE-контроллер, встроенный или внешний, можно увидеть и каталог /dev/ide/host1). А внутри него есть два подкаталога, соответствующие IDE-каналам — /dev/ide/host{/bus0,/bus1}, каждый из которых может делиться надвое — на каталоги target0 и target1, по количеству подключенных устройств. Внутри каталога target0(1) имеется минимум еще один подкаталог lun0. А уж в нем размещается непосредственно файл дискового устройства — disc.

Таким образом, полное обозначение дискового раздела будет выглядеть как

```
/dev/ide/host0/bus0/target0/lun0/disc
```

для первого диска на первом канале основного IDE-контроллера.

Как и в случае старой номенклатуры, правила ее распространяются и на недисковые ATA-накопители. С той только разницей, что именование привода в этой нотации (назовем ее полной) будет выглядеть так:

```
/dev/ide/host0/bus1/target0/lun0/cd
```

Предусмотрен и более краткий способ обращения к файлам устройств — через символические (изредка жесткие) ссылки; назовем этот способ краткой нотацией. Для файлов дисковых накопителей (независимо от интерфейса, IDE или SCSI) они собраны в каталоге /dev/discs (для файлов CD-приводов, например, в каталоге /dev/cdrom) с подкаталогами disc0, ..., disc#. И потому к приведенному в качестве примера диску можно обратиться и так:

```
/dev/discs/disc0/disc
```

а CD-привод будет именоваться следующим образом:

```
/dev/cdroms/cdrom0
```

Краткая нотация имеет динамический характер, то есть первый в порядке подключения диск всегда будет представлен устройством /dev/discs/disc0/disc, а второй — /dev/discs/disc1/disc, вне зависимости от того, на каком канале или в каком качестве (Master или Slave) они реально подключены.

Наконец, для совместимости со старыми временами (и старыми привычками) в большинстве дистрибутивов поддерживается и номенклатура обратной совместимости, принятая до внедрения devfs. То есть все тот же дисковый накопитель из примера можно обозвать просто — /dev/hda.

Таким образом, подключение devfs — не препятствие для использования старой номенклатуры накопителей. Правда, это требует соответствующей настройки демона devfsd (точнее, его конфигурационного файла), отвечающего за обратную совместимость. И, при желании, может быть отменено. В частности, такие дистрибутивы, как CRUX и Archlinux, на стадии установки обратной совместимости не используют, при подготовке дисков здесь именовать их приходится только в новой нотации, краткой или полной, без разницы.

При использовании файловой системы devfs имена файлов создаются только для реально существующих в системе устройств (в том числе, для устройств "горячего" подключения, типа PC-карт или USB-драйвов, и "на лету"). Поэтому, если в системе имеется только единственное IDE-устройство, скажем, жесткий диск как мастер на первом канале, бесполезно было бы искать файлы устройств с именами, отличными от приведенного в примере.

В последнее время в большинстве дистрибутивов Linux вместо файловой системы устройств, активно внедряется механизм udev (о чем будет говориться далее). При этом происходит возврат к старой номенклатуре устройств, в том числе и дисковых накопителей: /dev/hda и т. д. Однако именование устройств в udev может следовать (совместимости ради) и традициям devfs, это зависит от конкретного дистрибутива и настроек соответствующего пакета. Важно только, что, как и в devfs, в udev файлы создаются только для реально имеющихся в машине устройств.

Несколько отлична номенклатура накопителей во FreeBSD и DragonFlyBSD. Здесь файлы устройств ATA-дисков именуются в общем виде как ad#. Как нетрудно догадаться, аббревиатура ad происходит от ATA Disk, а # — численный идентификатор, соответствующий номеру конкретного накопителя в порядке его подключения к IDE-разъемам (табл. 9.2).

Таблица 9.2. *Номенклатура дисковых накопителей во FreeBSD и DragonFlyBSD*

Файл	Накопитель
/dev/ad0	Master на 1 IDE-канале
/dev/ad1	Slave на 1 IDE-канале
/dev/ad2	Master на 2 IDE-канале
/dev/ad3	Slave на 2 IDE-канале

Во FreeBSD и DragonFlyBSD по умолчанию также принята статичная нумерация файлов дисковых устройств, хотя это можно изменить при перекомпиляции ядра.

В отличие от Linux, во FreeBSD для приводов CD-ROM принята своя номенклатура — файлы этих устройств именуются как /dev/acd0, /dev/acd1, вне зависимости от контроллера, к которому они подключены. Имена SCSI-приводов для компакт-дисков — /dev/cd0 и т. д. Самих по себе устройств с таким интерфейсом в пользовательской машине почти наверняка нет. Однако в этом качестве перед системой предстают обычные ATAPI CD- и DVD-приводы, если в ядре включена поддержка эмуляции SCSI (CAM — Common Access Metod). А она требуется для записи DVD, и в некоторых случаях также и CD-дисков.

Во FreeBSD пятой ветки (и выше) также используется файловая система устройств devfs. Однако влияния на номенклатуру устройств она не оказывает: все низкуровневые утилиты в этой ОС разрабатываются совместно с ядром, и потому задача обратной совместимости тут не ставится. Единственно, что в каталоге /dev установленной FreeBSD пятой ветки будут присутствовать только файлы реально имеющихся устройств, например, /dev/ad0 и /dev/acd0. Тогда как в более старых ветвях мы увидели бы также созданные "про запас" файлы /dev/ad0, /dev/ad1 и /dev/ad2, хотя попытка обращения к ним ничего, кроме сообщения об ошибке, и не дала бы.

Наконец, прочие BSD-системы (Net- и OpenBSD) также располагают собственной терминологией в отношении дисковых устройств. Соответствующие им файлы именуются как wd0 и т. д. (табл. 9.3).

Таблица 9.3. *Номенклатура дисковых накопителей в NetBSD и OpenBSD*

Файл	Накопитель
/dev/wd0	Master на 1 IDE-канале
/dev/wd1	Slave на 1 IDE-канале
/dev/wd2	Master на 2 IDE-канале
/dev/wd3	Slave на 2 IDE-канале

Причем в отличие и от Linux, и от FreeBSD, в OpenBSD номенклатура дисков по умолчанию динамична: диски нумеруются по порядку, вне зависимости от линии контроллера и положения на ней (Master или Slave). То есть имена устройств в табл. 9.3 будут отвечать реальности только при наличии четырех дисков. Хотя, как и во FreeBSD, такой порядок может быть изменен при переконфигурировании ядра.

Жесткие диски — не единственные накопители, которые можно обнаружить в современной машине: все более возрастает роль разного рода сменных носителей, вытеснивших, наконец, из этой ниши трехдюймовый дисковод. Накопители такие бывают двух типов: с интерфейсом FireWire и USB. О первых не могу сказать ничего определенного, не только не пользовался, но и даже не видел. А вот USB-драйвы (в просторечии — флэшки) заслуживают нескольких слов.

Потому что это и есть те устройства из числа распространенных, которые изображают из себя SCSI-диски. И соответствующие им файлы устройств подчиняются правилам, установленным для последних. То есть в Linux по старой (и очень новой — при udev) номенклатуре они будут именоваться как /dev/sda, /dev/sdb. По правилам же devfs в полной нотации, точно как SCSI-диски:

```
/dev/scsi/host0/bus0/target0/lun0/disc
```

Впрочем, в краткой нотации USB-привод будет выглядеть обычным винчестером:

```
/dev/discs/disc1/disc
```

Во FreeBSD USB-накопители также предстают в качестве SCSI-дисков, и, в соответствии с принятыми здесь правилами, имена соответствующих им файлов устройства будут выглядеть как /dev/da1, /dev/da2.

К слову сказать, точно так же, как и флэшки, будут обычно выглядеть и цифровые камеры с USB-интерфейсом, точнее, встроенные в них или съемные носители информации; как, впрочем, и карт-ридеры для чтения последних. Да и именование внешних мобильных винчестеров с USB-интерфейсом (типа Fujitsu HandyDrive) подчиняется тем же правилам.

Теоретически ничто не мешает создавать файловые системы непосредственно на дисковых устройствах. Однако практически так почти никогда не поступают, диски принято делить на разделы (partitions в терминологии DOS/Windows и Linux, slices в терминологии FreeBSD). Однако прежде чем говорить о разбиении диска, необходимо сказать о "геометрии".

Немного о геометрии

Доступная BIOS геометрия диска описывается в терминах цилиндр/головка/сектор (cylinders/heads/sectors, C/H/S). Фигурально говоря, головки считывают информацию с концентрических магнитных дорожек (tracks), на которые поделена каждая дисковая пластина. Вертикальная совокупность треков с одинаковыми номерами на всех пластинах, составляющих диск как физическое устройство, и образует цилиндр. А сектора нарезают

пластину, вместе с ее треками, на радиальные фрагменты, именуемые блоками. Таким образом, блок лежит на пересечении (в пространстве) цилиндра, трека и сектора.

Число треков и секторов в современных дисках обычно фиксировано (вернее, предстает таковым в BIOS): 255 треков нарезается на 63 сектора каждый, что в совокупности дает 16 065 блоков на цилиндр. А количество цилиндров определяется объемом диска. Важно здесь только то, что головки диска механически двигаются синхронно по поверхности всех пластин. То есть если на одной пластине информация считывается с первого трека, то и все прочие головки перемещаются на ту же дорожку, каждая на своей пластине.

Повторяю, все это условно хотя бы потому, что понятие цилиндра в геометрическом смысле слова очень трудно применить к современным дискам. Но разбираться с этой геометрией — дело firmware и BIOS, для нас же интересны именно цилиндры, как совокупность треков, к которым осуществляется синхронный доступ, и блоки — минимальные кванты дискового пространства.

Образующие цилиндры треки создаются при первичной заводской разметке диска — низкоуровневом форматировании. Очевидно, что доступ к данным в пределах одного цилиндра или группы соседних будет выполнен быстрее, чем к данным, записанным частично на первый и, скажем, на последний цилиндр диска. Этот случай не столь уж невероятен, как может показаться: в DOS, где пространство, занятое стертыми файлами, помечается как неиспользуемое, но реально перезаписывается только тогда, когда по-настоящему свободное место на диске вообще исчерпано, подобная ситуация вполне могла бы возникнуть.

Так вот, чтобы свести к минимуму вероятность разнесения данных по разобщенным цилиндрам, и придуманы были дисковые разделы. В единый раздел объединяется группа смежных цилиндров.

Где кончается один раздел и начинается другой? Резонные люди из Одессы сказали бы, что полиция кончается именно там, где начинается Беня Крик. Так и для нас очевидно, что для каждого из разделов следует хранить сведения о его начале и конце (то есть номера первого и последнего из задействованных в нем цилиндров). Где их хранить? Для ответа на этот вопрос следует обратиться к понятию блока.

Как и треки, дисковые блоки (или физические, поскольку есть еще блоки логические, но это относится уже к файловым системам, о которых речь пойдет в следующем разделе) создаются при низкоуровневом форматировании, и пользователь влиять на них не может. Размер их также всегда одинаков и равен 512 байтам.

Обмен данными с диском возможен минимум 512-байтными блоками. То есть мелкие текстовые файлы размером в пару символов все равно захватят под себя аж 512 байтов, не меньше. С другой стороны, считывание данных блоками по 512 байтов будет происходить быстрее, чем если бы при каждом обращении головки к диску данные считывались бы побайтно. Однако и это относится уже к теме файловых систем.

Пока же нам интересен один-единственный блок, образованный первым сектором на первом треке первого цилиндра. Он резервируется под служебную область диска, именуемую главной загрузочной записью (MBR — Master Boot Record), которая и считывается BIOS при старте машины. Очевидно, что по прямому назначению MBR используется только в том случае, если диск определен в Setup BIOS как загрузочный (или просто является единственным в системе). Однако поскольку использование каждого конкретного диска остается на усмотрение пользователя, место под него отводится всегда.

Внутри нулевого блока, помимо прочего (в частности, кода какого-либо начального загрузчика, который может быть туда записан), есть еще один зарезервированный участок. Он предназначен для таблицы разделов (Partition Table), под которую испокон веков (со времен самой первой IBM PC, кажется) отведено 64 байта. В эту таблицу записываются (или могут быть записаны) данные о разделе (разделах) в определенном, доступном пониманию BIOS формате.

А формат этот предусматривает указание для каждого раздела его стартового блока, размера в байтах, идентификатора типа файловой системы (это, вопреки названию, совсем не то же самое, что файловая система, о которой речь пойдет в одном из последующих разделов) и (только для одного из разделов) флага активности (помечающего данный раздел как загрузочный). Всего информации, необходимой для описания дискового раздела, набегает 16 байтов. А поскольку, как мы помним, под всю таблицу разделов этих байтов отведено лишь 64, без калькулятора можно подсчитать, что предельное количество разделов на диске 4. Эти разделы называются первичными или, не совсем точно, физическими. В большинстве случаев такие разделы могут быть также поделены на части — разделы логические (о чем речь впереди).

Повторю еще раз: это относится только к машинам с PC BIOS, то есть обычным персоналкам. На всякого рода PowerPC, Sparc и тому подобных станциях все может быть совсем по-другому.

Как можно заметить, в описание раздела входит идентификатор типа файловой системы. Это некоторое число (в BSD обычно в десятичном представлении, в Linux, например, в шестнадцатеричном), которое ставится в соответствие с файловой системой. Так, раздел, предназначенный для

FreeBSD (и DragonFlyBSD — тип его называется 4.2BSD), имеет идентификатор 165 (десятичный) или A5 (шестнадцатеричный), раздел для Linux (Linux native) — 131 или 83 соответственно, FAT16 — 6, расширенный раздел (DOS Extended) — 5 и т. д. В табл. 9.4 приведены значения наиболее важных для пользователя Linux и BSD-систем идентификаторов типов разделов (по выводу командой `fdisk`, то есть в шестнадцатеричном исчислении) с их краткой характеристикой.

Таблица 9.4. *Значения идентификаторов типов разделов*

Значение (hex)	Название	Характеристика
1	FAT12	Файловая система DOS-дискет
4	FAT16<32M	Файловая система старых (ниже 4) версий DOS
5	Extended	Расширенный раздел DOS
6	FAT16	Обычный раздел DOS (до 2 Гбайт)
b	W95 FAT32	Раздел Windows 95 (от OSR2 и выше)
63	GNU HURD	Раздел ОС HURD
81	Minix	Раздел ОС Minix, используется на дискетах Linux
82	Linux swap	Раздел подкачки Linux
83	Linux	Родной (native) раздел Linux
8e	Linux LVM	Раздел для использования менеджером логических томов
a5	FreeBSD	Раздел FreeBSD и DragonFlyBSD (4.2BSD)
a6	OpenBSD	Раздел OpenBSD
a8	Darwin UFS	Раздел MacOS X
a9	NetBSD	Раздел NetBSD
fd	Linux raid auto	Раздел для использования в программных RAID

Для использования в Linux непосредственно предназначены, кроме Linux native и Linux swap, также разделы типа Minix (хотя с отмиранием дискет актуальность его теряется), Linux LVM и Linux raid auto. Кроме того, без всяких дополнительных ухищрений возможен доступ из Linux к разделам FAT любого рода. Прочие из перечисленных типов предназначены для одноименных ОС.

Присвоение разделу какого-либо идентификатора не значит, что тем самым на нем волшебным образом возникает соответствующая файловая система.

Нет, он используется только программами дисковой разметки и просто предопределяет, какого рода вторичная таблица разделов (Disk Label) может быть на нем записана.

Собственно о разделах

Итак, следствием установлено, что на одном физическом диске может быть создано до 4 (включительно) разделов, каждый их которых может быть приписан к отдельной операционной системе. А что дальше? А дальше следует изучить вопрос стилей разметки разделов.

Стили разметки разделов именуются Disk Label, что не следует путать с метками дисков (disk label) — произвольными именами, которые в DOS (и не только) можно присвоить дисковому разделу. Стиль же разметки — это формат вторичной таблицы разделов, записываемой в первый блок каждого первичного раздела. Эта таблица и определяет характер доступных действий над данным первичным разделом.

Пользователи Windows обычно не имеют причин задумываться над проблемой стилей разметки. Однако стилей таких существует немало — чтобы убедиться в этом, достаточно зайти в меню конфигурации ядра Linux, в подраздел Partition Types раздела File systems. Однако я опять забегаю вперед, нынче из всего этого изобилия нас будут интересовать только два стиля разметки — DOS и BSD.

В DOS/Windows используется (как не удивительно) DOS-стиль разметки разделов. Он основывается на BIOS-таблице, задействованной лишь частично. А именно, из четырех доступных записей Partitions Table заполняются только две (вернее, только два раздела можно создать средствами стандартного средства FDISK из DOS/Windows 96/98/ME; как обстоит дело в NT/2000/XP — просто не знаю).

В записи для первого раздела можно указать идентификатор типа файловой системы (например, FAT16 или FAT32), второму же разделу автоматически присваивается идентификатор типа Extended DOS. А уж Extended-раздел может быть далее поделен на логические разделы (Logical Partitions). В терминологии DOS/Windows они именуются логическими томами (Logical Volume). Однако мы зарезервируем этот термин для системы LVM (Logical Volumes Manager), разговор о которой пойдет в одной из ближайших интермедий.

Расширенный раздел выступает в качестве контейнера, в который последовательно, как в матрешку, вкладываются один логический раздел и еще один расширенный раздел. Последний, в свою очередь, выступает контейнером второго уровня и может включать еще один логический раздел и сле-

дующий по очереди расширенный, и так до бесконечности. Правда, аналогия с матрешкой не совсем строгая, потому что для пользователя все эти вложенные разделы видятся как равноправные части "головного" Extended-раздела. Да и насчет бесконечности тоже несколько преувеличено: на самом деле больше 63 логических разделов создать не удается.

В Linux также используется DOS-стиль разметки. Только тут уж фигурируют стандартные средства этой ОС: fdisk от Linux, имеющий весьма мало общего со своим тезкой из DOS/Windows, его front-end (оболочка) cfdisk или (почти) универсальная программа parted. Можно создать все четыре первичных раздела и пользовать их в свое удовольствие. Правда, опять же лишь одному из них можно присвоить идентификатор раздела расширенного и, соответственно, поделить на логические разделы.

Разделы в Linux представляются перед системой (и пользователем) в качестве таких же файлов блочных устройств, как и диски. И номенклатура их подчиняется тем же правилам. В старой (до devfs) нотации к обозначению целого диска (например, /dev/hda) просто добавляется численный идентификатор раздела (его порядковый номер, начиная с единицы. Идентификаторы с 1 по 4 зарезервированы за первичными разделами. Если один из них определен как расширенный, то логические разделы внутри него нумеруются, начиная с 5 (даже если первичных разделов на диске нет и в помине). То есть типичная картина разбиения диска под Linux будет выглядеть примерно так (табл. 9.5).

Таблица 9.5. *Примерная схема дисковой разметки в Linux*

Файл	Раздел
/dev/hda1	1-й первичный раздел
/dev/hda2	2-й первичный раздел
/dev/hda3	3-й первичный раздел, определенный как расширенный
/dev/hda5	1-й логический раздел внутри расширенного
/dev/hda6	2-й логический раздел внутри расширенного

При использовании devfs файлы устройств, соответствующих дисковым разделам, именуются part#, где # — порядковый номер (с 1-го по 4-й — для первичных разделов, с 5-го и далее — для расширенных). То есть в полной нотации это будет выглядеть так:

```
/dev/ide/host0/bus0/target0/lun0/part1
/dev/ide/host0/bus0/target0/lun0/part2
```

```
/dev/ide/host0/bus0/target0/lun0/part3
/dev/ide/host0/bus0/target0/lun0/part5
/dev/ide/host0/bus0/target0/lun0/part6
...
```

а в краткой — таким образом:

```
/dev/discs/disc0/part1
/dev/discs/disc0/part2
/dev/discs/disc0/part3
/dev/discs/disc0/part5
/dev/discs/disc0/part6
...
```

Интересен вопрос о разделах USB-накопителей (и встроенных носителей цифровых камер). Они поставляются в фабрично отформатированном под vfat виде, однако эта файловая система накладывается на самые разные схемы разбиения. Мне, в частности, приходилось видеть, что форматированию подвергалось все устройство, без разделов (в этом случае обращаться к флэшке из-под Linux следует как /dev/sda). В иных же случаях файловая система создавалась в 1-м или 4-м первичных разделах, файлы устройств при этом именуются как /dev/sda1 или /dev/sda4 соответственно.

Впрочем, при использовании devfs имя файла устройства, соответствующего флэшке, легко определить методом ползучего эмпиризма. Для этого достаточно просмотреть содержимое каталога /dev до и после подключения USB-драйва: файл, добавившийся во втором случае, и будет искомым именем устройства.

Особенности BSD-разметки

Совершенно иначе (как по существу, так и с точки зрения именования файлов устройств) выглядит BSD-стиль разметки (BSD Label), используемый во FreeBSD, Net- и OpenBSD. Здесь также может быть использована BIOS-таблица, заполнение которой создаст четыре первичных раздела. В терминологии FreeBSD они именуются слайсами (slices), чтобы отличать их от разделов (partitions) BSD-разметки. Слайсы в номенклатуре файлов устройств маркируются добавлением к имени файла диска литеры s и порядкового номера (в отличие от дисков, начиная с единицы), что для Master на 1 IDE-канале будет выглядеть так (табл. 9.6).

Таблица 9.6. *Номенклатура BSD-слайсов*

Файл	Раздел
/dev/ad0s1	1 слайс
/dev/ad0s2	2 слайс
/dev/ad0s3	3 слайс
/dev/ad0s4	4 слайс

Если одному или нескольким из слайсов будет присвоен идентификатор BSD-системы — 165 в десятичном исчислении (как уже говорилось, он называется 4.2BSD), то в его начальный блок можно записать собственно BSD-таблицу разделов (BSD Label). В соответствии с ее форматом, каждый слайс с ID 165 абсолютно равноправен и может быть поделен на логические разделы (собственно partitions, в терминологии FreeBSD, именуемые в остальных BSD-системах подразделами subpartitions).

Для партиций в BSD-таблице предусмотрено восемь (FreeBSD) или шестнадцать (DragonFlyBSD) записей. Соответствующие им разделы номенклатурно маркируются добавлением к имени файла слайса литеры — от a до h (или — до p в случае с DragonFly). То есть таких логических разделов, казалось бы, может быть создано 8 (или, соответственно, 16). Однако практически это не совсем так (вернее, совсем не так).

Начать с того, что одна из записей (третья по счету, маркируемая литерой c) резервируется для описания всего слайса в целом — например, ad0s1c, необходимость чего станет ясной в дальнейшем. Далее, первая запись таблицы, соответствующий файл устройства маркируется как ad#s#a, отводится для описания корневого раздела файловой системы. А очевидно, что на конкретной локальной машине корневой раздел может быть только один, вне зависимости от количества дисковых разделов и даже физических дисков.

Наконец, вторая запись (файл устройства — ad#s#b) предназначена исключительно для описания раздела подкачки (swap-раздела), который, во-первых, не может содержать данные, и во-вторых, является единственным на весь диск (ясно, что создавать по swap-разделу в каждом слайсе бессмысленно, хотя при наличии двух физических дисков поделить между ними пространство подкачки — идея вполне здоровая). В итоге на четырех слайсах физического диска средствами FreeBSD может быть создано 22 раздела: 1 корневой, один раздел подкачки и 20 разделов для хранения данных. Максимальное же количество разделов в случае с DragonFlyBSD предлагается подсчитать заинтересованным лицам.

Практически, однако, так никто, насколько я знаю, не делает. Создание слайсов преследует своей целью разместить на диске более чем одну операционную систему и сохранить возможность обмена данными между ними (теоретически к BSD-разделам можно обращаться из Linux, если пересобрать его ядро должным образом; хотя обратная процедура — обращение к Linux-разделу из BSD-системы — гораздо проще).

Если же весь наличествующий диск планируется отдать на растерзание какой-либо BSD-системе, то проще создать один-единственный слайс на (почти) весь его объем, оставив записи в BIOS-таблице для остальных неиспользованными. Ну, а семи (тем более, 15) позиций BSD-таблицы обычно достаточно для обособления всех необходимых ветвей файловой системы, таких, как /usr, /tmp, /var и /home. Впрочем, в ряде случаев целесообразно и разнесение разделов по двум слайсам, но не более.

BSD-разметка диска с записями в BIOS-таблице первого блока называется разметкой в режиме совместимости. Создается ли один слайс для BSD-системы или несколько отдельных, в режиме совместимости в начале диска резервируется пространство в размере 63 блоков (всего около 30 Кбайт), в котором не только сохраняется в неприкосновенности MBR, но и остается место для записи кода какого-либо стороннего загрузчика. В итоге диск остается доступным для других операционных систем, по крайней мере, теоретически.

Однако использование режима совместимости и BIOS-таблицы разделов в BSD не является обязательным. Вполне допустимо записать в MBR вместо таблицы BIOS непосредственно BSD-таблицу разделов. В этом случае понятно, что слайсов как таковых не создается, а все дисковое пространство представляет собой как бы единый слайс и может быть разбито на BSD-партиции по тем же правилам, что и отдельный слайс. И тут становится ясной необходимость резервирования третьего поля BSD-таблицы — именно в нем и описывается весь наш диск, целиком отведенный под BSD-систему.

Такое обращение с диском именуется режимом эксклюзивного использования, или Dangerously Dedicated. Вопреки названию, в нем не таится никакой опасности ни для данных пользователя, ни для его здоровья. А единственная подстерегающая его опасность — это то, что диск в эксклюзивном режиме не будет опознан никакой другой операционной системой, установленной на данном компьютере (обращению к диску по сети он препятствий не составит). Однако это чисто теоретическое неудобство, потому что ни одна из известных мне ОС все равно не умеет толком работать с BSD-разделами и файловой системой FreeBSD (особенно современной — UFS2). А, скажем, при наличии на другом физическом диске мультизагрузчика GRUB, FreeBSD с "эксклюзивного" диска вполне может быть им загружена.

В документации по FreeBSD встречаются указания, что "эксклюзивные" диски иногда не могут быть загрузочными, вероятно, потому, что BIOS не сможет опознать нестандартные записи в MBR. Однако, видимо, это относится к каким-либо старым версиям BIOS — мне с таким сталкиваться не приходилось, хотя я часто прибегал к эксклюзивному режиму при возможности отдать под FreeBSD целый физический диск.

Тем не менее, в документах проекта FreeBSD всегда подчеркивается, что эксклюзивный режим, в частности, из-за грошовой экономии дискового пространства следует использовать лишь в исключительных случаях. Один из резонов к такому использованию — несоответствие "геометрии" диска, видимой из BIOS, и того представления о ней, которое складывается у FreeBSD (подробнее на эту тему можно прочитать в официальном FreeBSD FAQ на **http://www.freebsd.org.ua/doc/ru_RU.KOI8-R/books/faq/index.html**).

Схема разметки диска в BSD-стиле принята и в других ОС этого семейства, с той разницей, что термин "слайс" ни в OpenBSD, ни в NetBSD не применяется, и имена файлов разделов именуются — /dev/wd0a, /dev/wd0b и т. д. А в NetBSD, кроме литеры c, предназначенной для описания всего диска, резервируется еще и литера d — файл устройства /dev/wd0d также предназначается для описания первичного раздела целиком.

RAID и LVM

Дисковые разделы любого рода создаются в целях не только размежевания, но и объединения, в результате которого части различных физических носителей предстают перед системой в качестве непрерывного дискового пространства, на котором может быть создана единая файловая система. И целям объединения служат разнообразные технологии программных RAID и LVM.

Понятие RAID родилось из потребности представить набор дисков в виде единого дискового пространства, и первоначально было реализовано посредством специализированных контроллеров с интерфейсами SCSI или ATA (ныне также и Serial ATA). Это так называемые физические RAID. Конечно, в них за объединение дисков отвечают соответствующие программы, однако они прошиты в микросхемах контроллера, расположенного либо на отдельной карте расширения, либо, как это стало обычным в последнее время, встроенного в материнскую плату.

Объединение дисков и их разделов возможно и при отсутствии специфического "железа", именно такая реализация называется программным RAID (Soft RAID). Оно требует поддержки ядра операционной системы и наличия соответствующего программного инструментария.

Вне зависимости от того, реализован он аппаратно или программно, RAID-массив предназначен для представления нескольких физических дисков (или, при программном подходе, также и их разделов) в виде единого устройства. Правда, дальше речь пойдет только о программных реализациях дисковых массивов. Различают RAID-массивы с избыточностью и без оной.

К массивам без избыточности относятся линейный режим (linear RAID) и RAID level 0. Линейный режим — просто последовательное объединение двух и более дисков, так, что они видятся системой как единое устройство. И запись на них осуществляется последовательно: сначала на первый диск, по заполнении его — на второй, и т. д. Диски или разделы, объединяемые в линейный RAID, могут быть произвольного объема, на их количество также не накладывается никаких ограничений.

Создаются линейные массивы исключительно из соображений удобства, если есть потребность в непрерывном дисковом пространстве, превышающем объем наличных дисков или их разделов: они не дают ни повышения быстродействия, ни дополнительной надежности хранения данных. Впрочем, и к снижению надежности они тоже не приводят: при разрушении одного диска или раздела данные со второго остаются доступными точно также, как и при использовании самостоятельных носителей.

Массивы нулевого уровня называются также RAID со strip-режимом, это попарное объединение дисков с распараллеливанием операций чтения/записи. В них каждая порция записываемых данных расслаивается на два равных по размеру логических блока (chunks), один из которых (в одну и ту же единицу времени) записывается на первый диск, другой — на второй. Что теоретически должно способствовать быстродействию дисковых операций. И способствует практически, но только при условии, что диски массива разнесены на разные IDE-каналы (о SCSI-дисках у нас тоже речи не будет). В противном случае выигрыша в производительности не только не будет, но весьма вероятно даже ее падение.

Очевидно, что в массив нулевого уровня можно объединить только четное количество дисков или разделов. Кроме того, желательно, чтобы они имели примерно одинаковый объем: в противном случае суммарный объем массива будет кратен размеру самого маленького диска, а все избыточное пространство окажется неиспользуемым.

Использование массивов нулевого уровня не способствует надежности хранения данных, при отказе хотя бы одного диска пропадут они все (тогда как в линейном режиме информация на исправном диске в первом приближении сохранится). Тем не менее, применение параллельного режима может быть оправданным.

Массивы с избыточностью призваны, в первую очередь, повышать надежность хранения данных. А если при этом еще и возрастает быстродействие

дисковых операций, это можно рассматривать как бесплатный бонус. На практике в Linux и BSD используются два уровня избыточных RAID: 1 и 5.

RAID level 1 (называемый также режимом зеркалирования, mirroring) — это простое дублирование записываемых данных на двух дисках или разделах, такой массив обладает стопроцентной избыточностью: при выходе из строя одного диска вся информация сохраняется на диске-дублере (хотя в процессе работы они выступают как равноправные). Очевидно, что, как и в случае с RAID level 0, число дисков или разделов должно быть четным, а их объем примерно одинаков.

Разумеется, ни о каком росте производительности в массивах первого уровня речи идти не может. Однако на сей предмет level 1 можно комбинировать с level 0 (это называют level 0+1, или иногда level 10), когда одна пара дисков в параллельном режиме зеркалируется второй парой. Правда, как нетрудно подсчитать, для этого желательно иметь не только четыре диска, но и соответствующее число отдельных IDE-контроллеров, иначе роста производительности ожидать не придется.

В RAID level 5 данные распределяются по всем составляющим массив дискам (или разделам) и дополняются контрольными суммами. По последним и осуществляется восстановление информации в случае отказа одного из устройств. Минимальное количество дисков (разделов) в таком массиве три, размер их должен быть примерно равным, и общий объем массива равен произведению объема наименьшего и их числа минус единица, так как для хранения контрольных используется часть пространства от каждого диска, в сумме равного объему единичного устройства. Массивы 5 уровня считаются весьма надежными и даже обещают некоторый прирост быстродействия.

Создание и использование программных RAID несколько различается в Linux и BSD-системах. И в обоих случаях требуют некоторых не вполне тривиальных действий. Хотя после создания массива с ними следует обращаться, как с обычными дисковыми разделами: создавать на них файловые системы и монтировать их. Впрочем, об этом разговор будет в *интермедии* после *главы 10*.

Программный RAID-массив — лобовой подход к проблеме объединения дисков. Попробуем подойти к ней с другого бока, благо, это допускается в виде технологии логических томов (LVM); правда, доступна она только в Linux.

В чем недостаток лобового подхода? Представим простую жизненную ситуацию. Приблизившись к исчерпанию доступного дискового пространства для собственных данных, пользователь, как правило, приобретает новый жесткий диск и добавляет его в систему. Однако физически ввинтить винчестер в корпус машины — это полдела, нужно еще и сделать его для системы доступным. Как? Возможны варианты.

Можно создать на диске физический раздел, на нем — новую файловую систему и смонтировать ее к файловой системе корневой в заранее созданный каталог, например, $HOME/newhome. Это проще всего, но не всегда лучше всего. Ведь в итоге единый пользовательский каталог будет состоять из двух физически различных файловых систем, что накладывает некоторые ограничения на манипулирование данными (например, невозможно создать жесткие ссылки на наборы данных другой файловой системы).

И вообще, это не очень удобно, все время помнить, какие данные находятся на одном физическом диске, какие — на другом. Гораздо проще, если весь пользовательский каталог видится как единое дисковое пространство, пусть даже таковым и не является.

Второй пример не менее жизненный. Послушавшись совета резонных людей, мы делим диск на многочисленные разделы, как это будет описано в следующей главе, для всех возможных файловых систем — под каталоги /usr, /usr/local, /opt, /var, /tmp, /home. И очень быстро убеждаемся, что под /usr/local отвели слишком много места, весь нужный нам софт по умолчанию желает собираться в каталоге /opt (с объемом которого мы явно пожадничали). А каталог /usr, вследствие изобилия Иксовых приложений, напротив, заполняется чересчур уж стремительно, что отнюдь не есть хорошо: производительность дисковых операций в любом UNIX стремительно падает по достижении определенного предела заполнения файловой системы. Ну, а каталог /tmp просто бездействует, роль хранителя временных файлов, как оказалось, с успехом исполняет файловая система в оперативной памяти tmpfs. И что, спрашивается, делать? Вариант тотальной архивации и переразбиения дисков просьба не предлагать.

Обеих ситуаций можно избежать, если предугадать такие коллизии заранее. И еще на стадии разбиения диска прибегнуть к технологии LVM (Logical Volume Manager), то есть управления логическими томами.

Замечание о терминологии

Пользователи DOS/Windows знают, что логическими томами (logical volumes) называются те части, на которые разбивается расширенный (extended) раздел DOS. Однако в случае LVM в понятие логического тома вкладывается, как станет ясно очень скоро, совершенно иной смысл. И потому в этой книге я именую части расширенного DOS-раздела незамысловато — логическими разделами (впрочем, примерно такая терминология принята в Linux-программах разбиения диска), а звание логического тома сохраняется только за элементом структуры LVM.

Теперь можно перейти к сути дела. Основа технологии LVM — в выделении двух уровней организации дискового пространства, физического и логического. Единица физической организации — физический том (Physical Volume). Это не что иное, как самый обычный дисковый раздел с определенным идентификатором типа файловой системы 8e (Linux LVM), который

присваивается разделу при его создании, например, программой fdisk. Физический том делится на физические же блоки (physical extents) — кванты дискового пространства (своего рода аналог обычного физического блока винчестера).

Единицей организации логической в LVM выступает группа томов (Volume Group). Она сливает воедино физические тома так, что для ОС они выглядят как единый носитель, и, следовательно, может представляться как нечто вроде логического аналога винчестера. Как и винчестер, группа томов может делиться на разделы (или, напротив, представлять собой единый раздел). Один такой раздел и есть логический том, на котором создаются обычные файловые системы и к которому непосредственно обращается пользователь, например, при монтировании. Он образован последовательностью логических блоков (logical extents), которые можно сопоставить с логическими блоками обычного раздела, размечаемыми при создании файловой системы.

Взаимосвязь физических и логических extents обозначается термином mapping. Причем возможны два варианта такой взаимосвязи: линейный (linear mapping) и чередующийся (striped mapping). В первом случае непрерывной последовательности физических extents просто ставится в соответствие столь же непрерывная последовательность extents логических. Во втором же — непрерывная последовательность логических extents соотнесена с физическими extents, чередующимися между физическими носителями (рис. 9.1).

```
.+-- Volume Group -------------------------------+
|                                                 |
|    +----------------------------------------+   |
| PV | PE |  PE | PE | PE | PE | PE | PE | PE |   |
|    +----------------------------------------+   |
|      .          .          .        .           |
|      .          .          .        .           |
|    +----------------------------------------+   |
| LV | LE |  LE | LE | LE | LE | LE | LE | LE |   |
|    +----------------------------------------+   |
|           .          .        .         .       |
|           .          .        .         .       |
|    +----------------------------------------+   |
| PV | PE |  PE | PE | PE | PE | PE | PE | PE |   |
|    +----------------------------------------+   |
|                                                 |
```

Рис. 9.1. Последовательность extents

Эта схема подобна той, что осуществляется в описанных ранее программных RAID-массивах нулевого (strip) уровня и преследует ту же цель — повышение производительности дисковых операций.

Из сказанного ясно, что технология LVM наиболее эффективна в случае, если в машине имеется более одного физического диска. Однако и в однодисковой системе ее можно использовать с большим успехом.

Создание логических томов требует поддержки ядром Linux и применения некоторого специфического инструментария, описанного в *интермедии* после *главы 10*.

Общие черты файловых систем POSIX-семейства

Итак, файловые системы обычно размещаются в отдельных дисковых разделах или их объединениях. Соотношение между файловыми системами не вполне однозначно. В одном разделе может быть размещена одна файловая система, в общем случае, она может располагаться более, чем в одном разделе. Такие технологии, как LVM (Logical Volumes Managers) или программный RAID, позволяют объединить несколько дисковых разделов в единое логическое устройство, которое целиком занимается одной файловой системой. Впрочем, после такого объединения (конкатенации) прежде самостоятельные разделы видятся системой точно так же, как и обычные разделы или слайсы диска.

Физически все файловые системы в ОС, соответствующих стандарту POSIX, организованы сходным образом. И в основе их лежит разделение на три пространственно разобщенные области: суперблок, область метаданных и блоки данных.

- Суперблок — это первый (после загрузочного) блок раздела, занятого данной файловой системой. Он содержит наиболее общую информацию о ней: размер логического блока (который следует отличать от физического дискового блока), их суммарное количество, число блоков данных, количество свободных и занятых блоков. Кроме того, в нем записывается и информация, специфичная для определенных файловых систем.
- Содержимое области метаданных составляет таблица inodes данных файловой системы. Для каждого файла она содержит его уникальный идентификатор, число ссылок на него (фигурально говоря, количество имен файла), размер, атрибуты принадлежности, доступа, времени и еще некоторые, о чем говорилось в *главе 8*. Адреса блоков данных файла, то есть

собственно физического расположения его контента на диске, также описываются в таблице inodes.

- Ну, а сами блоки данных, как легко догадаться, и составляют наполнение одноименной области.

Описанным способом была организована первая файловая система первозданного UNIX, получившая позднее название s5fs (то есть файловая система System V). Как уже говорилось в историческом обзоре (*см. главу 3*), в самой ее природе были заложены существенные ограничения, не способствовавшие ни надежности, ни быстродействию, ни эффективности использования дискового пространства. И потому разработчики всех более поздних файловых систем POSIX-совместимых ОС, приняв за основу организацию s5fs, вносили в нее те или иные усовершенствования, направленные на исправление указанных недостатков.

Борьба за повышение быстродействия файловых операций осуществлялась дроблением файловой системы на более или менее независимые части, получившие название групп цилиндров (в ОС BSD-семейства), групп блоков (в файловой системе Linux), вплоть до обособления allocation groups в файловой системе XFS, где они представляют фактически самостоятельные файловые подсистемы. В результате этого появилась возможность размещения логически связанной информации в физически смежных областях диска, что минимизировало перемещение дисковых головок, способствуя тем самым скорости доступа к данным.

Расчленение файловой системы одновременно способствовало и росту их надежности. Так как давало возможность в разных ее частях дублировать критически важную информацию, а именно: суперблок, утрата которого (вследствие, например, физического повреждения дисковой поверхности) ранее влекла за собой невозможность доступа к данным вообще.

Экономия в использовании дискового пространства достигалась прямо противоположным приемом — так называемой внутренней фрагментацией, то есть разделением на части логического блока файловой системы. И теперь маленькие файлы могли занимать уже не блок целиком, а только его часть. Венцом творения в этой области явилась файловая система ReiserFS, в которой даже данные маленьких файлов удается целиком вписать в область ее метаданных.

Еще одна общая особенность всех файловых систем UNIX-подобных ОС — это кэширование дисковых операций, как чтения, так и записи. Строго говоря, само по себе кэширование к устройству файловой системы имеет немного отношения, поддерживаясь на уровне ядра, а включаясь (или отключаясь) при монтировании. Однако оно очень важно с точки зрения

восприятия файловых операций пользователем, и потому здесь уместно сказать о нем несколько слов.

С кэшированием при чтении более-менее ясно — это упреждающее считывание групп блоков файловой системы и помещение их в специальный буфер в оперативной памяти. Кэширование же при записи выражено в том, что любые изменения данных файлов (например, при сохранении текста в текстовом редакторе), их метаданных (например, счетчика ссылок или атрибутов), а также записей в каталоге при создании или удалении новых файлов не переносятся на диск немедленно, а помещаются в буфер оперативной памяти. Операции же записи изменений на диск производятся через некоторое время.

Существуют два режима кэширования файловых операций: частично синхронный и чисто асинхронный. При первом изменения в метаданных записываются на диск немедленно, а отложенная запись применяется только в отношении измененных данных. При чисто асинхронном режиме кэшируются как данные, так и метаданные.

Как уже было отмечено, режим кэширования определяется при монтировании файловых систем (*об этом пойдет речь в следующей главе*). И любая из файловых систем Linux или BSD может быть смонтирована как в том, так и в другом режиме. Однако файловые системы BSD традиционно используются в частично синхронном режиме, а все файловые системы, поддерживаемые в Linux, — в асинхронном.

Кэширование дает просто потрясающий эффект с точки зрения быстродействия файловых операций, особенно выраженный при асинхронном режиме. Расплатой за что становится возможное нарушение целостности файловых систем при аппаратных сбоях, или некорректном завершении работы. Ведь в этом случае буферизованные изменения данных и, что хуже, метаданных теряются.

В BSD-системах в качестве метода борьбы с этими явлениями была разработка матода SoftUpdates (о котором будет сказано чуть позже). В Linux же основным направлением в повышении надежности файловых операций явилось развитие так называемых журналируемых файловых систем. Журнал — это часть файловой системы, отведенная под протоколирование выполненных и предстоящих файловых операций, а также под сохранение данных о текущем (в некие моменты времени) ее состоянии. Что, в случае аварийного завершения работы, нарушившего целостность данных и (в первую очередь) метаданных, дает возможность откатиться назад, в последнее "нормальное" состояние.

Основные типы файловых систем POSIX-мира

Таковы общие особенности файловых систем UNIX и его производных. Теперь охарактеризуем наиболее используемые в свободных ОС POSIX-семейства их разновидности.

Файловая система BSD-семейства

Для файловой системы BSD-систем, начиная с 4.2BSD, используется общее родовое наименование FFS (Fast File System). Название это отражает их большее быстродействие относительно s5fs, хотя среди современных файловых систем они в числе самых медленных. Реализация FFS для FreeBSD и DragonFlyBSD носит название UFS (UNIX File System). Во FreeBSD, начиная с пятой ветки, используется усовершенствованная, 64-разрядная разновидность последней — UFS2. Она с некоторых пор поддерживается также в NetBSD и DragonFly, хотя и не является в них используемой по умолчанию. Фактически FFS и разновидности UFS — единственные нативные файловые системы для ОС BSD-семейства, хотя на уровне обмена данными в них можно использовать файловые системы FAT-семейства и, с оговорками, некоторые типы файловых систем Linux.

Как уже говорилось ранее, дисковый раздел, на котором создается файловая система — это, говоря метафорически, группа смежных цилиндров, разбитых на физические блоки по 512 байтов. Однако специфика файловых систем BSD-клана заключается в том, что при их создании раздел делится еще и на части фиксированного объема (зависящего от объема раздела). Это так называемые группы цилиндров, каждая из которых включает три самостоятельные части:

- суперблок, одинаковый по содержанию для всех групп;
- блок группы цилиндров, ее описывающий; одной из важнейших его частей является индексная таблица (или таблица inodes);
- область блоков данных.

Рассмотрение целесообразно начать с конца списка, то есть с области данных.

Область данных (она занимает подавляющий объем пространства каждой группы цилиндров) образована блоками файловой системы. Подобно физическому блоку, о котором речь была ранее, блок файловой системы (именуемый также блоком логическим) представляет собой квант информации, доступный за одну операцию чтения/записи, но уже не дисковой головки, а операционной системы. Очевидно, что логический блок не может

быть меньше блока физического (то есть 512 байтов), и размер его обязательно кратен целому числу последних.

Понятие логического блока введено для повышения производительности дисковых операций. Не требует доказательства утверждение, что скорость обмена данными квантами по 1 Кбайт будет выше, чем 512-байтными, 2 Кбайт — еще быстрее и т. д. И потому, с точки зрения быстродействия файловых операций, выгоден максимальный размер логического блока файловой системы.

С другой стороны, увеличение размера логического блока ведет к непроизводительному расходу дискового пространства: данные файлов, размер которых меньше блока файловой системы, все равно занимают его целиком. Также целый блок потребуется и для хранения файловых хвостов, то есть остатков сверх объема, кратного размеру логического блока. Для борьбы с этим в UFS введено понятие фрагмента — логической части блока файловой системы, которая может быть записана или считана независимо от остальной его части.

Ясно, что размер фрагмента все равно не может быть меньше физического блока. При этом UFS накладывает на него и встречное ограничение: минимальный размер фрагмента определяется в 1/8 логического блока. Другие же возможные значения — 1/4 и 1/2 блока файловой системы (очевидно, что выделение фрагмента в размере блока равносильно отказу от фрагментации блока вообще, хотя это, как будто бы, не запрещено).

Каким образом определяется принадлежность блока данных тому или иному файлу? С помощью индексной таблицы, именуемой также таблицей индексных дескрипторов, или таблицей inodes. Она образована некоторым (конечным) количеством записей фиксированной длины (128 байтов в UFS, 256 — в UFS2), каждая из которых однозначно соответствует одному файлу, как реально существующему, так и только могущему быть созданным.

Такая запись индексной таблицы носит название inode. Каждый содержит так называемые метаданные файла. Для реально существующего файла они включают в себя идентификатор, тип файла и еще кое-какие атрибуты, рассмотренные ранее. Для файлов несуществующих свободные записи в таблице inodes просто резервируются. А поскольку количество этих записей — величина конечная, именно объем индексной таблицы и определяет, сколько файлов реально может быть создано в данной файловой системе. Исчерпание свободных записей в ней приводит к тому, что, вне зависимости от объема свободного дискового пространства, ни одного нового файла создать не удастся.

Таблицы inodes содержатся в блоке группы цилиндров наряду с картами свободных/занятых inodes и блоков данных. Однако сведений об объеме ин-

дексной таблицы в блоке группы цилиндров не обнаруживается. Место их размещения — тот самый суперблок, который идет первым в любом разделе (вообще-то первым будет загрузочный блок, но реально он занят только в загрузочном же разделе — во всех остальных для него просто зарезервировано место). А также дублируется в каждой группе цилиндров, чем обеспечивается устойчивость к механическим повреждениям диска. Кроме этого, в суперблоке же записывается размер блока файловой системы и их суммарное количество, число блоков данных, размер блокового фрагмента и их число в блоке, число блоков, реально занятых файлами, объем партиции, перманентно резервируемый как свободный, и еще множество характеристик файловой системы.

При всех своих многочисленных достоинствах файловые системы BSD не могут похвастаться одним — быстродействием. Причина в частично синхронном ее режиме, принятом по умолчанию. Конечно, UFS может быть смонтирована и асинхронно, но в этом случае резко падает ее устойчивость к сбоям при относительно небольшом выигрыше в производительности. И при этом она, не будучи журналируемой, сама по себе не имеет механизма гарантии собственной целостности.

Исправлению обоих недостатков служит механизм SoftUpdates. Он детально описан в специальной статье Макказика и Ганджера (**http://www.osp.ru/os/1999/07-08/13.htm**). В двух же словах суть этого механизма в сведении к минимуму синхронных операций записи без явно асинхронного манипулирования метаданными, с одной стороны, но и без предварительного журналирования метаданных (как в файловых системах типа ReiserFS или XFS) — с другой.

Реализуется это за счет так называемых зависимостей обновления. Что такое эти зависимости, интуитивно ясно из примера создания нового (для простоты, пустого) файла. Для этого требуется выполнение следующих действий:

1) запись в таблице inodes, с заполнением полей типа файла, его идентификатора, счетчика ссылок, прав доступа в соответствии с маской и т. д.;
2) изменение карты свободных/занятых inodes в блоке группы цилиндров (соответствующий новому файлу inode должен быть помечен в ней битом занятости);
3) внесение записи вида "идентификатор — имя файла" в структуру каталога, в котором новый файл создается, что обеспечивает ту самую единственную ссылку в соответствующем поле inode файла.

С точки зрения целостности файловой системы, эти операции должны быть выполнены именно в указанной последовательности. То есть наличие в каталоге имени файла с идентификатором незаполненного (еще не созданного или уже уничтоженного) inode — явный непорядок: для исправления такого

рода безобразий и запускается программа проверки диска после аварийного завершения сеанса.

Выполнять связанные зависимостями операции обновления в синхронном режиме долго (каждая потребует своего обращения к диску), в чисто асинхронном нет гарантии сохранения последовательности (обновления в кэше могут быть записаны тогда, когда бог на душу положит). Так вот, механизм SoftUpdates обеспечивает, с одной стороны, контроль последовательности выполнения зависимых обновлений (что способствует целостности состояния файловой системы), с другой группирует их в единую атомарную операцию синхронного обращения к диску, за счет сокращения числа коих и растет производительность. В этом и состоит объяснение парадокса SoftUpdates — неожиданного увеличения и надежности, и быстродействия.

Файловые системы Linux

В Linux длительное время также имелась только одна нативная файловая система, носящая имя Ext2fs. О ней написано немало, поэтому буду краток. Название ее расшифровывается как "вторая расширенная файловая система"; "расширенная" она — по сравнению с файловой системой ОС Minix, послужившей прототипом Linux (и до сих пор используемой на отформатированных в этой ОС дискетах), "вторая" потому что ранние версии Linux базировались на Extfs с более ограниченными возможностями.

По способу организации хранения данных она типичная представительница файловых систем UNIX. Отличительная ее особенность — дробление дискового раздела на группы блоков и наличие нескольких копий суперблока, что повышает надежность хранения данных. Кроме того, для нее характерен очень эффективный механизм кэширования дисковых операций, что обеспечивает замечательное их быстродействие, едва ли не рекордное среди всех известных файловых систем. Оборотная сторона — относительно слабая устойчивость при аварийном завершении работы (вследствие мертвого зависания или отказа питания), поскольку отложенность записи изменений файлов делает весьма высокой вероятность нарушения связи между их inodes и блоками данных.

Конечно, времена, когда некорректный останов Linux-машины грозил полным разрушением файловой системы, остались в далеком прошлом. Однако в любом случае останов системы без штатного размонтирования файловых систем приводит к тому, что в них не устанавливается упоминавшийся ранее неоднократно "бит чистого размонтирования". А без этого утилиты обслуживания диска (такие как программа проверки fsck) при перезагрузке не воспринимают их как целостные и начинают проверку, которая при современных объемах дисков может занять немалое время.

Проблема нарушения целостности файловой системы при некорректном завершении работы в большей или меньшей мере характерна для всех ОС семейства UNIX. И потому с давних пор в них разрабатываются журналируемые файловые системы. Журнал — это нечто вроде log-файла дисковых операций, в котором фиксируются не выполненные, а только предстоящие манипуляции с файлами, вследствие чего оказывается возможным самовосстановление целостности файловой системы после сбоя.

Во избежание недоразумений следует подчеркнуть, что журналирование направлено на обеспечение целостности файловой системы, но ни в коем случае не гарантирует сохранность пользовательских данных как таковых. Так, не следует ожидать, что журналирование волшебным образом восстановит не сохраненные перед сбоем изменения документа, загруженного в текстовый редактор.

Более того, в большинстве журналируемых файловых систем фиксируются грядущие операции только над метаданными изменяемых файлов. Обычно этого достаточно для сохранения целостности файловой системы и, уж во всяком случае, предотвращения долговременных их проверок, однако не предотвращает потери данных в аварийных ситуациях. В некоторых из файловых систем возможно распространение журналирования и на область данных файла. Однако, как всегда, повышение надежности за счет этого оплачивается снижением быстродействия.

Текущие версии ядра Linux поддерживают в качестве нативных четыре журналируемые файловые системы: ReiserFS и Ext3fs, специфичные для этой ОС, XFS и JFS — результаты портирования в Linux файловых систем, разработанных первоначально для рабочих станций под ОС Irix (SGI) и AIX (IBM) соответственно. Правда, широкое признание получили только три первых, так что о JFS я говорить не буду.

Файловая система ReiserFS оказалась для Linux исторически первой из журналируемых, она поддерживается каноническим ядром с **http://www.kernel.org**, начиная с первых версий ветви 2.4.x. И была единственной, разработанной "с нуля" специально для этой ОС Хансом Райзером и его фирмой Namesys (**http://www.namesys.com**). В ReiserFS осуществляется журналирование только операций над метаданными файлов. Что, при определенном снижении надежности, обеспечивает высокую производительность: на большинстве типичных пользовательских задач она лишь незначительно уступает Ext2fs. А на такой, достаточно обычной, операции, как копирование большого количества мелких файлов, существенно ее опережает.

Кроме этого, ReiserFS обладает уникальной (и по умолчанию задействованной) возможностью оптимизации дискового пространства, занимаемого

мелкими, менее одного блока, файлами (а следует помнить, что в любой UNIX-системе такие файлы присутствуют в изобилии): они целиком хранятся в своих inode, без выделения блоков в области данных: вместе с экономией места это способствует и росту производительности, так как и данные, и метаданные (в терминах ReiserFS, stat-data) файла хранятся в непосредственной близости и могут быть считаны одной операцией ввода/вывода.

Вторая особенность ReiserFS — то, что хвосты файлов (их конечные части), меньшие по размеру, чем один блок, могут быть подвергнуты упаковке. Этот режим (tailing) также включается по умолчанию при создании ReiserFS, обеспечивая около 5% экономии дискового пространства. Что, правда, несколько снижает быстродействие, и потому режим тайлинга можно отменить при монтировании файловой системы. Однако упаковка хвостов автоматически восстанавливается на файловой системе, в которую было инсталлировано новое ядро, что по некоторым причинам требует внимательного отношения.

ReiserFS не совместима с Ext2fs на уровне утилит обслуживания файловой системы. Однако соответствующий инструментарий, объединенный в пакет reiserfsprogs, уже давно включается в штатный комплект современных дистрибутивов (или, в крайнем случае, может быть получен с сайта Namesys, **http://www.namesys.com**).

Более серьезная проблема с совместимостью в том, что распространенные загрузчики Linux (и Lilo, и GRUB, хотя и по разным причинам) иногда не способны загрузить ядро Linux с раздела ReiserFS, оптимизированного в режиме тайлинга. А поскольку, будучи отключен, этот режим обладает свойством самовосстановления, пользователь может столкнуться с тем, что после пересборки ядра система просто откажется загружаться. Так что ReiserFS не рекомендуется к употреблению в загрузочном разделе.

В настоящее время разработана новая версия этой файловой системы — Reiser4. Собственно, переработка ее была столь кардинальна, что выходит за рамки версионности. Однако на момент написания этих строк Reiser4 не поддерживается каноническим ядром Linux (требуются патчи от Эндрю Мортона). Нет ее поддержки и в подавляющем большинстве распространенных дистрибутивов, так что ничего определенного я о ней сказать не могу.

В отличие от ReiserFS, Ext3fs не более, чем журналируемая надстройка над классической Ext2fs, разработанная Стивеном Твиди в компании Red Hat и поддерживаемая ядром Linux, начиная с версии 2.4.16. Как следствие такого происхождения, она сохраняет со своей прародительницей полную совместимость, в том числе и на уровне утилит обслуживания (начиная с версии

1.21 объединяющего их пакета e2fsprogs). И переход от Ext2fs к Ext3fs может быть осуществлен простым добавлением файла журнала к первой, не только без переформатирования раздела, но даже и без рестарта машины.

Из этого вытекает первое преимущество Ext3fs, особенно весомое в случае большого парка компьютеров. Второе же — чуть ли не максимальная надежность: Ext3fs является единственной системой из рассматриваемых, в которой возможно журналирование операций не только с метаданными, но и с данными файлов. Ибо в Ext3fs предусмотрено три режима работы: полное журналирование (full data journaling), журналирование с обратной записью (writeback), а также задействуемое по умолчанию последовательное журналирование (ordered).

Режим полного журналирования, как легко догадаться из названия, распространяется и на метаданные, и на данные файлов. Все их изменения сначала пишутся в файл журнала и только после этого фиксируются на диске. В случае аварийного отказа журнал можно повторно перечитать, приведя данные и метаданные в непротиворечивое состояние. Этот механизм практически гарантирует от потерь данных, однако является наиболее медленным.

В режиме отложенной записи, напротив, в файл журнала записываются только изменения метаданных файлов, подобно всем рассмотренным далее файловым системам. То есть никакой гарантии сохранности данных он не предоставляет, однако теоретически обеспечивает наибольшее (в рамках Ext3fs) быстродействие.

В последовательном режиме также физически журналируются только метаданные файлов, однако связанные с ними блоки данных логически группируются в единый модуль, называемый транзакцией (transaction). И эти блоки записываются перед записью на диск новых метаданных, что, хотя и не гарантирует сохранности самих данных, весьма последней способствует. Причем при меньших накладных расходах по сравнению с полным журналированием, обеспечивая промежуточный уровень быстродействия. Опять же, в теории, на практике все может быть не совсем так.

Файловая система XFS, в отличие от молодых ReiserFS и ext3fs, развивается фирмой SGI для ее собственного варианта UNIX на протяжении более десятка лет: впервые она появилась в версии Irix 5.3, вышедшей в 1994 г. Но в Linux она была портирована лишь недавно (текущая ее версия свободно доступна с SGI's XFS page, **http://oss.sgi.com/projects/xfs**) и штатно поддерживается ядром, лишь начиная с его ветки 2.6.X.

XFS — единственная 64-разрядная файловая система из используемых в Linux. Однако уникальность ее не только в этом.

Особенностями XFS являются:

- механизм allocation group, то есть деление единого дискового раздела на несколько равных областей, имеющих собственные списки inodes и свободных блоков, для распараллеливания дисковых операций; в сущности, это почти самостоятельные файловые субсистемы; логическое журналирование только изменений метаданных, но с частым сбросом их на диск для минимизации возможных потерь при сбоях;
- механизм delayed allocation — ассигнование дискового пространства при записи файлов не во время журналирования, а при фактическом сбросе их на диск, что, вместе с повышением производительности, предотвращает фрагментацию дискового раздела;
- списки контроля доступа (ACL, Access Control List) и расширенные атрибуты файлов (extended attributes), рассмотрение которых далеко выходит за рамки нынешней темы.

В результате XFS предстает как очень сбалансированная файловая система, ориентированная на размещение в очень больших дисковых разделах для работы с очень большими файлами.

И, наконец, JFS (разработка IBM для собственной версии UNIX, AIX) уже долгое время поддерживается ядром Linux в качестве нативной. Однако по ряду причин широкого распространения здесь она не получила. Так что за отсутствием собственного опыта от ее характеристики воздержусь.

На уровне обмена данными и Linux, и BSD поддерживают множество файловых систем, некоторые только в режиме чтения, другие — и записи тоже. Важнейшими для пользователя являются файловые системы CD- и DVD-дисков (ISO9660 и UDF соответственно) и вариации на тему FAT, а также NTFS (последняя практически только в режиме чтения).

Кроме того, Linux и BSD теоретически поддерживают файловые системы друг друга. Теоретически потому, что прочитать из Linux раздел с UFS или UFS2 — задача нетривиальная, запись же на него, как пишут в документации, — дело рискованное для сохранности данных. Во FreeBSD и DragonFlyBSD же поддерживается чтение и запись для файловых систем Ext2fs и Ext3fs без особых сложностей (по крайней мере, я с проблемами здесь не сталкивался). Ведутся работы по восприятию ReiserFS и XFS (пока только в режиме чтения), существуют и патчи для доступа к JFS.

Все упомянутые выше файловые системы могут располагаться не только на реальных блочных устройствах (дисках и дисковых разделах), но и на виртуальных RAM-дисках. Впрочем, кроме того, что они располагаются в оперативной памяти (занимая некий фиксированный ее участок), в остальном ведут себя точно так же, как и обычные дисковые разделы.

Виртуальные файловые системы

До сих пор речь шла о файловых системах, размещаемых на блочных устройствах (block devices based). Однако POSIX-совместимые ОС поддерживают еще и несколько типов виртуальных файловых систем. Они не занимают места на диске (или почти не занимают), располагаясь в оперативной памяти, и служат для специальных целей.

Исторически первой из таких виртуальных файловых систем была файловая система процессов procfs. Как можно догадаться из названия, она представляет протекающие в системе процессы в виде файлов каталога /proc, откуда и получают информацию о процессах команды типа `ps` и `top`. Однако и пользователь может извлечь оттуда немало полезной для себя информации.

В каталоге /proc каждому процессу соответствует подкаталог, именем которого является идентификатор процесса (просто номер в порядке запуска). Внутри такого подкаталога обнаруживается набор файлов (разный, в зависимости от ОС). Содержимое некоторых из них можно посмотреть обычным способом и обнаружить там информацию об имени запустившей процесс команды, используемых ею адресах памяти и т. д.

Множество важной информации содержится в файлах корня /proc. Из них обычными командами просмотра можно извлечь сведения о процессоре (/proc/cpuinfo), текущей конфигурации ядра системы (/proc/config.gz), загруженных его модулях (/proc/modules), устройствах, подсоединенных к шине PCI (/proc/pci), в частности, иногда только тут можно узнать подробности о своей сетевой карте и внутреннем модеме, необходимые для правильной их настройки.

Следующей виртуальной файловой системой является файловая система устройств — devfs. Она по умолчанию задействована во FreeBSD пятой ветки и во многих (хотя и далеко не всех) современных дистрибутивах Linux. Чтобы понять ее назначение, нужно вспомнить, как происходило обращение с файлами устройств ранее.

А происходило оно так. Некий набор файлов устройств генерировался при начальной установке системы. Каждый файл устройства характеризовался своим старшим (major) и младшим (minor) номерами. Первый определял класс устройств, например, диски, терминалы и псевдотерминалы, параллельные или последовательные порты и т. д. Младший же номер был идентификатором конкретного устройства в данном классе. Очевидно, что сочетание старшего и младшего номеров должно быть уникальным.

Файлы устройств генерировались в некотором соответствии с реальностью, но по принципу явной избыточности. Например, при наличии IDE-контроллера, поддерживающего до четырех устройств, создавались файлы

для всех теоретически подключаемых к нему дисков, даже если в наличии имелся только один. То же и с псевдотерминалами: файлов вида pty* в каталоге /dev можно было обнаружить сколько угодно (и понятно, ведь каждый запущенный в X-сессии экземпляр эмулятора терминала вроде xtrem требовал своего такого устройства). В результате каталог /dev приобретал объем просто необозримый.

И тем не менее, подчас наличных файлов оказывалось недостаточно для представления всех нужных устройств. И тогда пользователю приходилось их создавать самостоятельно. Файлы для большинства распространенных устройств можно было создать с помощью специально предназначенного сценария /dev/MAKEDEV. Однако иногда он оказывался бессильным (особенно для новых устройств, сценарием еще не учтенных), и приходилось прибегать к чисто ручному созданию файлов устройств командой `mknod`. Что было не так уж и трудно само по себе, но требовало знания старшего номера создаваемого устройства (младший при этом вычислялся легко).

Появление devfs избавило пользователя от этих забот. Отныне файлы устройств пересоздаются автоматически при старте системы, и только в соответствии с наличным (и поддерживаемым текущей конфигурацией ядра) оборудованием, определяемым в ходе загрузки. Если же в ходе работы какого-либо устройства из числа поддерживаемых недостанет — соответствующий ему файл будет создан сам собой.

Более того, devfs сделала очень легким "горячее" подключение устройств (типа карт PCMCIA, USB-накопителей, цифровых камер и сканеров с этим интерфейсом). Достаточно воткнуть USB-драйв в соответствующий разъем, и в каталоге /dev можно будет наблюдать появление соответствующего ему файла.

В настоящее время в Linux файловая система устройств постепенно заменяется системой поддержки динамического именования устройств udev, однако использование ее не стало еще общепризнанным.

В отличие от devfs, udev не новая поддерживаемая ядром файловая система, пусть и виртуальная, а обычная пользовательская программа (входящая в одноименный пакет). Более того, при ее использовании необходимость в поддержке devfs отпадает вообще.

Правда, для своего функционирования udev нуждается в еще одной виртуальной файловой системе sysfs, но ее поддержка в ядрах серии 3.6.X осуществляется автоматически (а сама эта файловая система монтируется по умолчанию в каталог /sys). Основываясь на информации из которой, udev и присваивает имена всяческим устройствам (в том числе и при горячем их подключении).

Как уже говорилось, любой POSIX-системе имена файлов конкретных устройств более или менее безразличны, так как оперирует она не с ними, а с их идентификаторами. Ранее в качестве таковых выступали старший и младший номера устройства, определяющие их класс и конкретный экземпляр его. В udev же в ход пошли непосредственные идентификаторы устройств — серийный номер винчестера, его положение на канале IDE-шины, и т. д., сочетание которых для каждого диска (раздела и т. д.) оказывается уникальным. Программа udev извлекает эти сведения из файловой системы sysfs и, руководствуясь определенными правилами, ставит им в соответствие "человеческие" имена (вроде /dev/hda).

Во FreeBSD и многих дистрибутивах Linux используется также файловая система в оперативной памяти, именуемая mfs в первом случае и tmpfs — во втором. Реализованы они по-разному, но с точки зрения пользователя выглядят почти одинаково, как дисковые разделы, смонтированные в некоторые каталоги. И на самом деле они заменяют собой блочные устройства там, где требуется быстрая, но не обязательно долговременная, запись. То есть их целесообразно использовать для всякого рода промежуточных каталогов при архивации/разархивации, пакетной конвертации графических файлов, а также компиляции программ.

В отличие от "нормальных" файловых систем для хранения данных, mfs и tmpfs не нуждаются в создании: для ее использования достаточно факта поддержки ядром (непосредственно или в виде модуля) и монтирования в какой-либо каталог. Обычно таким каталогом выступает /tmp, содержимое которого не должно сохраняться после рестарта системы. В BSD-системах в некоторых случаях целесообразно монтирование mfs в каталог /usr/obj, предназначенный для промежуточных продуктов компиляции при сборке ядра и базовых компонентов. А в Linux одно из штатных мест монтирования tmpfs — каталог /dev/sh. В Source Based дистрибутивах имеет смысл размещать на tmpfs также каталоги для продуктов промежуточной сборки.

И в заключение нужно сказать несколько слов о свопинге и разделе для него. Строго говоря, свопинг не имеет отношения к файловым системам, однако лучшего места для этой темы я придумать не смог.

Итак, ранее неоднократно упоминалось о разделе, предназначенном для свопинга (области диска, куда может выгружаться часть содержимого оперативной памяти). В Linux и BSD механизм его несколько различен. В первой системе освобождение оперативной памяти происходит при ее переполнении. В системах же BSD-семейства в swap-раздел помещаются фрагменты ОЗУ, не использовавшиеся в течение определенного времени.

С точки зрения программ (и их пользователя), физическая память и раздел свопинга представляют собой единое пространство виртуальной памяти. Что

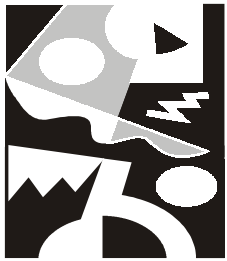
особенно четко выражено в BSD, где ОЗУ может рассматриваться как своего рода кэш по отношению к виртуальной памяти.

Для того чтобы использовать область подкачки, нужно создать соответствующий раздел (первичный или логический — без разницы, с соответствующим идентификатором, каковой для Linux swap будет 82). В BSD-системах под свопинг отводится подраздел (subpartition) внутри слайса.

Рекомендованный размер раздела подкачки во всех системах — удвоенный объем оперативной памяти. Что нынче может показаться излишеством: в машинах типичных современных конфигураций (с ОЗУ минимум 256, а то и 512 Мбайт) под Linux раздел подкачки в ходе обычной пользовательской работы практически не задействуется, а в BSD-системах используется лишь ничтожная его доля.

Тем не менее, большие объемы swap-раздела целесообразны при активном использовании временных систем в оперативной памяти — tmpfs и mfs. В этом случае при заполнении ОЗУ эти файловые системы автоматически могут быть расширены за счет раздела подкачки. Казалось бы, это смена шила на мыло: какой смысл подменять disk based файловую систему на систему, частично размещаемую на swap-разделе (то есть также на диске)? Однако некоторый резон к тому есть, ибо в область свопинга попадают наименее используемые части ОЗУ, так что некоторый выигрыш в быстродействии операций с временными файлами сохраняется. Правда, не стоит обольщаться, выигрыш этот не очень велик, и в наиболее благоприятных условиях не превысит 10%.

Кроме выделения раздела под свопинг, требуется явным образом определить его в этом качестве и активизировать специальными командами. Поддержка виртуальной памяти — одна из жестких функций ядра и Linux, и BSD-систем. Единственное, в чем оно еще нуждается, — это в указании дискового раздела в файле /etc/fstab. Впрочем, это будет темой ближайшей *интермедии* после *главы 10*.



Глава 10

Файловая иерархия

- Принципы построения иерархии каталогов.
- Типовой набор каталогов POSIX-системы.
- Корневая файловая система.
- Ветвь /usr.
- Ветвь /usr/local.
- Ветвь /opt.
- Ветвь /var.
- Каталог /mnt.
- Ветвь /home.
- Ветвь /tmp.

Одной из отличительных особенностей логического устройства файловой системы POSIX-семейства является их иерархическая, или древовидная организsация (правда, как я уже говорил, выглядит это дерево немного странно). То есть здесь нет, как в DOS или Windows любого рода, обозначений (например, буквенных или каких-либо иных) для отдельных носителей и их разделов: все они включаются в единую структуру в качестве подкаталогов главного каталога, называемого корневым. Процесс подключения файловых систем на самостоятельных физических носителях (и их разделах) к корню файлового дерева называется монтированием, а подкаталоги, содержимое которых они составляют, именуются точками монтирования.

Принципы построения иерархии каталогов

Исторически в UNIX сложилась определенная структура каталогов, весьма сходная в разных представителях этого семейства в общих чертах, но не-

сколько различающаяся в деталях. В частности, файловая иерархия в BSD-системах почти одинакова, отличаясь от таковой в Linux. А в той существенные отличия обнаруживаются между разными дистрибутивами. Вплоть до того, что структура файловой иерархии является одним из дистрибутив-специфичных признаков.

Такое положение дел затрудняет сочинение кроссплатформенных приложений. И потому существует и активно развивается проект стандартизации файловой иерархии FHS (Filesystem Hierarchy Standard), основополагающий документ которого ныне доступен в русском переводе (**http://rus-linux.net/MyLDP/file-sys/fhs-2.2-rus/index.html**), за что спасибо Виктору Костромину.

Проект FHS был направлен первоначально на упорядочивание структуры каталогов в многочисленных дистрибутивах Linux. Позднее он был приспособлен для других UNIX-подобных систем (в том числе и BSD-клана). И ныне прикладываются активные усилия, чтобы сделать его стандартом для POSIX-систем не только по имени, но и фактически.

Стандарт FHS покоится на двух основополагающих принципах: четком отделении в файловой иерархии каталогов разделяемых и неразделяемых, с одной стороны, и неизменяемых и изменяемых — с другой.

Противопоставление разделяемых и неразделяемых каталогов обусловлено изначально сетевой природой UNIX. То есть данные, относящиеся к локальной машине (например, файлы конфигурирования устройств), должны лежать в каталогах, отдельных от тех, содержимое которых доступно с других машин в сети, локальной или глобальной (примером чему являются не только пользовательские данные, но и программы).

Суть противопоставления неизменяемых и изменяемых каталогов легко пояснить на примере. Так те же общие пользовательские программы по природе своей должны быть неизменяемыми (вернее, доступными для модификации только администратору системы, но не самому пользователю, применяющему их в своей деятельности). В то же время эти программы при своей работе генерируют не только файлы данных, скажем, тексты или изображения (изменяемая их природа ясна без комментариев), но всякого рода служебную информацию типа log-файлов, временных файлов и т. п.

Тем не менее, несмотря на активное продвижение во многих дистрибутивах Linux, статуса истинного стандарта FHS не обрел. А с традиционной файловой иерархией BSD-систем он соотносится лишь частично.

И причина в том, что FHS игнорирует еще одно противопоставление, очень важное для пользователя: противопоставление легко восстановимых частей файловой системы и тех ее компонентов, которые восстановимы с трудом или невосстановимы вообще.

К первым, как ни странно, можно отнести базовую систему: ведь переустановка ее с дистрибутивного носителя, в случае фатального краха, дело не столь уж сложное. К трудновосстановимым частям файловой системы, очевидно, относятся пользовательские данные: даже в случае регулярного их резервирования (а многие ли пользователи столь аккуратны?) развертывание их из архивов потребует немалого времени (и почти неизбежно повлечет некоторые потери).

Кроме того, в BSD-системах и Source Based дистрибутивах Linux к трудновосстановимым каталогам я отнес бы все, связанное с пакетным менеджментом: дерево портов FreeBSD или pkgsrc в NetBSD (и системах, его заимствовавших), их аналоги в дистрибутивах Linux, собственно исходники портированных программ да и исходные тексты системы тоже. Ибо даже если все это имеется на дистрибутиве, эти компоненты файловой системы, как правило, поддерживаются пользователем в актуальном состоянии путем синхронизации в сети с серверами проекта (иначе их использование лишено смысла). И их утрата повлечет как временные (особенно при модемном подключении), так и финансовые потери.

Типовой набор каталогов POSIX-системы

Собственно говоря, для функционирования необходимо наличие лишь одной файловой системы, той, что монтируется в корневой каталог файлового дерева. Корневой каталог и его непременные ветви обязательно должны составлять единую файловую систему, расположенную на одном носителе: диске, дисковом разделе, программном либо аппаратном RAID-массиве или логическом томе в понимании LVM. И в нем должны располагаться все компоненты, необходимые для старта системы.

Просмотреть состав корневого каталога можно командой

```
$ ls -1 /,
```

которая в любой POSIX-системе покажет некий минимальный джентльменский набор каталогов:

```
bin/
boot/
etc/
root/
sbin/
```

Именно в них собраны все файлы, без которых система не может существовать. Прочие каталоги примерно такие:

```
home/
mnt/
```

```
opt/
tmp/
usr/
var/
```

Они не обязательны (по крайней мере, теоретически, практически обойтись без них трудновато), не каждый из них присутствует во всех системах и дистрибутивах, и каждый из них может быть точкой монтирования собственной ветви файлового дерева.

Кроме этого, в большинстве случаев в корне файловой системы POSIX-совместимых ОС присутствуют еще два подкаталога:

```
dev/
proc/
```

Это обычно точки монтирования виртуальных файловых систем, устройств и процессов соответственно (хотя если файловая система устройств не используется, каталог /dev обязательно должен быть компонентом корневой файловой системы. Наконец, в Linux-системах, как правило, в корне файлового дерева лежит еще и каталог /lib, предназначенный для главных системных библиотек. А при использовании механизма udev неизбежным оказывается еще и каталог /sys, в который монтируется виртуальная файловая система sysfs.

Корневая файловая система

Корневая файловая система является неразделяемой (не предназначенной для совместного использования разными машинами сети) и неизменяемой (изменения в нее может вносить только администратор системы, но не пользовательские программы и, тем более, пользователи). Причем крайне не рекомендуется создавать в ней подкаталоги сверх предусмотренных стандартом (и перечисленных ранее).

Наполнение корневой файловой системы подбирается с таким расчетом, чтобы машина могла стартовать и сохраняла минимальную функциональность даже при аварийной загрузке (или в однопользовательском режиме), когда все остальные файловые системы не монтируются (и /usr или /var могут оказаться недоступными).

В соответствии с этим старт машины обеспечивается файлами каталогов /boot и /etc. В первом размещаются ядро системы — исполнимый файл "особого назначения" и все, что требуется для его загрузки: в Linux, например, это системная карта (файл /boot/System.map), а во FreeBSD — загружаемые модули ядра. Впрочем, подчас ядро размещается непосредственно в

корне файловой системы, и тогда каталог /boot может отсутствовать вовсе, а под модули ядра может отводиться каталог /modules.

Каталог /etc предназначен для общесистемных конфигурационных файлов, определяющих условия ее загрузки. Содержимое его очень сильно зависит от системы (а в Linux еще и от дистрибутива), и потому рассматривать его здесь я не буду, к этой теме придется еще не раз возвращаться.

Минимально необходимая функциональность обеспечивается содержимым каталогов /bin и /sbin: в них собраны исполняемые файлы важнейших пользовательских и системных программ, соответственно, тех самых, которые позволят выполнить комплекс ремонтно-спасательных мероприятий и привести машину в человеческий вид после сбоя.

Разнесение системных и пользовательских программ по подкаталогам корня достаточно условно. Ни одна из команд этих для решения пользовательских задач по-настоящему не предназначена. Просто в каталоге /bin собраны команды администрирования, к которым время от времени обращается (или может обратиться) и обычный пользователь, а каталог /sbin предназначен для команд, о которых пользователю и знать-то не положено. И которыми он, в большинстве случаев, все равно не сможет воспользоваться по причине отсутствия соответствующих полномочий (например, требуемых прав доступа к файлам устройств).

Для запуска POSIX-программ (в том числе и тех, что собраны в каталогах /bin и /sbin), как правило, требуется доступ к функциям общесистемных библиотек (в первую очередь главной библиотеки libc). И потому (почти) непременный компонент корневого каталога — подкаталог /lib, в коем они и собраны.

В Linux каталог /lib служит еще одной важной цели: в его подкаталоге (/lib/modules) собраны загружаемые модули ядра (во FreeBSD их место — каталог /boot/kernel).

Во FreeBSD каталога /lib в корневой файловой системе не обнаруживается, соответствующие компоненты здесь размещаются в /usr/lib. Исторически во FreeBSD важнейшие общесистемные программы собирались так, что требуемые им библиотечные функции встраивались в их исполняемые файлы (так называемая статическая линковка, о которой речь пойдет в *главе 14*). Во FreeBSD пятой ветки программы из каталогов /bin и /sbin линкуются динамически, то есть при отсутствии каталога /usr (а во Free это почти всегда отдельная ветвь файловой системы) они не функционируют. В компенсацию этого предусмотрен выходящий за рамки стандартов каталог /restore, содержащий те же программы, но слинкованные статически (как следует из имени каталога, единственным назначением его содержимого являются аварийно-спасательные работы).

И, наконец, /root. Это обычный домашний каталог одноименного пользователя, администратора системы. Поскольку никакой практической работы он не делает (или, по крайней мере, делать не должен), содержимое его — лишь собственные конфигурационные файлы суперпользователя (пользовательской командной оболочки, любимого редактора и т. д.).

Ветвь /usr

Исторически каталог /usr предназначался для пользовательских программ и данных. Ныне эти функции распределены между каталогами /usr/local и /home (хотя и сейчас во FreeBSD по умолчанию последний представляет собой символическую ссылку на /usr/home). Каталог же /usr, не изменяемый, но разделяемый, служит вместилищем основной части прикладных программ и всего, что к ним относится: исходных текстов, конфигурационных файлов, разделяемых библиотек, документации и тому подобного хозяйства.

Состав каталога /usr существенно различается в BSD-системах и в Linux. Во-первых, в него помещаются только неотъемлемые части операционной системы (того, что во FreeBSD объединяется понятием Distributions). Приложения же, устанавливаемые из портов или пакетов, имеют место своей прописки подкаталог /usr/local, который может представлять отдельную ветвь файлового дерева.

В Linux каталог /usr служит вместилищем всех программ (и их компонентов), штатно включенных в состав дистрибутива. А подкаталог /usr/local предназначается обычно для программ, самостоятельно собираемых из исходников.

В любом случае, обычный состав каталога /usr (по выводу команды `ls -l`) следующий:

```
X11R6/
bin/
etc/
include/
lib/
libexec/
local/
sbin/
share/
src/
```

Как уже говорилось, подкаталог /usr/local — отдельная ветвь файлового дерева, и потому будет рассмотрен отдельно же. Назначение же прочих каталогов таково:

- /usr/bin и /usr/sbin предназначены для исполняемых файлов пользовательских и системных программ (здесь граница между ними еще более условна, чем в случае корневого каталога), назначение которых выходит за рамки обеспечения базового функционирования системы;
- /usr/etc предназначается для конфигурационных файлов отдельных приложений;
- /usr/include содержит так называемые заголовочные файлы, необходимые для линковки исполняемых файлов с библиотечными компонентами;
- /usr/lib и/usr/libexec — каталоги для разделяемых библиотек, от которых зависят пользовательские приложения;
- /usr/share — вместилище самых разнообразных, архитектурно независимых, компонентов: здесь можно видеть и документацию в разных форматах, и примеры конфигурационных файлов, и данные, используемые программами управления консолью (шрифты, раскладки клавиатуры), и описание часовых поясов;
- /usr/src — каталог для исходных текстов; в Linux тут штатно помещаются только исходники ядра (ядер) системы, в BSD же клонах полный набор исходников того комплекса, который во FreeBSD именуется Distributions; исходники самостоятельно собираемых программ помещать сюда, как правило, нежелательно;
- /usr/X11R6 — каталог для компонентов оконной системы Икс — исполняемых файлов (/usr/X11R6/bin), библиотек (/usr/X11R6/lib), заголовков (/usr/X11R6/include), документации (/usr/X11R6/man); файлы Иксовых приложений сюда помещаться не должны (за исключением оконных менеджеров), их место в /usr, /usr/local или /opt, в зависимости от системы.

Кроме этого, в каталоге /usr могут обнаружиться подкаталоги /usr/var и /usr/tmp, обычно символические ссылки на соответствующие ветви корневого каталога. А в некоторых дистрибутивах Linux непосредственно в /usr помещается и основная общесистемная документация — man-страницы (в подкаталог /usr/man).

Наконец, в BSD-системах и некоторых Source Based дистрибутивах Linux (например, Gentoo) в каталоге /usr размещается подкаталог для системы управления пакетами: портов FreeBSD и OpenBSD (/usr/ports), их аналогов в других системах (/usr/portage в Gentoo).

Ветвь /usr/local

Как уже было сказано, ветвь /usr/local в Linux предназначена для самостоятельно собираемых из исходников (не входящих в дистрибутив) программ. А во FreeBSD она служит вместилищем большей части пользовательских приложений, почти всего того, что выходит за рамки Distributions и устанавливается из пакетов или портов. Соответственно этому, структура каталога в целом повторяет таковую ветви /usr (с понятными исключениями):

```
bin/
etc/
include/
lib/
man/
sbin/
share/
```

Содержимое подкаталогов также аналогично: исполняемые файлы программ (/usr/local/bin и /usr/local/sbin), их конфиги (/usr/local/etc), библиотеки, с которым они связаны, и их заголовочные файлы (/usr/local/lib и /usr/local/include соответственно), man-страницы (/usr/local/man) и всякая архитектурно независимая всячина (/usr/local/share), в том числе и документация в иных форматах.

Ветвь /opt

Каталог /opt предусмотрен стандартом FHS, но реально используется не во всех дистрибутивах Linux, а в BSD-системах и вовсе отсутствует. Тем не менее, все больше программ пишется в расчете на инсталляцию именно в этот каталог.

Исторически каталог /opt предназначался в Linux для коммерческих приложений и всякого рода программ не вполне свободного характера. Ныне же его назначение — размещение больших программных комплексов, таких как библиотека Qt, KDE со всеми его компонентами и приложениями, OpenOffice.org и т. п. Структура каталога должна быть такой: /opt/pkg_name. Вот как выглядит она в моей системе (Archlinux):

```
$ ls -1 /opt
gnome/
kde/
OpenOffice.org1.1.2/
qt/
```

Каждый из подкаталогов имеет собственную внутреннюю структуру:

```
$ ls -1 /opt/*

/opt/gnome:
bin/
lib/
man/
share/

/opt/kde:
bin/
etc/
include/
lib/
share/

/opt/OpenOffice.org1.1.2:
help/
LICENSE
LICENSE.html
program/
README
README.html
setup@
share/
spadmin@
THIRDPARTYLICENSEREADME.html
user/

/opt/qt:
bin/
doc/
include/
lib/
mkspecs/
phrasebooks/
plugins/
templates/
translations/
```

Назначение подкаталогов внутри /opt/pkg_name легко угадывается, по аналогии с /usr и /usr/local. Например, /opt/kde/bin предназначается для исполняемых файлов системы KDE и ее приложений, /opt/kde/etc — для конфигурационных ее файлов, /opt/kde/include — для файлов заголовков, /opt/kde/lib — для библиотек и /opt/kde/share — для разделяемых файлов, в том числе и документации. В KDE нет документации в man-формате, если же она имеется, то можно видеть подкаталог /opt/pkg_name/man.

Можно видеть, что структура каталога /opt отступает от исторически сложившейся (и внутренне обоснованной) POSIX-традиции объединения в каталоги однотипных компонентов: исполняемых файлов, библиотек и т. д. И при большом количестве инсталлированных в него программ создает определенные трудности, приходится либо перегружать значениями переменную `$PATH`, обеспечивающую быстрый доступ к командам (*о чем будет говориться в главе 12*), либо создавать специальный каталог /opt/bin и помещать в него символические ссылки на исполняемые бинарные модули программ. Поэтому в ряде дистрибутивов Linux (например, в CRUX) каталог /opt не используется принципиально. Как, впрочем, и во всех BSD-системах. Вполне возможно, что так оно и лучше...

Ветвь /var

Как явствует из названия, каталог /var предназначен для хранения изменяемых файлов, генерируемых в ходе нормальной жизнедеятельности различных программ — программных кэшей, log-файлов, спулинга печати и почтовых систем, почтовых ящиков, описаний запущенных процессов и т. д. В частности, именно в каталог /var помещаются так называемые дампы — слепки состояния оперативной памяти, генерируемые при аварийном завершении работы для выявления причин его. Отличительная особенность всех этих компонентов — их изменчивый в процессе сеанса работы характер и то, что они, тем не менее, должны сохраняться при перезагрузке системы.

Внутренняя структура /var очень сильно меняется от системы к системе, и поэтому на деталях ее устройства я задерживаться не буду. Замечу только, что этот каталог — логичное место для помещения компонентов всякого рода портообразных систем управления пакетами, как это сделано, например, в дистрибутиве Archlinux, где под нее отведен подкаталог /var/abs (abs — Archlinux Building System).

Каталог /mnt

Каталог /mnt предназначен для монтирования временно используемых файловых систем, располагающихся, как правило, на сменных носителях.

В только что установленной системе он обычно пуст, и структура его никак не регламентирована. Пользователю вольно создать в нем подкаталоги для отдельных видов носителей. Например, в моей системе это /mnt/cd, /mnt/dvd, /mnt/usb и /mnt/hd — для дисков CD, DVD, флэшки и съемного винчестера.

Во FreeBSD штатными каталогами для монтирования CD и дискет являются /cdrom и /floppy непосредственно в корневом каталоге. Это не вполне согласуется со стандартом, но по своему логично: в корень вынесены точки монтирования устройств, существующих (как CD-ROM) или до недавнего времени существовавших (флоппи-дисковод) в любой машине.

С некоторого времени во многих дистрибутивах Linux, наряду с каталогом /mnt, появился еще и каталог /media, имеющий сходное назначение, но используемый для автоматического, а не ручного, монтирования сменных устройств.

Ветвь /home

Каталог /home предназначен для помещения домашних каталогов пользователей. Содержимое его никак не регламентировано, но обычно он имеет вид вроде /home/{username1,...,username#}. Хотя в крупных системах с большим количеством пользователей их домашние каталоги могут быть объединены в группы.

В каталоге /home могут располагаться домашние каталоги не только реальных, но и некоторых виртуальных пользователей. Так, если машина используется в качестве Web- или FTP-сервера, можно видеть такие подкаталоги, как /home/www или /home/ftp соответственно.

Ветвь /tmp

Осталось поговорить только о каталоге для хранения временных файлов /tmp. Как и компоненты /var, они генерируются различными программами в ходе нормальной их жизнедеятельности. Но, в отличие от /var, для компонентов /tmp не предполагается их сохранения вне текущего сеанса работы. Более того, все руководства по системному администрированию рекомендуют регулярно (например, при рестарте машины) или периодически очищать этот каталог. И потому в качестве /tmp целесообразно монтировать файловые системы в оперативной памяти — tmpfs (в Linux) или mfs (во FreeBSD). Кроме того, что это гарантирует очистку его содержимого при перезагрузке, так еще и способствует быстродействию, например, компиляции программ, временные продукты которой не записываются на диск, а помещаются в виртуальный каталог типа /tmp/obj.

Во многих системах можно увидеть каталоги вроде /usr/tmp и /var/tmp. Это, как правило, символические ссылки на /tmp.

Стратегия разделения файловых систем

В заключение разговора о файловой иерархии следует подчеркнуть, что гарантированно в одной файловой системе (фигурально говоря, на одном дисковом разделе, хотя это и не совсем точно) должны находиться только каталоги, перечисленные в разделе о корневой файловой системе. Все же прочие каталоги: /usr, /opt, /var, /tmp и, конечно же, /home — могут представлять точки монтирования самостоятельных файловых систем на отдельных физических носителях или их разделах.

Более того, в локальной сети каталоги эти вполне могут располагаться даже на разных машинах. Так, один компьютер (сервер приложений) может содержать разделяемые в сети каталоги /usr и /opt, другой (файл-сервер) — вмещать все домашние каталоги пользователей и т. д.

Осталось только решить, какие файловые системы целесообразно вычленить из общего файлового дерева, и зачем это делать. Ответ на эти вопросы очень сильно зависит от используемой ОС, а в случае с Linux еще и от его дистрибутива. Тем не менее, общие принципы отделения файловых систем наметить можно. Для этого следует вспомнить о противопоставлении, с одной стороны, неизменяемых и изменяемых каталогов, с другой — легко восстановимых, трудно восстановимых и практически невосстановимых данных. Проделаем такую попытку применительно к пользовательскому десктопу, в случае сервера расчеты будут существенно иными.

Очевидно, что корневая файловая система в составе каталогов /bin, /boot, /etc, /root, /sbin, содержащих легко восстановимые с дистрибутивного носителя и практически не изменяемые данные, должны лежать на изолированном дисковом разделе. В Linux к ним должен добавиться еще и каталог /lib. С другой стороны, при использовании в качестве загрузчика GRUB (вне зависимости от операционной системы) рекомендуется вынести на отдельный раздел каталог /boot.

В старых источниках о Linux можно прочитать о другом резоне к выделению раздела для каталога /boot, причем в самом начале диска: из-за невозможности загрузки ядра программой Lilo с цилиндра номером выше, чем 1023. В современных версиях загрузчиков таких ограничений нет. Тем не менее, если уж раздел под /boot создается, резонно сделать его первым на диске, а непосредственно за ним разместить раздел подкачки: это добавит пять копеек быстродействия при осуществлении свопинга.

И еще из области истории: требование, чтобы корневой и загрузочный разделы Linux-системы были непременно первичными, также давно снято.

И эти файловые системы вполне могут помещаться на логических разделах внутри Extended Partition.

Столь же ясно, что изменяемые ветви файловой системы — каталоги /var и /tmp — должны быть вынесены за пределы корневого раздела. Причем последний, как неоднократно говорилось ранее, вообще целесообразно разместить в файловой системе оперативной памяти (tmpfs или mfs). В случае, если каталог /var содержит подкаталоги для портообразных систем пакетного менеджмента, подобно /var/abs, /var/cache/pacman/src и /var/cache/pacman/pkg в Archlinux, они также должны образовывать самостоятельные файловые системы.

Теперь каталог /usr, содержащий либо компоненты базовой системы (как в BSD), либо основную массу пользовательских приложений (как в большинстве дистрибутивов Linux). Он содержит легковосстановимые данные и, по-хорошему, должен бы быть практически неизменяемым, и потому, безусловно, заслуживает выделения в самостоятельном разделе. Причем из его состава целесообразно вычленить, с одной стороны, подкаталоги /usr/X11R6 и /usr/local, с другой — подкаталоги для портообразных систем пакетного менеджмента: /usr/ports, /usr/pkgsrc и /usr/pkg в BSD-системах, /usr/portages в Gentoo Linux и т. д. Причем от последних следует обособить подкаталоги для помещения исходников, скачиваемых из сети при сборке портов — /usr/ports/distfiles, /usr/pkgsrc/disfiles, /usr/portages/distfiles и подобные им.

В BSD-системах, кроме этого, из каталога /usr имеет смысл выделить подкаталоги /usr/src и /usr/obj, содержащие исходные тексты базовых компонентов (включая ядро) и промежуточные продукты их компиляции, образуемые в результате процедур `make buildworld` и `make buildkernel`.

И, наконец, каталог /home, содержащий изменяемые и часто невосстановимые данные, подлежит вынесению из корня файловой иерархии в обязательном порядке. Причем я всегда стараюсь разместить его либо на отдельном слайсе (в BSD), либо на первичном разделе (в Linux).

Предложенная схема разделения файловых систем может показаться излишне усложненной. Однако она дает гарантию обособления легковосстановимых, трудновосстановимых и невосстановимых данных, что облегчит переустановку системы в случае крайней необходимости, и даже миграцию с системы на систему.

Дополнительный ее плюс в том, что для отдельных ветвей файлового дерева, в зависимости от характера размещенных на ней данных, в Linux можно подобрать физически оптимальную файловую систему. Например, для раздела под /boot нет смысла использовать что-либо помимо Ext2fs. Корневой раздел обычно рекомендуется форматировать в надежной и при этом наиболее совместимой Ext3fs. Под каталоги с огромным количеством мелких файлов, такие как /var/abs в Archlinux, /usr/portages в Gentoo, целесообразно задейст-

вовать ReiserFS: ведь умелое обращение с мелкими файлами — это ее профиль. А в каталоге /home, где возможно появление огромных мультимедийных файлов, ко двору может прийтись XFS. Такие меры могут способствовать повышению и надежности хранения данных, и быстродействию файловых операций.

Пользователи BSD-систем безальтернативно привязаны к файловым система типа FFS. Однако и у них есть пространство для маневра. Во-первых, за счет варьирования размеров блока и фрагмента отдельных файловых систем, способствующего либо производительности дисковых операций, либо экономии дискового пространства. А во-вторых, некоторые ветви файлового дерева (такие как /tmp или /usr/obj), вопреки рекомендациям, можно безбоязненно монтировать в чисто асинхронном режиме, выиграв на этом процент-другой производительности.

Вот и подошел к концу затянувшийся рассказ о трех китах POSIX'ивзма: жирные, однако, кашалоты получились. Впереди разговор о том, как наш самый главный кашалот — то есть пользователь — взаимодействует с двумя другими, каким образом он запускает процессы и общается с файлами. Но сначала практическая интермедия о дисковой разметке, создании и монтировании файловых систем. Материал ее требует знакомства не только с предыдущими главами, но и с *главой 12*, посвященной командам.

Интермедия: инструменты дисковой разметки, форматирования и монтирования

- Разметка диска.
- RAID и LVM.
- Создание файловых систем.
- Монтирование.
- Дополнительные утилиты.

Полный цикл подготовки носителя к использованию его в какой-либо POSIX-совместимой ОС включает в себя следующие стадии: разбиение диска на разделы, создание на них файловых систем и их монтирование в файловой иерархии. Об инструментах для этих действий, различающихся в Linux и BSD-системах, и пойдет речь в этой интермедии.

Разметка диска

Как уже говорилось, понятия разметки диска несколько различаются в Linux и BSD-системах. И потому естественно, что выполняются они по-разному.

Разметка диска в Linux

В Linux создание и первичных, и логических разделов — единый процесс, выполняемый с помощью одной и той же программы. Для разбиения диска можно использовать следующие средства:

- низкоуровневую утилиту командной строки sfdisk — инструмент очень гибкий, но сложный в обращении и требующий очень большой аккуратности, все изменения дисковой разметки совершаются в реальном времени;
- интерактивную диалоговую программу fdisk, почти столь же гибкую, как и sfdisk, но более простую и, главное, более безопасную в обращении, изменения дисковой разметки происходят тут только после соответствующего подтверждения пользователем правильности своих действий;

- интерактивную меню-ориентированную программу cfdisk, которая считается еще более простой в использовании, чем fdisk (для которого она служит оболочкой, front-end), и столь же безопасной с точки зрения сохранности данных;
- универсальную (в теории) утилиту parted, которая может использоваться как в режиме командной строки, так и в интерактивном; она позволяет создавать не только дисковые разделы, но и файловые системы на них;
- графические оболочки последней, qparted и gparted.

Кроме того, существуют еще и "продвинутые" дисковые менеджеры типа Disk Druid или HardDrake, а также коммерческие программы типа Partition Magic или Acronis. Так что инструментов разбиения масса, остается только выбрать. Поэтому, воспользовавшись правом выбора, ограничусь рассмотрением традиционных fdisk и cfdisk, а также parted, претендующей на универсальность.

Начнем с fdisk: именно им больше всего пугали в старые времена начинающих пользователей Linux, предлагая дружественные альтернативы типа Disk Druid. Однако при ближайшем рассмотрении выясняется, что ничего устрашающего нет. Запускается просто:

```
$ fdisk /dev/hd?
```

В качестве имени устройства фигурирует имя файла физического диска целиком, например, для мастера на первом канале это /dev/hda. При использовании devfs (и отказе от совместимости со старой номенклатурой) можно прибегнуть к форме

```
$ fdisk /dev/discs/disc0/disc
```

или к указанию полного имени файла устройства для IDE-диска:

```
$ fdisk /dev/ide/host0/bus0/target0/lun0/disc
```

Да и дальше не сложнее: мы получаем в свое распоряжение интерфейс, требующий ввода определенной команды, исполнение которой сводится к ответу на несколько вопросов. С полным списком доступных команд можно ознакомиться благодаря прекрасной системе помощи, вызываемой командой `m`. Так, команда `p` выведет текущий список дисковых разделов с указанием их типа и размера. Далее, разделы можно создавать (командой `n`) или удалять (командой `d`), однако до команды записи изменений (`w`) никаких необратимых действий, могущих разрушить ранее существующие файловые системы, не последует: неудачно созданные разделы можно удалить и на их месте создать новые. И в любой момент командой `q` можно без всяких последствий выйти из программы.

При создании раздела средствами fdisk сначала определяется, будет он первичным (primary) или расширенным (extended). Рассмотрим сначала первый случай. При нем далее просто указывается номер раздела (от 1 до 4). В этих

пределах номер может быть любым, можно сначала создать раздел 2, а потом 1, или даже весь диск отвести под раздел 4 (именно так размечались фабричным способом zip-диски, а также некоторые флэшки). Номер раздела останется на века, именно он будет идентифицировать файл устройства, соответствующий созданному разделу (например, /dev/hda2 или /dev/discs/disc1/part2).

Далее задается начальный цилиндр создаваемого раздела (по умолчанию первый свободный, для пустого диска просто первый). Однако никто не мешает указать любой другой цилиндр в качестве стартового (на неразбитом пространстве, разумеется). А потом конечный цилиндр (по умолчанию последний физический на неразбитом дисковом пространстве) или просто размер раздела в мегабайтах, например, +300M (и +, и M обязательны, иначе объем диска окажется весьма странным). При задании размера в единицах, отличных от цилиндров, он всегда будет округляться до ближайшего числа, кратного целому количеству последних. Так что не следует удивляться, если вместо искомого раздела в 20 Мбайт возникнет 16-мегабайтный, а вместо 22-мегабайтного — раздел в 24 Мбайт.

При создании расширенного раздела все происходит точно так же, задание номера (очевидно, что в том же диапазоне 1—4), указание начального цилиндра и конечного (или объема в мегабайтах). Однако это еще полдела, нужно поделить расширенный раздел на логические. И потому при следующей команде на создание раздела `n` нам будет предложен уже выбор между первичным (если число их еще не исчерпано) и логическим (ведь второй extended-раздел средствами fdisk создать нельзя):

```
Command (m for help): n

Command action

   l   logical (5 or over)

   p   primary partition (1-4)
```

Дальше же логический раздел создается аналогично первичному.

Для каждого вновь создаваемого средствами fdisk первичного раздела по умолчанию устанавливается идентификатор типа файловой системы Linux native (83 в шестнадцатеричном исчислении), как, впрочем, и для раздела логического. Расширенный раздел также автоматически получает правильный идентификатор своего типа — 5. Однако типы эти не есть нечто неизменное. Более того, по крайней мере, в одном случае изменение типа раздела — необходимость. Это потребуется и для использования таких современных технологий, как Software RAID или LVM, о которых будет говориться позднее.

Делается это командой `t`, после чего запрашивается номер раздела, тип которого должен быть изменен, а затем идентификатор желаемого типа. Пол-

ный список поддерживаемых типов файловых систем (и их идентификаторов) можно вывести командой `l`. Идентификатор типа файловой системы раздела — отнюдь не файловая система, которая на нем размещается. И на разделе Linux native, как это подчеркивает название, можно создать любую файловую систему из числа тех, которые поддерживаются Linux в качестве родных (Ext2fs, Ext3fs, XFS, ReiserFS, JFS).

Теоретически fdisk позволяет присвоить созданному разделу идентификатор типа почти любой из мыслимых файловых систем, от FAT12 до Free-, Open- и NetBSD. Однако сами по себе файловые системы средствами fdisk не создаются, и потому для разделов чуждого типа в дальнейшем потребуется их форматирование (в терминах DOS) в родной среде (например, DOS-командой `FORMAT` для FAT-раздела). Тем не менее, смысл в такой операции есть — резервирование места под ОС, которые будут установлены позже.

Сказанного достаточно, чтобы понять: великое достоинство fdisk — исключительная гибкость: можно определить раздел строго определенного размера и точно позиционировать его на диске. Или зарезервировать в любом месте накопителя неразбитое пространство, с двух сторон окруженное созданными разделами.

Программа cfdisk описывается в литературе гораздо реже, хотя во многих дистрибутивах она пропагандируется как более удобная, чем fdisk (на мой взгляд, все как раз наоборот, но это дело вкуса). Запустить ее можно одноименной командой, без всякого аргумента (хотя таковой в виде имени файла устройства, как в fdisk, и не возбраняется).

После запуска программы выводится информация о диске, первом физическом или том, что был указан в качестве аргумента (имя файла устройства, размер, число головок, секторов, цилиндров), таблица существующих разделов (если, конечно, они действительно существуют) и меню из следующих пунктов:

- **Bootable**;
- **Delete**;
- **Help**;
- **Maximize**;
- **Print**;
- **Quit**;
- **Type**;
- **Units**;
- **Write**.

Это для диска с существующими разделами. Если же диск не разбит (или в таблице разделов курсор зафиксирован на неразбитом пространстве), меню ограничивается пунктами:

- **Help**;
- **New**;
- **Print**;
- **Quit**;
- **Units**;
- **Write**.

Смысл пунктов, думаю, понятен из их названий, как и возможности программы вообще. Замечу лишь, что здесь, как и в fdisk, до выбора пункта **Write** (в котором будет запрошено подтверждение действия) никаких необратимых изменений не происходит: через **Quit** всегда можно покинуть программу без боязни за существующие разделы и данные на них. И еще, по умолчанию размеры разделов в таблице указаны в мегабайтах. Однако через пункт **Units** (Единицы измерения) можно переключиться на показ его в секторах или цилиндрах.

Для создания раздела выбирается пункт **New**, выводящий подменю:

- **Primary**;
- **Logical**;
- **Cansel**.

После выбора типа раздела просто задается желаемый его размер (в мегабайтах) и запрашивается, приписать ли раздел к началу диска или его концу. А потом остается только сохранить разбиение в таблице разделов выбором пункта **Write** (повторяю, с запросом подтверждения, и не просто как **y**, а вводом полного слова **yes**, дабы дать дополнительные мгновения на раздумье). То есть все почти как в fdisk. Это и неудивительно: cfdisk по сути лишь интерфейсная для fdisk оболочка (front-end). Хотя cfdisk несколько менее гибок: например, его средствами раздел в середине неразбитого дискового пространства создать нельзя.

И, наконец, parted. Или, точнее, GNU parted, как подчеркивается в заголовке man-страницы. Эта программа предлагается в рамках проекта GNU (**http://www.gnu.org**) как универсальное средство для работы не только с дисковыми разделами, но и с файловыми системами. И действительно, она позволяет не только выполнить разметку диска, но и создать на разделах файловые системы, а также осуществляет проверку их целостности, удаление, перемещение, копирование и изменение размера разделов существующих.

Использоваться parted может двояким образом: в интерактивном и в командном режиме. Начнем с первого, то есть просто запустим программу одноименной командой, без опций и аргументов. В ответ она выдаст нам предупреждение об отсутствии гарантии, информацию о первом физическом диске системы — имя устройства в полной нотации devfs, данные о геометрии (цилиндры/сектора/головки), предупреждение о том, где кончается 1024 цилиндр, и выведет приглашение командной строки в виде

```
(parted)
```

Интерфейс программы построен по принципу sh-совместимых оболочек и весьма сходен с таковым, например, загрузчика GRUB. Поддерживаются, в частности, редактирование командной строки (обычными управляющими последовательностями, например, <Ctrl>+<D> — удаление символа в позиции курсора, <Ctrl>+<H>— перед оной), просмотр истории команд, автодополнение (клавишей <Tab>). Действия по организации диска выполняются с помощью мнемонически прозрачных команд (`print` — просмотр, `mkpart` — создание раздела, `rm` — его удаление и т. д.). Синтаксис команд также shell-подобный: обычно требуется указание аргумента — номера устройства (Minor, в терминологии программы) и некоторых дополнительных опций. Выход из программы — командой `exit` или комбинацией <Ctrl>+<D>.

Полный список доступных команд с возможными опциями и аргументами, а также краткими, но внятными комментариями можно получить, введя в командной строке

```
(parted) help
```

или просто нажав <Enter> в ответ на приглашение. Список этот включает команды:

- выбора устройства для редактирования (`select /dev/hd?`);
- действий с существующими разделами (`print` — просмотр таблицы разбиения, `chech` — проверка целостности файловой системы раздела, `rm` — удаление раздела, `cp` — копирование файловой системы в другой раздел, `resize` — изменение размера раздела, `move` — перемещение раздела в пределах диска);
- манипуляций по разбиению диска (`mkpart` — создание раздела, `mkpartfs` — создание раздела с файловой системой заданного типа, `mkfs` — создание файловой системы на существующем разделе).

Подробную справку по каждой команде можно получить, введя

```
(parted) help имя_команды
```

Кроме того, справка по использованию команды будет выведена, если дать ее без аргументов и опций. И, разумеется, программа parted сопровождается

документацией в форматах man и info, из которой можно получить исчерпывающие сведения о ее использовании.

В отличие от fdisk или cfdisk, в parted не предусмотрено специальной команды для записи изменений, все действия выполняются в реальном времени, без откладывания. То есть, например, команда

```
(parted) rm #
```

приведет к немедленному удалению раздела с указанным номером. Соответственно, parted требует исключительно аккуратного обращения. Однако в обмен на это предоставляет, во-первых, исключительную гибкость при задании размера раздела и его позиционировании. Во-вторых, он делает доступными множество дополнительных манипуляций разделами и файловыми системами. Правда, в полном объеме — только для файловых систем Ext2fs, ReiserFS и FAT16/FAT32.

Чтобы оценить возможности parted, рассмотрим для примера процесс разбиения вновь приобретенного диска. Для чего после запуска программы сначала выбираем соответствующее устройство:

```
(parted) select /dev/hd?
```

и затем командой

```
(parted) print
```

убеждаемся, что устройство это разбиению не подвергнуто, даем команду для создания раздела:

```
(parted) mkpart type_part type_fs start end
```

Под типом раздела здесь могут выступать значения:

- `primary` (для первичного раздела);
- `extended` (для расширенного);
- `logical` (для логического раздела в расширенном).

Возможные значения для типа файловой системы:

- `ext2`;
- `ext3`;
- `reiserfs`;
- `linux-swap`;
- `FAT`;
- `xfs`;
- `jfs`;
- `hp-ufs`;
- `sun-ufs`.

Однако само по себе создание файловых систем при этом выполнено командой `part` не будет, о чем я скажу чуть позже.

Начало (`start`) и конец (`end`) раздела указываются в мегабайтах, например, 0 и 3000 при создании раздела в 3 Гбайт от начала диска. И начало, и конец можно задать дробными (с точностью до третьего знака и разделителем — десятичной точкой) числами, что обеспечивает необходимую точность разбиения (при наличии калькулятора или способности к счету в уме).

Как легко понять из формата команды, раздел заданного размера может быть создан в любом месте диска (не обязательно в начале его или в конце). Раздел, созданный первым по времени (вне зависимости от положения на диске), получит номер (Minor) 1, созданный вторым (пусть и в начале диска) — Minor 2, и т. д. То есть по гибкости команда `mkpart` из parted ничуть не уступает программе fdisk.

Далее на дисковых разделах должны быть созданы файловые системы. Вообще-то, это будет темой отдельного разговора. Однако поскольку именно эта возможность делает программу parted столь универсальной, затрону ее здесь вскользь. Создание файловой системы осуществляется командой

```
(parted) mkfs # type_fs
```

где под `#` выступает тот самый номер (Minor) раздела, который был присвоен ему при создании. А доступные для создания файловые системы — ext2, ext3, reiserfs, linux-swap и FAT. На попытку приписать разделу, скажем, xfs последует сообщение о невозможности сего действа. Можно надеяться, что это явление временное, и поддержка всех файловых систем для Linux будет включена в грядущие версии parted.

Дисковый раздел и файловая система на нем могут быть созданы так же одной командой:

```
(parted) mkpartfs type_part type_fs start end
```

К опциям ее относится все то, что было сказано чуть раньше об командах `mkpart` и `mkfs`.

Таким образом, создание разделов (и, добавлю, файловых систем) средствами программы parted в интерактивном режиме весьма просто и удобно (при должной, естественно, аккуратности). Однако основные ее преимущества проявляются при использовании в командном режиме. Чтобы прибегнуть к нему, программу parted следует запустить с указанием аргумента (имени файла дискового устройства), одной из его встроенных команд и необходимых последней опций. В итоге одной строкой типа

```
$ parted /dev/hda mkpartfs primary ext2 0 100 && \
parted /dev/hda mkpartfs primary linux-swap 101 1124 && \
parted /dev/hda mkpartfs primary ext2 1125 ###
```

можно создать полностью готовый к использованию в Linux диск, никаких дальнейших действий в этом направлении не потребуется.

Слайсы и разделы в BSD

В отличие от Linux, разбиение диска в BSD-системах осуществляется в два этапа и двумя отдельными программами. Сначала диск нарезается на слайсы (или создается один слайс, в режиме совместимости или для эксклюзивного использования). А затем уже слайс, отведенный для BSD, разбивается на партиции. Далее речь пойдет о FreeBSD и DragonFlyBSD, однако и в Net- или OpenBSD принципиальных отличий не будет.

Выполнению первой задачи — созданию слайса — служит утилита fdisk. Это еще более мощное средство работы с дисками, чем одноименная программа из Linux. Однако ее нельзя назвать легкой в использовании, в этом отношении она более сходна с sfdisk.

Запущенная без опций и аргументов, команда просто выдает информацию о первом физическом диске машины (вернее, о том диске, на котором размещается корневая файловая система BSD). И информацию богатую: здесь мы увидим и имя файла текущего дискового устройства (например, /dev/ad0), и сведения о геометрии (количество цилиндров, головок, секторов на трек, блоков на цилиндр), размер физического блока.

А дальше последует информация о слайсе или слайсах, на этом диске проживающих. И тут для каждого слайса мы увидим идентификатор типа файловой системы, его размер (в блоках и мегабайтах), флаг активности (если таковой имеет место быть), данные о начале и конце (номер цилиндра/головки/сектора). Если на диске существует менее четырех слайсов, несуществующие (то есть соответствующие незаполненным записям таблицы разделов) будут помечены как `unused`.

Как уже сказано, вся эта информация относится к диску с корневой файловой системой. Чтобы получить аналогичные сведения о других накопителях, имя файла соответствующего устройства нужно указать в явном виде в качестве аргумента команды `fdisk`. Например,

```
$ fdisk /dev/ar0
```

предоставит их для диска, подключенного к разъему IDE RAID контроллера. Сведения эти могут показаться избыточными. Однако можно вывести и более краткую (и при этом только существенную) информацию. Чему послужит опция `-s`. В ответ на команду

```
$ fdisk -s /dev/ad#
```

мы получим только самое главное: имя файла устройства, количество цилиндров, головок и секторов, а также краткие сведения только о сущест-

вующих слайсах (стартовый сектор, размер слайса, идентификатор типа файловой системы и флаг активности), примерно в следующем виде:

```
/dev/ad0: 155061 cyl 16 hd 63 sec
Part        Start        Size Type Flags
   1:           0   156301488 0xa5 0x80
```

Все сказанное преследовало своей целью только получение информации. Чтобы с помощью команды `fdisk` осуществить какие-либо активные действия по разметке диска, необходимо ознакомиться с другими ее опциями. Их не так много, и важнейшей, пожалуй, является опция `-I`. Включенная в команду

```
$ fdisk -I /dev/ar0
```

она создаст на диске первый и единственный слайс в режиме совместимости, начиная с 63 сектора. Очевидно, что если диск перед этим был как-то разбит и содержал данные, разметка диска и его содержимое будут безвозвратно уничтожены. Впрочем, такое поведение типично для всех утилит дисковой разметки в любой ОС. Правда, в отличие от одноименной утилиты из Linux, в BSD fdisk выполняет переразметку диска немедленно. И к тому же, тут нас даже не спросят о подтверждении действий, так что следует быть внимательным.

Зато много вопросов последует при использовании опции `-i`, которая позволяет выполнить разметку диска в интерактивном режиме. Данная с именем файла устройства в качестве аргумента, то есть в форме

```
$ fdisk -i /dev/ar0
```

она сперва напомнит нам, а какой, собственно, диск подвергается надругательству, и сообщит его параметры (как записанные в Disk Label, так и считанные из BIOS — в общем случае они совпадать не обязаны):

```
******* Working on device /dev/da0 *******
parameters extracted from in-core disklabel are:
cylinders=124 heads=64 sectors/track=32 (2048 blks/cyl)

parameters to be used for BIOS calculations are:
cylinders=124 heads=64 sectors/track=32 (2048 blks/cyl)
```

И сразу же спросит, а нет ли у нас желания скорректировать BIOS-геометрию диска. Ответ по умолчанию (`no`) очевиден, если нет сообщения о "плохой" BIOS-геометрии, которая совпадает с геометрией, описанной в Disk Label. А вот если факт "плохой" геометрии имеет место быть, стоит задуматься.

В большинстве случаев расхождения между "геометрией" BIOS и Disk Label не страшны. Однако если все же потребуется приведение их в соответствие, после положительного ответа на предложенный вопрос придется вручную задать "правильные" значения числа цилиндров, треков и секторов.

Дальнейшее занятие после исправления геометрии (или вместо нее) — это ручное создание слайсов. Для этого сначала запрашивается идентификатор типа файловой системы (по умолчанию стоит существующий, если диск был размечен, или 0 — для диска нового), следует указать его десятичное значение (165 для BSD-слайса). Затем стартовый сектор (0 при "эксклюзивной" разметке, 63 при разметке в режиме совместимости) и размер слайса в физических, по 512 байтов, блоках (при использовании всего диска, очевидно, он будет равен полному их числу, в противном случае потребуются некоторые арифметические вычисления).

После этого будет предложено точно специфицировать начало и конец слайса. Если отказаться, они будут взяты из предыдущих определений, если согласиться, нужно будет указать первые и последние цилиндр, головку, сектор. Они и будут выведены в виде

```
sysid 165 (0xa5),(FreeBSD/NetBSD/386BSD)
    start 0, size 260000 (126 Meg), flag 0
    beg: cyl 0/ head 0/ sector 1;
    end: cyl 126/ head 60/ sector 32
```

Подтвердив свои действия положительным ответом на вопрос

```
Are we happy with this entry? [n] y
```

можно при желании перейти к созданию второго раздела

```
The data for partition 2 is:

Do you want to change it? [n]
```

по той же схеме, что и первого. Ясно, что если создается всего один слайс, следует отказаться от изменений остальных потенциальных записей таблицы разделов, в этом случае они останутся помеченными как неиспользуемые. В любом случае будет задан последний вопрос — подтверждение выполнения:

```
Do you want to change the active partition? [n]
```

При положительном ответе все сделанные изменения вступят в силу (и на ранее размеченном диске можно будет распроститься с его содержимым). Так что следует предварительно просмотреть все ранее введенное (благо, в BSD это легко сделать пролистыванием буфера истории виртуальной консоли) и при обнаружении ошибки отказаться от изменений и запустить

команду fdisk снова. Впрочем, из нее можно в любой момент выйти без последствий и стандартным образом — комбинацией клавиш <Ctrl>+<C>.

В общем, интерактивное создание единственного слайса не так уж и страшно. Если же слайсов потребуется несколько, придется вооружиться калькулятором (в базовом комплекте BSD есть bc, запускается из командной строки, очень прост и удобен в обращении).

Хотя есть и еще один способ создания слайсов (предварительным описанием их параметров, а заодно и дисковой геометрии, в обычном текстовом файле, посредством любого привычного редактора). После этого fdisk запускается в форме

```
$ fdisk -f configfile /dev/ad#
```

А добавив еще и опцию -t, можно предварительно протестировать правильность своей разметки, не записывая изменений на диск. Впрочем, сам я этого не проделывал, оставляя желающим для самостоятельных упражнений: все необходимые сведения, в том числе и формат файла, можно почерпнуть, выполнив

```
man (8) fdisk.
```

Наконец, для разметки диска в эксклюзивном режиме можно обойтись без команды fdisk вообще: достаточно обнулить начальные его блоки с помощью команды dd, которая осуществляет копирование с преобразованием. Она требует двух аргументов: имени копируемого (if — input file) файла и имени файла устройства, на которое он копируется (of — output file). Можно задать также размер блока копируемых данных и количество их. То есть в нашем случае это будет выглядеть так:

```
$ dd if=/dev/zero of=/dev/ad# bs=1k count=1
```

или

```
$ dd if=/dev/zero of=/dev/ad# count=2
```

В обоих случаях под /dev/zero понимается "нулевое" устройство, а в качестве /dev/ad# выступает размечаемый диск, дополнительные же опции показывают, что нулями должны быть заполнены первые два физических его блока.

Созданный при помощи fdisk слайс, вне зависимости от того, "эксклюзивный" он или "совместимый", еще не пригоден к какому-либо иному использованию. Предварительно его еще нужно разбить на разделы или хотя бы создать один раздел, ad#s1c, описывающий слайс целиком. Этой цели служит утилита disklabel (в DragonFlyBSD), или, во FreeBSD 5.X, bsdlabel (с более дружественным, как говорят оптимисты, интерфейсом). Впрочем, обращение с обеими практически одинаково.

Запущенные без опций (с обязательным аргументом в виде имени устройства), обе команды служат исключительно целям информирования о текущем положении вещей, выводя для BSD-слайса нечто вроде следующего:

```
$ bsdlabel /dev/ad0s1

# /dev/ad0s1:
8 partitions:
#        size   offset    fstype  [fsize bsize bps/cpg]
  a:   524288        0    4.2BSD    2048 16384 32776
  b:  2074624   524288      swap
  c: 156301488        0    unused    0     0     0
  # "raw" part, don't edit
  d:   524288  2598912    4.2BSD    0     0     0
  e: 10240000  3123200    4.2BSD    0     0     0
  f: 142938288 13363200   4.2BSD    0     0     0
```

Для приведенного вывода не лишними будут некоторые комментарии. Литеры слева — это буквенные обозначения существующих партиций, для каждой из которых приведены: размер в блоках, смещение первого блока от начала диска (оффсет), тип файловой системы и ее параметры: размер фрагмента, блока, плотность записей. Не будем пока обращать внимание на то, что в соответствующих колонках для всех разделов, кроме `a`, стоят нули.

Среди партиций обращает на себя внимание помеченная литерой `c`: это тот самый "контейнер" для остальных разделов. Ясно, что смещение для него нулевое, а размер равен полному количеству физических блоков диска. Для прочих партиций смещение легко вычисляется.

В поле `fstype` раздела `c` неслучайно стоит значение `unused`, только он имеется в наличии на только что размеченном с помощью fdisk носителе. Как же создать остальные необходимые партиции?

Как ни странно, один из способов предельно прост: посредством обычного текстового редактора. Для этого `bsdlabel` запускается с опцией `-e`:

```
$ bsdlabel -e /dev/ad0s1
```

В ответ будет вызван редактор, определенный в переменной `EDITOR` профильного файла суперпользователя, при отсутствии ее таким редактором будет /usr/bin/vi. И мы увидим следующее:

```
# /dev/da0:
8 partitions:
#        size   offset    fstype   [fsize bsize bps/cpg]
  c:   254787        0    unused        0     0
  # "raw" part, don't edit
```

Если под файловую систему планируется отвести весь слайс целиком, достаточно в поле `fstype` заменить значение `unused` на `4.2BSD`. Для создания нескольких разделов нам опять же потребуется некоторая арифметика, аналогичная примененной в интерактивном режиме программы fdisk.

Каждый раздел, начиная с `a`, должен получить значение начального оффсета (первый соответствует начальному блоку всего слайса, остальные — сумме оффсета и размера предыдущего), размера (опять же в блоках), типа файловой системы (для "рабочих" партиций 4.2BSD, для раздела подкачки swap). Поля параметров файловой системы можно не заполнять. А можно проставить в них нули, в таком случае параметры файловой системы будут определяться только при создании (форматировании раздела).

Во FreeBSD для подготовки носителей широко применяется (а для начинающих пользователей рекомендуется) универсальная утилита sysinstall; в DragonFlyBSD с этой ролью более-менее справляется программа BSD Installer.

Обе они позволяют добиться поставленной цели — дисковой разметки, а заодно создания файловой системы и ее монтирования, быстро и с минимальным риском для имеющихся данных. Важно только понимать, BSD Installer в современном своем виде имеет некоторые ограничения функциональности. В частности, он позволяет выполнить BSD-разметку только для всего диска или для уже существующего размера с идентификатором 165: разбиение диска на слайсы для него — задача непосильная.

RAID и LVM

Как уже говорилось, в процессе подготовки диска к созданию файловых систем подчас возникает задача не только размежевания, то есть разбиения диска на разделы, но и объединения отдельных разделов в их массивы или логические тома. К рассмотрению инструментария для этого мы теперь и обратимся.

Как и средства разметки, инструменты слияния дисков в массивы существенно различны в Linux и BSD-системах. Как это в обычае в Linux, программный RAID любого выбранного уровня можно создать двумя способами. Однако на каком бы способе мы ни остановились и какой бы RAID ни выбрали, в любом случае потребуется выполнение некоторых условий и действий.

Первое условие, разумеется, физическое наличие более, чем одного диска. Причем очевидно, что число их для линейного режима значения не имеет, а для level 0 предпочтительно четное количество. При линейном режиме порядок подключения дисков к контроллерам большой роли не играет, при

выборе же нулевого RAID желательно разнести их на разные линии IDE-контроллера.

Далее, на дисках необходимо создать (средствами fdisk, cfdisk, parted или любыми специфичными утилитами) два раздела и присвоить идентификатор fd (в шестнадцатеричной нотации), который так и называется — RAID Auto detection. В принципе, это не обязательно, но здорово упрощает жизнь.

Теперь ядро системы: для использования программного RAID в его конфигурации должны быть включены соответствующие опции в пункте **Multi-device support** главного меню, генерируемого по команде

```
$ make menuconfig
```

Рассмотрим их:

- общая поддержка множественных устройств (Multiple devices driver support (RAID and LVM));
- общая поддержка RAID;
- поддержка предполагаемого режима, линейного Linear (append) mode или параллельного RAID-0 (striping) mode.

Общая поддержка множественных устройств может быть только встроена в ядро, прочие же опции либо встраиваются, либо подключаются в качестве модулей. В рассматриваемом нами случае это безразлично. Встраивание поддержки RAID в ядро обязательно только в том случае, если в массиве располагается корневая файловая система и (или) он выступает в качестве загрузочного устройства (ни того, ни другого мы договорились не делать). Да и то, при должном конфигурировании виртуального загрузочного диска (командой `initrd`) даже в этих случаях можно обойтись модулями.

Теперь можно приступать к созданию RAID-массива. Для этого потребуется соответствующие инструменты, объединяемые в один из двух пакетов, традиционный raidtools и более новый mdadm.

Фундаментальный The Software-RAID HOWTO (**http://unthought.net/Software-RAID.HOWTO/**) написан Якобом Остергардом (Jakob Ostergaard) и переведен на русский язык (**http://www.linux.org.ru/books/HOWTO/Software-RAID-HOWTO-2.4.html**) Максимом Дзюманенко. Немало внимания RAID-массивам уделил в своей серии публикаций на сайте IBM Дэниэл Роббинс (часть 1 на **http://www-106.ibm.com/developerworks/linux/library/l-raid1/index.html**, часть 2 на **http://www-106.ibm.com/developerworks/linux/library/l-raid2/index.html**). И все эти документы посвящены исключительно использованию инструментария raidtools (**http://people.redhat.com/mingo/raidtools**).

А вот о mdadm сведения можно почерпнуть только из его штатной документации. Главное же, mdadm много превосходит raidtools в удобстве употреб-

ления (в очередной раз вынужден подчеркнуть, я рассуждаю с позиций пользователя, а не администратора сервера).

Итак, mdadm (**http://www.cse.unsw.edu.au/%7Eneilb/source/mdadm/** либо канонический сайт Linux **http://www.kernel.org/pub/linux/utils/raid/mdadm/**) в виде тарбалла исходников. Обращение после обычной распаковки состоит в последовательности команд `make` и `make install` (обратим внимание, программа столь проста, что в предварительном конфигурировании посредством

```
./configure
```

не нуждается).

По завершении установки мы обнаруживаем единственный исполняемый двоичный файл /sbin/mdadm и пару относящихся к нему man-страниц (mdadm.conf и mdadm), содержащих вполне достаточно информации для того, чтобы приступить к делу создания собственного RAID.

Нетрудно догадаться, что раз из всего пакета в итоге образовался только один файл, именно его и следует запустить для создания массива. Для образования RAID команда `mdadm` требует одной из двух опций: `-C` (эквивалент `--create`) или `-B` (сиречь `--build`), в обоих случаях обратите внимание на регистр краткой формы. В чем разница между ними?

Опция `-B` создает массив без собственного суперблока. Что для пользователя снимает ряд преимуществ инструмента mdadm, и потому к этой опции мы возвращаться не будем. А вот опция `-C` этот суперблок учреждает, и ею определяется вся сила программы mdadm.

Итак, опция `-C`. Элементарная логика подсказывает, что она, будучи основной, требует аргумента — имени файла устройства, соответствующего создаваемому массиву (например, /dev/md0 или, при задействованной файловой системе устройств, /dev/md/0), а также указания некоторых дополнительных данных, как-то: его уровня (режима), количества устройств в нем и, наконец, имен файлов устройств, массив составляющих. Что и достигается указанием опций `--level=#` (сокращенно `-l #`) и `--raid-devices=##` (в краткой форме `-n ##`), после чего перечисляются имена файлов, вроде /dev/hda3, /dev/hdb3). В итоге простейший случай создания RAID параллельного режима выглядит следующим образом:

```
$ mdadm --create /dev/md0 --level=0 --raid-devices=2 /dev/hd[a,b]3
```

Для массива нулевого уровня допустимые значения опции `--level` также `raid0` или `stripe`. А для массива линейного режима значение `linear` (то есть `-l linear`).

Для параллельного режима можно задать еще один параметр — размер блока данных (chunk), в виде одноименной опции `--chunk=` (значение в кило-

байтах, в краткой форме `-c ##`). Теоретически рассуждая, чем больше величина chunk, тем выше должно быть быстродействие массива. Однако практические измерения этого не подтверждают. И потому вполне можно опустить данную опцию, при этом значение по умолчанию составит 64 Кбайт.

В любом случае после указанной выше команды RAID создан, в чем легко убедиться командой

```
$ less /proc/mdstat
```

Более того, он сразу же готов к использованию, на нем можно создать ту или иную файловую систему. Хотя, с другой стороны, никто не запрещает поделить его программой fdisk на разделы (по моим наблюдениям, cfdisk на это не способен).

Внимательный читатель, особенно знакомый с документацией по raidtools, спросит: а где же тут конфигурационный файл, описывающий RAID-массив? Отвечаю: при использовании mdadm, установке соответствующих идентификаторов на составляющих массив разделах (вспомним добрым словом RAID Auto detection) и, наконец, создании массива с собственным суперблоком (вот зачем нужна была опция `-C`) необходимости ни в каком конфиге не возникает.

Однако при желании конфигурационный файл для массива создать можно, в некоторых случаях это упростит дальнейшее им управление. Для чего опять обращаемся к команде `mdadm`. Кроме упомянутых основных опций, она имеет еще несколько дополнительных. И в форме

```
$ mdadm --detail --scan
```

способна вывести информацию о существующем массиве. Достаточно перенаправить ее вывод в файл (таковым традиционно будет /etc/mdadm.conf) — и соответствующий конфигурационный файл будет создан.

Теперь время вспомнить о других опциях команды `mdadm`. С одной из них мы познакомились раньше, это `--help` для получения подсказки. Каковую, кстати, можно конкретизировать, указав эту опцию совместно с какой-либо из других основных опций. Например, команда

```
$ mdadm --create --help
```

распишет нам процесс создания массива в деталях.

С другой опцией (`--detail`, или `-D`, все опции, отнесенные к числу основных, в сокращенной форме даются в верхнем регистре) мы столкнулись только что: она призвана выводить информацию о существующих массивах. Близкий смысл имеют и опции `--query` и `--examine` (в сокращенной форме, соответственно, `-Q` и `-E`), за деталями можно обратиться к man-странице.

А вот опции `--assemble` (`-A`) и `--monitor` (или `--follow` (`-F`)) предназначены для управления существующим массивом. В частности, они позволяют добавить к нему новое устройство или удалить существующее. Правда, при выполнении некоторых условий. В общем, создание RAID-массива средствами mdadm — процесс очень простой. Так что, если нет необходимости в дополнительных возможностях, предоставляемых LVM, при наличии двух дисков есть резон им и ограничиться.

Справедливости ради следует сказать пару слов и об инструментарии raidtools и способах обращения с ним. В отличие от mdadm, он требует обязательного наличия конфигурационного файла — /etc/raidtab. Причем создать его нужно (вручную, в текстовом редакторе) до запуска каких-либо команд по созданию RAID.

Впрочем, структура /etc/raidtab очень проста. Стоит только помнить, что каждый из перечисленных далее пунктов выступает в отдельной строке, значения отделяются пробелами или табулятором:

- имя файла RAID-устройства (`raiddev /dev/md0`, например);
- уровень массива или его режим `raid-level 0` для параллельного режима или `raid-level linear` для линейного;
- количество устройств в массиве `nr-raid-disks 2`;
- размер chunk в килобайтах, например, `chunk-size 32` (очевидно, что для линейного режима эта величина бессмысленна, поэтому здесь можно поставить любое значение);
- вслед за этим можно (а скорее, нужно) указать также, что массив должен нести собственный суперблок `persistent-superblock 1`;
- перечисляются имена всех объединяемых устройств с их порядковыми номерами (`device /dev/hda3 raid-disk 0 device /dev/hdb3 raid-disk 1`).

Закончив редактирование /etc/raidtab, активизируем RAID командой

```
$ mkraid /dev/md0
```

и просмотром файла proc/mdstat убеждаемся, что все произошло так, как и задумывалось.

Сложнее, конечно, чем использование mdadm, но не намного, не так ли? Тем более, что весь процесс создания RAID именно применительно к инструментарию raidtools в деталях расписан в перечисленных ранее документах.

Главное различие между mdadm и raidtools таково: первый содержит один большой универсальный бинарный файл, что удобно для интерактивной работы с RAID. Пакет же raidtools содержит несколько специализированных небольших файлов, а размер файла — важный параметр при использовании загрузочного initrd. Поэтому по традиции в сценариях инициализации ис-

пользуются команды из raidtools. Даже если массив создается с помощью mdadm, после перезагрузки проблем возникнуть не должно, если для объединяемых в массив разделов был установлен идентификатор типа `fd`.

В BSD-системах также имеются два разных механизма создания программных RAID-массивов: классический ccd (Concanetade Disk driver) и более новый vinum. Это не два разных набора инструментов, а именно две различные технологии, обладающие разными возможностями: vinum и ccd. Заинтересованным в механизме vinum предлагается обратиться к FreeBSD Handbook в русском переводе (**http://www.freebsd.org.ua/doc/ru_RU.KOI8-R/books/handbook/index.html**) или DragonFly Handbook в оригинале (**http://leaf.dragonflybsd.org/~justin/handbook/**).

Итак, ccd — драйвер слияния дисков. Средство, как я уже сказал, весьма древнее (man-страница датирована 1995 г.), а значит, проверенное временем. Позволяет создавать программные RAID-массивы типа нулевого (stripping) и первого (mirroring) уровней.

Механизм ccd не позволяет разместить на массиве корневую файловую систему. Поэтому перед его использованием необходимо установить нашу ОС на первый из наличных дисков, задействовав только раздел /dev/ad0s1a и оставив остальное пространство неразмеченным. После этого на обоих дисках разметка разделов для конкатенации может быть выполнена двумя способами. Первая схема выглядит более сложной: сначала деление слайсов на обоих дисках на симметричные разделы под будущие файловые системы, а потом попарное их слияние.

Суть второго способа интуитивно понятна (и аналогична построению RAID в Linux с помощью mdadm): два дисковых раздела одинакового объема сливаются воедино, а потом это объединенное пространство, воспринимаемое как единый слайс, делится на разделы под нужные файловые системы. Итак, на каждом диске требуется создать по слайсу на всем свободном пространстве (объем их должен быть примерно равным), а уж эти слайсы побить на попарные (и равновеликие) разделы, которые после конкатенации составят ветви /var, /usr, /home и все, которые еще планируется (например, /usr/ports и /usr/ports/distfiles).

Разметку слайсов можно выполнить через sysinstall (BSD Installer в DragonFlyBSD) или вручную. С первым способом все понятно — однако он не всегда желателен (а иногда и не проходит). Для ручной разметки переходим в однопользовательский режим (в данном случае это не обязательно, но лучше все действия, связанные с разметкой дисков, выполнять из него — для страховки). Далее размечаем оба диска: первый в интерактивном режиме командой

```
$ fdisk -i /dev/ad0
```

а второй — как единый слайс (напомню, что подвергаемые конкатенации диски желательно разнести на разные IDE-контроллеры):

```
$ fdisk -I /dev/ad2
```

Далее, вооружившись командой `bsdlabel` (или `disklabel`), запускаемой с опцией `-e`, калькулятором командной строки bc и, конечно же, man (8) bsdlabel, на первом (/dev/ad0s1) диске, в дополнение к имевшимся

```
  a:   524288        0    4.2BSD     2048 16384 32776
  b:  2074624   524288      swap
  c: 156301488        0    unused        0     0
```

дописываем строки, определяющие полуразделы под будущие файловые системы, например, /var, /usr и /home. В результате получаем:

```
  d:   524288  2598912    4.2BSD        0     0     0
  e: 10240000  3123200    4.2BSD        0     0     0
  f: 142938288 13363200   4.2BSD        0     0     0
```

После этого обращаемся ко второму диску (/dev/ad2s1), на котором создаем парные к первому разделы под swap: /var, /usr и /home (того же размера, что и на первом диске), что в итоге дает

```
  b:  2074624        0      swap
  c: 156301312        0    unused   0      0
  # "raw" part, don't edit
  d:   524288  2074624    4.2BSD    0      0      0
  e: 10240000  2598912    4.2BSD    0      0      0
  f: 142938112 12838912    4.2BSD   0      0      0
```

Теперь наступает время конкатенации партиций, и для этого имеется специальная утилита ccdconfig. И эту процедуру можно проделать двумя разными способами. Правда, оба требуют перехода в однопользовательский режим, теперь уже обязательно, если мы не сделали этого ранее.

Первый способ — последовательный запуск для каждой пары сливаемых партиций примерно таким образом:

```
$ ccdconfig ccd0 128 none /dev/ad0s1d /dev/ad2s1d
$ ccdconfig ccd1 128 none /dev/ad0s1e /dev/ad2s1e
$ ccdconfig ccd1 128 none /dev/ad0s1f /dev/ad2s1f
```

Здесь `ccd#` — имя создаваемого RAID-устройства, `128` (в примере это значение по умолчанию) — размер (в килобайтах) chunk, флаг `none` отменяет зеркалирование (то есть создает именно sripped-массив; чтобы сделать массив уровня 1, потребовалось бы указать флаг `CCDF_MIRROR` или его шестнадцате-

ричное значение `004`). Аргументы понятны — это имена файлов партиций, которые подвергаются конкатенации.

В результате в каталоге /dev будут автоматически созданы новые устройства — /dev/ccd0, /dev/ccd1, /dev/ccd2 (речь идет о FreeBSD версии 5.X, использующей файловую систему устройств; в четвертой ее ветке и в DragonFlyBSD эти устройства потребовалось бы предварительно создать скриптом /dev/MAKEDEV или командой `mknod`), а в каталоге /etc возникнет конфигурационный файл ccd.config.

Второй способ — создать предварительно конфигурационный файл /etc/ccd.conf (которому на самом деле можно дать произвольное имя и поместить где угодно) в текстовом редакторе и описать в нем объединяемое примерно таким образом:

```
# ccd   ileave  flags   component devices
ccd0    128     none    /dev/ad0s1d /dev/ad2s1d
ccd1    128     none    /dev/ad0s1e /dev/ad2s1e
ccd2    128     none    /dev/ad0s1f /dev/ad2s1f
```

После этого запустить ту же утилиту конфигурации следующим образом:

```
$ ccdconfig -C
```

в результате чего все необходимые сведения будут взяты из /etc/ccd.conf (если конфигу дали другое имя, следует явно указать его, как аргумент с полным путем), и устройства благополучно созданы.

Сбросить уже имеющиеся настройки ccd (если они почему-либо не устраивают) можно командой

```
$ ccdconfig -U
```

после чего они безболезненно переконфигурируются заново.

Дальнейшее обращение с конкатенированными массивами происходит точно так же, как и с обычными партициями: их нужно разметить на BSD-разделы, в которых создаются файловые системы, монтируемые в целевые каталоги. За одним исключением: прибегнуть к помощи sysinstall или BSD Installer теперь уже точно не удастся, так что вся надежда только на собственные руки.

Итак, для начала размечаем созданные массивы:

```
$ bsdlabel -w /dev/ccd0 auto
$ bsdlabel -w /dev/ccd1 auto
$ bsdlabel -w /dev/ccd2 auto
```

Это даст нам на каждом из них по партиции `c`, к использованию непригодной. И потому повторяем процедуру в ручном режиме:

```
$ bsdlabel -e /dev/ccd#
```

Как мы помним, это вызов текстового редактора для прямой модификации дисковой разметки. Копируем строку, описывающую партицию `c`, изменяем маркирующую ее литеру (например, на `d`) и тип раздела (с `unused` на `4.2BSD`). Необходимости указывать размеры блока, фрагмента и прочего нет, эти значения будут определены при создании файловой системы.

После создания файловых систем их остается только смонтировать, сначала во временные каталоги типа /mnt/var, mnt/usr и /mnt/home, затем переместить в них содержимое одноименных каталогов из корня, и, наконец, прописать монтирование конкатенированных разделов в /etc/fstab примерно таким образом:

```
/dev/ccd0e      /vaufs      rw,noatime
/dev/ccd1e      /usufs      rw,noatime
/dev/ccd2e      /home ufs   rw,noatime
```

После чего следует перезагрузка и радость от обретенных RAID. Причем с точки зрения быстродействия, радость эта будет вполне обоснованной: измерения показали прирост скорости на типичных файловых операциях для ccd-RAID составляет от 10 до 50% против одиночного диска.

Теперь остается сказать несколько слов об инструментарии для реализации технологии LVM в Linux.

Для использования LVM требуется поддержка ядром системы. Это достигается включением двух опций. При использовании `make menuconfig` в секции Multi-device support нужно включить, во-первых, поддержку Multiple devices, во-вторых — Device Mapping (это в ядре 2.6.X, в ядрах 2.4.X это называлось Logical volume manager (LVM) support).

Далее, дисковое пространство (на одном или нескольких накопителях) следует разметить в виде разделов с идентификатором Linux LVM (8e в шестнадцатеричном исчислении).

И, естественно, потребуется софт для работы с LVM. Он входит в пакет lvm (lvm2), в составе которого три группы утилит, предназначенных для работы с физическими томами (pv*), логическими группами (lg*) и логическими томами (lv*). Так, команда `pvcreate` создает физические тома, команда `pvscan` сообщает об наличествующих, а команда `pvdisplay` выводит о них полную информацию. А тройки команд `vgcreate`, `vgscan`, `vgdisplay` и `lvcreate`, `lvscan`, `lvdisplay` проделывают то же для групп томов и логических томов соответственно.

Как и положено уважающим себя UNIX-программам, все компоненты пакета хорошо документированы, и с их опциями можно ознакомиться на стандартных man- и info-страницах. Ранее упомянутый сайт (**http://tldp.org/HOWTO/LVM-HOWTO.html**) содержит много информации по практическому применению этих команд.

Далее все требуемые действия могут быть расписаны по шагам.

1. Первый из них — запуск команды

   `$ vgscan`,

   которая отыщет все имеющиеся в наличии потенциальные физические тома, то есть разделы типа 8e и создаст необходимые файлы конфигурации — /etc/lvmtab и /etc/lvmtab.d, о чем любезно нас проинформирует.

2. Второй шаг — собственно превращение дискового раздела Linux LVM в физический том командой

   `$ pvcreate /dev/hda5`

   После этого командой `pvscan` можно проверить, что система думает о наших томах. В ответ она сообщит путь к каждому из файлов устройств физических томов, их объем и степень занятости (пока, разумеется, нулевую), а также состояние (на данном этапе не активное).

3. Третий шаг — создание из физических томов логической группы (Volume Group). Для этого потребуется команда `vgcreate` с именем группы в качестве первого аргумента и имени файла устройства раздела, как аргумента второго.

   Имя группы произвольно, в путях к файлам устройств физических томов при использовании devfs должна применяться полная нотация (как это вывела команда `pvscan`). По умолчанию тома нарезаются на физические блоки — extents размером 4 Мбайт. При желании иметь другой размер блока tuo можно явно задать опцией

   `-s ##m`

   Резонные люди рекомендуют использовать extents в 32 Мбайт. То есть требуемая команда будет иметь вид вроде

   `$ vgcreate -s 32m all /dev/ide/host0/bus0/target0/part5`

   В этом случае максимальный размер любого из будущих логических томов ограничивается фантастической (пока) величиной 2 Тбайт. Если же остановиться на extent по умолчанию, предел тома составил бы всего-навсего 256 Гбайт. Именно всего-навсего, согласитесь, ныне эта величина не кажется недостижимой.

   Теперь можно выполнить команду `pvscan`, дабы убедиться в результате. Она, кроме прежней информации, сообщит нам, что физические тома активированы и включены в группу.

А полную информацию о вновь созданной группе мы получим командой `vgdisplay`, которая, в числе прочего, сообщит нам имя группы, режим доступа к ней и ряд других данных:

```
--- Volume group ---
VG Name                 all
VG Access               read/write
VG Status               available/resizable
VG #                    0
MAX LV                  256
...
MAX LV Size             2 TB
Max PV                  256
Cur PV                  1
Act PV                  1
VG Size                 73 GB
PE Size                 32 MB
Total PE                2336
...
VG UUID                 TiYr9D-5Ub5-ordV-Zcv6-A7Eg-AIqD-1ALFe6
```

4. Четвертый шаг — собственно создание логического тома или томов, по желанию. Это — некий аналог нарезания на разделы физического жесткого диска. И осуществляется он командой `lvcreate`, для которой в качестве опций нужно указать размер тома и его имя (`-L` и `-n` соответственно), а аргументом — имя ранее созданной группы. Размер тома можно указывать в гигабайтах. И очевидно из названия, что под логический том можно отвести как весь объем группы, так и ее часть, в последнем случае у нас останется место и для других томов. Например, для создания томов под обычно обособляемые файловые системы последовательность команд примет такой вид:

```
$ lvcreate -L 10G -n lvusr all && \
    lvcreate -L 2G -n lvar all && \
    lvcreate -L 2G -n lvopt all && \
    lvcreate -L 55G -n lvhome all && \
    lvcreate -L 2G -n lvtmp all
```

Она создаст нам тома под все намеченные к использованию файловые системы. Имена томов, разумеется, произвольны и даны просто в удобном виде.

Замечу, что приведенной последовательностью команд создаются логические тома с линейным соответствием физических и логических extents.

Предписать использование схемы чередования (striped) можно дополнительной опцией `-i`. А опция `-I [значение]` задаст размер чередующихся блоков (в килобайтах). Однако это имеет смысл только при наличии двух физических дисков (да еще, как неоднократно подчеркивалось ранее, на отдельных IDE-каналах).

Теперь выполним проверку: командой `lvscan` отыскиваем все новообразованные логические тома, узнаем пути к файлам соответствующих им устройств, их размеры и убеждаемся в том, что тома (прекрасный каламбур, господа?) активированы

```
lvscan -- ACTIVE            "/dev/all/lvusr" [10 GB]
lvscan -- ACTIVE            "/dev/all/lvar" [2 GB]
lvscan -- ACTIVE            "/dev/all/lvopt" [2 GB]
lvscan -- ACTIVE            "/dev/all/lvhome" [55 GB]
lvscan -- ACTIVE            "/dev/all/lvtmp" [2 GB]
lvscan -- 5 logical volumes with 71 GB total in 1 volume group
lvscan -- 5 active logical volumes
```

А командой

```
$ lvdisplay /dev/all/lv*
```

получаем о каждом томе всю информацию, которую в силах предоставить нам система, например:

```
$ lvdisplay /dev/all/lvhome
--- Logical volume ---
LV Name                /dev/all/lvhome
VG Name                all
LV Write Access        read/write
LV Status              available
LV #                   4
# open                 1
LV Size                55 GB
Current LE             1760
Allocated LE           1760
Allocation             next free
Read ahead sectors     1024
Block device           58:3
```

5. Остается пятый, последний, шаг: создание на каждом логическом томе файловой системы и ее монтирование, но тут уж никакой специфики нет.

Только что был описан пример для случая организации логических томов на одном диске. На двух и более — ничуть не сложнее. Придется только повторить команду `pvcreate` должное число раз, для создания на каждом из физических томов (Physical Volume):

```
$ pvcreate /dev/hda5
$ pvcreate /dev/hdc5
...
```

А при объединении их в логическую группу указать в качестве аргументов каждый из физических томов:

```
$ vgcreate имя_группы /dev/hda5 /dev/hdc5
```

При создании же собственно логического тома с помощью опций `-i` и `-I` можно попытаться повысить производительность дисковых операций, команда вроде

```
$ lvcreate -i 2 -I 8 -L 60G -n lvhome имя_группы
```

создаст схему соответствия stripped между физическими и логическими extents, то есть иллюзию чередования записи на два физических диска блоками (chunks) по 8 Кбайт.

Создание файловых систем

Linux поддерживает в качестве нативных несколько файловых систем, и потому для работы с каждой из них существует собственный набор инструментов: Ext2fsprogs — для файловых систем Ext2fs и Ext3fs, reiserfsprogs, xfsprogs и jfsutils — для файловых систем ReiserFS, XFS и JFS соответственно. Каждая из этих файловых систем может быть создана командой `mkfs` с опцией `-t`, значением которой выступает желаемый тип файловой системы, и аргументом — именем файла устройства (дискового раздела).

Однако сама по себе команда `mkfs` — лишь оболочка, служащая для вызова специфичной команды, создающей файловую систему определенного типа:

- `mk2efs` или `mkfs.ext2` — Ext2fs или Ext3fs (в последнем случае обе они требуют опции `-j`, предписывающей создать журнал);
- `mkreiserfs` — ReiserFS;
- `mkfs.xfs` — XFS;
- `mkfs.jfs` — JFS.

Выглядит это примерно так. Команда

```
$ mke2fs /dev/hda1
```

создаст файловую систему Ext2fs на первичном разделе /dev/hda1. Команда

```
$ mke2fs -j /dev/hda3
```

создаст файловую систему Ext3fs на первичном разделе /dev/hda3. Опция `-j` предписывает добавление журнала), в этом случае новая файловая система получит некоторые параметры журналирования. Определить их вручную позволяет следующая форма этой команды:

```
$ /sbin/mke2fs -J опции_журналирования /dev/hd?#
```

Возможные значения опций журналирования:

- `size=размер`, задающая объем журнального файла в мегабайтах;
- `device=внешний_журнал` для подключения новой файловой системы к журналу, ранее созданному на другом дисковом разделе;
- `data=режим`, определяющая режим журналирования. Значения последней опции могут быть такими:
  - `journal` (полное журналирование, при котором фиксируются операции не только с метаданными файлов, но и их данными);
  - `ordered` (по умолчанию), при которой данные и метаданные группируются в единый блок транзакции;
  - `writeback`, в котором никакого журналирования данных не осуществляется.

Можно использовать и специальную команду `/sbin/mkfs.ext3`, возможности ее идентичны таковым `/sbin/mke2fs` (ибо она не что иное, как символическая на нее ссылка). Но самое интересное — возможность преобразования существующей Ext2fs в Ext3fs простым добавлением журнала, не только без потери данных, но и без перезапуска системы (и даже без размонтирования соответствующего устройства). Делается это командой

```
$ tune2fs -j /dev/hd?#
```

Она просто добавляет файл журнала /.journal в корневой каталог модифицируемой файловой системы (если последняя не была размонтирована) или задействует для журнала скрытый inode (если перед модификацией файловая система была размонтирована). Добавлю, что обратное преобразование еще проще и осуществляется перемонтированием файловой системы.

Файловая система ReiserFS создается специально предназначенной для этого командой `/sbin/mkreiserfs` из пакета reiserfsprogs. Для нее доступны многочисленные опции (`-s` для задания размера журнала, `-f` для принудительного переформатирования ранее существовавшей файловой системы иного типа и т. д.), с которыми можно ознакомиться посредством man (8) mkreiserfs.

Для создания XFS также существует собственная команда `mkfs.xfs` (из пакета xfsprogs). В ней предусмотрено несколько опций, каждая из которых имеет ряд субопций, принимающих численные значения:

- `-b`, которая посредством субопции `size=##` позволяет задать размер блока данных в байтах, который должен быть кратен размеру страницы оперативной памяти (для платформы i386 4 Кбайт) и может варьировать в диапазоне от 512 до 65 536 (по умолчанию 4096);
- `-d`, определяющая параметры области данных файловой системы, такие как количество самостоятельных областей раздела (Allocation groups, субопция `agcount`) или, напротив, их размер (субопция `agsize`);
- `-l`, специфицирующая параметры журнального файла, например, его размер (субопция `size`).

При использовании `mkfs.xfs` для достижения максимальной производительности рекомендуется в явном виде задать количество allocation groups, иначе оно будет определяться автоматически, что ведет к непроизводительным расходам ресурсов. Это делается, исходя из эмпирического расчета: одна allocation group на 4 Гбайт дискового пространства. Далее можно установить размер файла журнала, здесь рекомендованное значение составляет 32 Мбайт. То есть для дискового раздела объемом в 20 Гбайт команда приобретет вид

```
$ mkfs.xfs -d agcount=5 -l size=32m /dev/hda1
```

Кроме всех перечисленных, команда `mkfs.xfs` имеет опцию `-f` принудительного создания файловой системы XFS поверх любой существующей. Ее достаточно, если последняя была Ext2fs или Ext3fs. Если же XFS создается поверх ReiserFS, после этого возможны ошибки при монтировании новой файловой системы. Впрочем, то же относится и к обратной процедуре (замене XFS на ReiserFS). Это связано с тем, что команда монтирования может распознать созданную XFS как дефектную ReiserFS, и наоборот.

Во избежание этого перед таким замещением приходится прибегать к несколько шаманскому приему — обнулению начальных областей раздела (хранящего метаданные файловой системы) командой

```
$ dd if=/dev/zero of=/dev/hd?#
```

Ждать заполнения нулями всего устройства не обязательно, достаточно дать этой команде поработать секунд 10—20, после чего прервать ее комбинацией клавиш <Ctrl>+<D> и перейти к созданию новых файловых систем.

Наконец, и JFS имеет специфическую команду для собственного создания, которая выглялит примерно так:

```
$ mkfs.jfs /dev/hda7
```

Впрочем, тут за подробностями милости просим к страницам документации, опыта работы с JFS у меня нет.

Процесс создания файловой системы в BSD сводится к выделению суперблока и записи общих параметров файловой системы, созданию таблицы inodes (в UFS и UFS2, в отличие от некоторых современных файловых систем для Linux, все inodes создаются раз и навсегда, а не выделяются динамически, по мере надобности) и разметке блоков в области данных. И за все это отвечает одна-единственная программа, именуемая newfs. Она требует единственного аргумента — имени файла форматируемой партиции, например,

```
$ newfs /dev/ad0s1a
```

После этого все базовые свойства файловой системы будут определены по умолчанию. Однако пользователь в силах влиять на них с помощью опций команды `newfs`, задаваемых в командной строке или раз и навсегда глобально, через пункт **Options** окна программы sysinstall. Важнейшие из этих опций мы и рассмотрим.

Опция `-b` определяет размер логического блока файловой системы. Минимальный размер его — 4096 байтов, максимальный — 65 536. Однако, как говорят, при максимально возможном размере возможны всякие неприятности, и потому надежным считается верхнее ограничение в 32 768 байта. А с точки зрения здравого смысла, значение в 16 384 байтов представляется в большинстве случаев разумным.

Опция `-f` устанавливает размер фрагмента блока. Рекомендуется определить его в 1/8 размера блока, что по умолчанию и составит 2048 байта. Значения в 1/4 или 1/2 блока также допустимы, но очень не рекомендуются в документации.

Опция `-i` очень важна, она-то косвенно и устанавливает количество записей в индексной таблице (то есть максимальное количество файлов в файловой системе). Значение этой опции дается в байтах, отводимых на элемент индексной таблицы, и должно быть кратным размеру блокового фрагмента. Значение по умолчанию — учетверенный объем фрагмента, что определяет средний прогнозируемый размер файла (8192 байтов). А максимальное количество файлов в конкретной файловой системе легко рассчитать, исходя из объема отведенной под нее партиции.

Еще одна интересная опция `-m`, значение которой указывается в процентах от суммарного объема дискового пространства, отведенного на партицию. И представляет собой объем, резервируемый от записи обычными пользователями (но не root, тот всегда имеет возможность записать свои действия на диск при наличии физически свободного пространства). Он определяется потому, что быстродействие файловых операций в UFS просто катастрофи-

чески падает, когда количество свободных блоков в области данных близко к исчерпанию (проверено на практике). И потому некий объем файловой системы резервируется принудительно (по умолчанию 8%).

С этой опцией связана еще одна (`-o`), которая определяет алгоритм выделения свободных блоков данных при создании новых файлов. Дело в том, что UFS в состоянии размещать их двумя способами. Первый — методом плотной упаковки с целью минимизации внутренней фрагментации и экономии дискового пространства. И называется он оптимизацией по объему (опция `-o` принимает значение `space`). Второй же метод (`-o time`) обеспечивает быстрейшее выделение свободных блоков с целью увеличения скорости создания файлов. Так вот, значение `-o` по умолчанию коррелирует со значением `-m`: если больше или равно 8%, применяется оптимизация по времени, если меньше — по объему. Хотя явным образом можно указать прямо противоположные значения.

Вообще-то, все приведенные опции очень важны, интересны и полезны для общего образования, однако пользователю вряд ли придется к ним прибегать: их значения по умолчанию, как и почти все в BSD-системах, разумны и приемлемы в подавляющем большинстве случаев. А вот опция `-U` при запуске команды `newfs` по умолчанию не задействуется. Обеспечивает же она поддержку того самого механизма Soft Updates, который (парадоксально, но правда) способствует как повышению быстродействия файловых операций, так и устойчивости файловой системы.

Монтирование

Для монтирования файловых систем любых типов в Linux предназначена универсальная утилита mount. В общем случае она требует двух аргументов: имени монтируемого устройства и точки монтирования. Например, команда

```
$ mount /dev/hda1 /boot
```

смонтирует ранее созданную файловую систему в каталог /boot.

Команда `mount`, как правило, безошибочно распознает поддерживаемые типы файловых систем (а в их числе все нативные файловые системы Linux, все вариации на тему FAT и файловые системы CD- и DVD-дисков). Если же по каким-то причинам она делать это отказывается, тип файловой системы нужно задать в явном виде с помощью опции `-t`, например:

```
$ mount -t vfat /dev/sda /mnt/usb
```

для монтирования USB-накопителя с файловой системой VFAT (вариант FAT, поддерживающий длинные имена и принятый в Windows, начиная с версии 95).

Команда `mount` в Linux имеет много опций. Она позволяет смонтировать одну и ту же файловую систему в разные каталоги с помощью опции `--bind`. Например, tmpfs таким образом может быть смонтирована не только в каталог /dev/shm (штатное ее место, с точки зрения стандарта POSIX), но и в каталог /tmp:

```
$ mount --bind tmpfs -t tmpfs /dev/shm ;
$ mount --bind tmpfs -t tmpfs /tmp
```

Как уже говорилось, файловая система tmpfs не нуждается в создании, для ее использования достаточно поддержки в ядре и самого факта монтирования.

Более того, в разные точки можно смонтировать и ветвь каталога, не представляющую отдельной файловой системы. Это целесообразно, например, для совместного использования репозитория исходных текстов разными Source Based дистрибутивами.

Наконец, в качестве файловой системы можно смонтировать и отдельный файл, например, ISO-образ CD-диска, для чего служит опция `--loop`.

Непременное требование: каталог, выступающий в качестве точки монтирования, должен уже существовать до отдачи команды `mount` (создать таковой она почему-то не в состоянии, это следует сделать заблаговременно командой `mkdir dirname`). И желательно, чтобы он был пуст: монтирование в каталог с файлами фатальных последствий нынче не влечет (раньше влекло, вплоть до общей паники системы), но все содержимое его станет недоступным вплоть до размонтирования файловой системы. Правда, и никуда не денется тоже.

Отдельно стоит предостеречь от монтирования tmpfs в каталог /tmp в ходе рабочего сеанса: это вызовет мгновенный крах некоторых программ, в частности — оконной системы Икс. Как же быть, если нужно задействовать tmpfs именно в этом качестве? — спросите вы меня. На это я отвечу под занавес.

BSD-инструментарий для монтирования файловых систем отличается тем, что здесь отсутствует универсальная команда `mount`. Вернее, она есть, но не носит универсального характера, предназначаясь только для монтирования родных файловых систем — UFS и UFS2. Форма

```
$ mount /dev/ad#s#? /mount_point
```

только их и в состоянии смонтировать. Для монтирования чуждых файловых систем из числа поддерживаемых предназначены специальные команды:

- `mount_msdos` — для FAT-разделов;
- `mount_ext2` — для файловых систем Linux (только для Ext2fs и Ext3fs, причем в последней журналирование просто игнорируется);

- `mount_cd9660` — для компакт-дисков с файловой системой ISO 9660 (и ее вариаций с расширениями RockRidge и Joliet). Тот же результат может быть достигнут и командой `mount -t type_fs`, но это лишь опосредованный вызов специальной команды для данного типа файловой системы.

Все варианты команды `mount` в BSD существенно беднее своей Linux-коллеги по возможностям. Здесь не допускается ни раздельное монтирование одной и той же файловой системы в разные точки, ни монтирование ветвей файловой системы. А монтирование файла как устройства требует некоторых дополнительных телодвижений.

Выше был описан процесс ручного монтирования файловых систем и в Linux, и в BSD. И действительно, к ручному монтированию подчас прибегать приходится. Однако постоянно используемые файловые системы (такие как /usr, /home, не говоря уже о корне) целесообразно монтировать автоматически в ходе загрузки машины. За этот процесс отвечают соответствующие системы инициализации. А вот то, что должно монтироваться, описывается в конфигурационном файле /etc/fstab.

Формат его одинаков во всех POSIX-системах, представляя собой простую базу данных, строки которой описывают каждое требующее монтирования устройство, а поля — следующие (разделитель полей — пробел или пробелы в любом количестве, а также символы табуляции):

- имя файла устройства (в Linux с использованием devfs его лучше указывать соответственно номенклатуре файловой системы устройств);
- точка монтирования;
- тип файловой системы;
- опции монтирования (их много, по умолчанию в большинстве случаев стоит значение `default`);
- `dump` и `pass` — разрешают/запрещают резервное копирование (дамп) файловой системы и ее проверку; реальное значение нынче почти потеряли.

Указание в поле опций монтирования значения `default` подразумевает, в том числе, и то, что файловая система будет смонтирована автоматически при старте системы. Однако это не подходит для сменных носителей, каковых в момент загрузки может не быть в приводе. И потому в строках, описывающих CD-ROM, USB-драйвы и тому подобные устройства, значение этого поля должно указываться как `noauto`. А сама по себе запись в /etc/fstab для них призвана упростить процедуру монтирования: в качестве аргумента команда `mount` потребует только имя каталога — точки монтирования.

Для пояснения всего сказанного приведу свой файл /etc/fstab для системы Archlinux:

```
/dev/discs/disc0/part1 / reiserfs notail,noatime,nodiratime 0 0
/dev/discs/disc0/part3 /var reiserfs notail,noatime,nodiratime 0 0
/dev/discs/disc0/part4 /home reiserfs notail,noatime,nodiratime 0 0
```

Можно видеть, что в поле опций монтирования у меня приведено довольно много значений (через запятую, без пробела). К ним мы еще вернемся, нынче отмечу лишь, что опции `noatime`, `nodiratime` запрещают обновление того самого временного атрибута — времени последнего доступа, о котором говорилось ранее, и призваны повысить производительность файловых операций.

По завершении использования файловой системы она в обязательном порядке подлежит размонтированию. Для постоянно смонтированных файловых систем, описанных в /etc/fstab, это делается автоматически при перезагрузке системы (или подготовке ее к выключению).

А вот файловые системы на сменных носителях перед извлечением должны быть размонтированы руками. Для чего служит команда `umount` с единственным аргументом — точкой монтирования (или именем файла устройства, без разницы, но не обоих вместе). То есть любая из команд типа

```
$ umount /mnt/cd
```

или

```
$ umount /dev/cdroms/cdrom0
```

одинаково успешно размонтирует файловую систему на диске в приводе, без чего извлечь его оттуда можно только с помощью скрепки.

По умолчанию монтирование и размонтирование файловых систем — прерогатива исключительно root. Однако часто целесообразно разрешить эту операцию и обычному пользователю, по крайней мере в отношении сменных носителей. В Linux это достигается просто указанием значения `user` в поле опций монтирования соответствующей строки файла /etc/fstab, например:

```
/dev/cdroms/cdrom0 /mnt/cd iso9660 ro,user,noauto,unhide 0 0
```

В BSD — несколько сложнее, нужно в принципе разрешить монтирование устройств пользователю, что делается директивой

```
$ sysctl vfs.usermount=1
```

Затем требуется изменение прав доступа к соответствующим устройствам, что предлагается в качестве домашнего задания.

И в заключение — об активизации свопинга. Этот раздел не несет в себе никакой файловой системы и потому в монтировании не нуждается. Однако

его качество должно быть задано явным образом специальной командой, например:

```
mkswap /dev/hda2
```

Здесь нужно быть внимательным: применение этой команды к разделу с файловой системой приведет к уничтожению последней (и, соответственно, всех ее данных).

В монтировании swap-раздел также не нуждается, вместо этого его нужно активизировать:

```
$ swapon /dev/hda3
```

При старте машины активация swap происходит автоматически при наличии в файле /etc/fstab строки вида:

```
/dev/discs/disc0/part2 swap swap defaults 0 0
```

При наличии в машине двух дисков область свопинга целесообразно разделить между ними, это способствует быстродействию. Для чего в /etc/fstab вносится уже две строки:

```
/dev/discs/disc0/part2 none swap sw,pri=1 0 0
/dev/discs/disc1/part1 none swap sw,pri=1 0 0
```

Здесь важна опция `pri`, значением которой выступает приоритет swap-раздела. Причем сама по себе цифра роли не играет, лишь бы приоритеты для swap-разделов обоих дисков были равны.

Обратной процедуре (дезактивации области свопинга) служит команда

```
$ swapoff /dev/hda3
```

Она также выполняется автоматически при перезагрузке или подготовке к выключению.

Дополнительные утилиты

В заключение этой интермедии стоит сказать несколько слов об утилитах, имеющих отношение к теме дисковой разметки и файловых систем. А именно, о командах `df` и `du`, позволяющих получить информацию о свободном и используемом дисковом пространстве. Обе они универсальны для всех POSIX-совместимых ОС, в базовый комплект которых входят всегда (начиная с Version 1 AT&T UNIX).

Начнем с `df` (от disk free). Данная без опций и аргументов, она выведет информацию об использовании всех смонтированных в данный момент файловых системах в такой форме:

```
$ df
Filesystem  512-blocks     Used    Avail Capacity  Mounted on
```

```
/dev/ad0s1a     508126    92666   374810    20%    /
/dev/mfs71       15806     2692    11850    19%    /tmp
/dev/ad0s1d     508126    34788   432688     7%    /var
/dev/ad0s1e   21553172  4542548 15286372    23%    /usr
/dev/ad0s2c   51936140 24257440 23523812    51%    /home
procfs                8        8        0   100%    /proc
```

Форма вывода достаточно прозрачна: сначала идет имя файла смонтированного устройства (дискового раздела — в примере для DragonFlyBSD), далее их размер в 512-байтных (то есть физических) блоках, затем объем использованного и доступного дискового пространства, процент занятости и, наконец, точка монтирования.

Аргументами команды `df` могут быть, с одной стороны, имена файлов устройств (диска целиком, дискового раздела, слайса, подраздела, в зависимости от системы), с другой — точка монтирования интересующей файловой системы. В любом случае вывод будет содержать те же сведения об объектах, перечисленных в качестве аргументов (их может быть сколько угодно).

А вот форма вывода определяется опциями команды `df`. Так, в качестве единиц измерения дискового пространства могут быть представлены не только физические блоки, но также килобайты, мегабайты или гигабайты: для этого потребуется указание опций `-k`, `-m` или `-g` соответственно. Опция `-h` предписывает вывод в тех единицах, которые подходят по смыслу (то есть для 10-гигабайтного раздела это будет `G`, для 50-мегабайтного — `M`).

Добавление опции `-i` даст при выводе еще и информацию о количестве занятых и свободных inodes, то есть числе теоретически возможных в данной файловой системе файлов. Опция `-l` ограничит вывод только локальными файловыми системами (без учета смонтированных в сети). А опция `-t`, значением которой выступает тип файловой системы, выведет только сведения о "заказанных ФС".

Теперь о `du`. Как следует из ее названия (от disk usage), она выводит информацию о дисковом пространстве, занимаемом файлами каталога, указанного в качестве ее аргумента, включая объем содержимого всех вложенных подкаталогов. Выглядит это примерно так:

```
$ du ~/soft/
1010    soft/boot
679622  soft/
```

Не возбраняется в качестве аргумента и указание единичного файла:

```
$ du soft/pkgsrc.tar.gz
26608   soft/pkgsrc.tar.gz
```

По умолчанию du выводит результаты своей работы в физических блоках (512 байтов). Однако, как и в случае с df, это легко изменить с помощью опции -h, которая подберет подходящие единицы измерения:

```
$ du -h soft/
1010K   soft/boot
664M    soft/
```

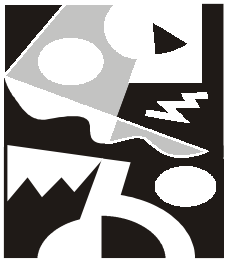
А опция -k обеспечит просто вывод в килобайтах.

Есть у команды du и другие опции. Так, опция -d ограничит подсчет уровнем вложенности подкаталогов, указанным в качестве ее значения, опция -I исключит из подсчета объема файл или каталог, выступающий ее значением.

Если в качестве аргумента команды du указать каталог, выступающий точкой монтирования самостоятельной файловой системы, ее вывод даст информацию о дисковом пространстве, занятом на соответствующем разделе, подобно колонке Used в выводе команды df. При этом результаты, полученные той или другой командой, отнюдь не обязаны совпадать. Это связано с различными принципами работы df и du, прямо вытекающими из их названий.

Команда df, как ей и положено по имени, подсчитывает суммарный объем блоков, помеченных в суперблоке файловых систем как свободные, du же, напротив, считает объем, занятый каждым файлом, исходя из описания в его метаданных. А поскольку файловые операции в той или иной форме всегда кэшируются, легко представить себе ситуацию, когда после удаления файлов (то есть исключения их имен из записей в каталогах) соответствующего освобождения в "таблице занятости" еще не произошло, и блоки данных удаленных файлов учтены при подсчете df, но в вывод du уже не попадут.

Глава 11



Терминалы, режимы, интерфейсы

- Апология консоли.
- Что такое терминал.
- Понятие виртуального терминала.
- О режимах.
- Об интерфейсах.

Рассказ о терминалах принято начинать с тех времен, когда машины были большими. Не отступлю от этой традиции и я. Но сначала несколько слов о необходимости этого рассказа вообще.

Апология консоли

Текстовая консоль с командным интерфейсом в век графических режимов и интегрированных объектных сред выглядит весьма скромно. Примерно как серый кремень на фоне своих блистающих сородичей — аметистов и цитринов, сердоликов и огненных опалов. Почему же в последнее время тема эта снова и снова обсуждается на страницах сетевых и печатных изданий? Для ответа на этот вопрос придется углубиться в историю, хотя и очень отдаленную в масштабах вечности.

Казалось бы, не так давно на ниве настольных систем сосуществовали, не всегда мирно, но почти на равных, и DOS во всех ее проявлениях (MS-DOS, IBM DOS и самая среди них прогрессивная и потому не вполне совместимая с прочими DR DOS), и Mac OS, и милая Amiga со своей multimedia-ориентацией, и Windows всякого рода, чина и номера, и OS/2, и почти забытая Geoworks (она же GEOS), не говоря уже о сияющем NextStep. А еще туда (почти) всерьез стремились и HP-PA с ее HP-UX, и Sparc со своими SunOS и Solaris, и казавшаяся недосягаемой по производительности Alpha с Digital UNIX.

Неожиданно оказалось, что параллельно миру коммерческого софта существует другой мир — свободного софта и открытых исходников. Где программы пишутся для решения своих личных задач, для самообразования и поддержания профессиональной формы, для повышения престижа в собственном сообществе, а то и просто из любви к искусству и спортивного азарта. Мир этот был почти столь же стар, как и мир коммерческий. Однако долгое время развивался он тихо и неприметно, как млекопитающие в эру динозавров. И лишь недавно (и также исторически мгновенно) об этом мире узнали широкие круги компьютерной общественности. Началось явление, которое войдет в историю ушедшего тысячелетия под названием бума Linux и открытых исходников.

Разумеется, Linux не был ни единственной, ни первой, ни даже наиболее развитой системой, распространяемой свободно и открыто. Одновременно и рядом существовали и Free-, и Net-, и OpenBSD, и недавно реанимированный Hurd, если ограничиться только UNIX-подобными системами. К каковым мир свободных ОС отнюдь не сводится: время от времени появлялись и свободные операционные системы, отношение которых к UNIX было весьма опосредованным (например, феноменальная AtheOS, ныне преобразованная в Syllable).

Однако именно Linux, в силу ряда исторических, психологических и просто случайных причин, привлек к себе внимание широких масс околокомпьютерных трудящихся. Обсуждение этих причин далеко выходит за рамки сегодняшней темы. Рискну сформулировать только одну, на мой взгляд, главную: если бы Linux не существовал, его следовало бы выдумать.

В человеческой популяции всегда были, есть и (надеюсь) будут люди, которым рано или поздно становится скучно в мире, пусть удобном и уютном, но однообразном. И именно эта категория людей неожиданно для себя видит, что за "окнами" существует жизнь, не столь уютная и приглаженная, но загадочная и порой опасная (если не для жизни, то для "железа"). Другая, и уже этим интересная. На мой взгляд, далеко не случайно, что Linux-бум начался именно в тот момент, когда засилье Windows казалось безграничным...

На текущий же момент мы наблюдаем беспрецедентный рост сообщества пользователей Linux (и прочих свободных POSIX-систем). И новые пользователи Linux сейчас количественно резко преобладают над юниксоидами старого закала и энтузиастами Linux первого призыва. Однако выросло поколение пользователей (выросло — как пользователи, не обязательно как организмы), лихо налагающих фильтры в Photoshop или вращающих городские кварталы в 3D Max, но не имеющих представления не только о разбиении диска, но даже не заставших командной строки DOS.

Это я отнюдь не в упрек: способность ряда моих знакомых применять Photoshop для анализа снимков или 3D Max — для построения архитектурных ансамблей вызывает у меня просто искреннее восхищение. Скорее это

комплимент искусству Windows драпировать свои внутренности. Результат — естественное стремление Windows-мигранта и в Linux действовать в привычной среде и привычными методами. Благо, стремление это удовлетворяется интегрированными объектными средами графического режима, такими как KDE и GNOME. Да и современные дистрибутивы к тому подталкивают.

Однако, как я пытался показать в *главе 4*, довольно быстро выясняется, что приемы работы, заимствованные из Windows, в Linux часто оказываются менее эффективными, чем традиционные инструменты UNIX-систем. А большинство их не требует для своей работы ничего, кроме классической UNIX-консоли, именуемой также терминалом.

Что такое терминал

А теперь обратимся ко временам более древним, "когда компьютеры были большими". Размером не то что с самогонный аппарат, а с целый перегонный цех. А также очень дорогими, как сами по себе, так и в эксплуатации. Счастливым обладателям таких машин казалось непозволительной роскошью оставлять их в индивидуальном пользовании. И потому эти машины были многопользовательскими.

Устройствами ввода при этом служили перфокарты, а вывод подавался на печатающие устройства. Потом в качестве средств взаимодействия с машиной (и хранимыми в ней программами) стали использоваться клавиатуры, заимствованные у пишущих машинок, и дисплеи, или мониторы, созданные по аналогии с телевизором.

В результате появилась возможность оснастить рабочее место каждого пользователя сочетанием устройства ввода, с которого пользователь вводил свою задачу, и устройства вывода, на котором он мог просмотреть результат ее обсчета. Это сочетание и получило название терминала. А для того, чтобы пользователи могли проделывать это одновременно, родилось понятие систем разделения времени (строго говоря, ресурсов машины вообще, ныне это чаще называют истинной многозадачностью), одной из реализаций которых и была UNIX. Таким образом, понятие машинного времени утратило смысл, а все терминалы стали равны между собой. Как, впрочем, и пользователи, власть их была ограничена, каждый управлял только своим терминалом и не имел (теоретически) никакой возможности повлиять на систему в целом.

Однако имелся среди пользователей один умник. Он назывался root-оператором (или просто root), обладал всевластием в масштабе данной системы и реализовывал это всевластие с собственного всевластного терминала. Этот терминал всевластия получил название системной консоли.

В это же примерно время появились устройства хранения информации — винчестеры, названные так по аналогии маркировки первых их представителей с номенклатурой патрона для винтовки системы Генри 30-го калибра — 0,03 дюйма (они же три линии или 7,62 мм по-нашему). На винчестерах, наряду с общесистемными программами, нашлось место и для пользовательских данных. Однако пользователей было много, а машина с винчестером одна. И чтобы пользовательские данные не путались между собою, потребовалось разграничить их друг от друга. Что и проделывалось с одного из терминалов, который был равнее других. Им была та самая системная консоль, или просто консоль, в первоначальном смысле этого термина.

Физически консоль представляла собой точно такой же терминал, как и все остальные, а ее большая равность определялась исключительно полномочиями лица, за ней сидящего. Ибо root, от которого его всевластие и зависело, мог засадить в систему все хитрости свои, все меры защиты и все такое. В частности, запрещение авторизоваться как root с любого другого терминала, кроме консоли. А для управления пользователями он делал так, чтобы на консоль выводились все системные сообщения (в том числе сообщения об ошибках и информация об авторизации пользователей), почему консоль root и получила название системной.

Потом PC уравняла пользователей в правах. За каждой такой машиной сидел один-единственный пользователь, и была она самодостаточным агрегатом, имеющим собственные устройства ввода, вывода и хранения информации. Тем не менее, UNIX, мигрировав на персоналки, сохранил свою многозадачную и многопользовательскую природу. Однако если многозадачность легко реализовывалась программными средствами, то как реализовать доступ нескольких пользователей (а UNIX и на персоналке почти всегда имеет минимум двух пользователей — root и обычного) к машине, физически имеющей лишь одну комбинацию устройств ввода/вывода?

Понятие виртуального терминала

Конечно, и к персоналке в принципе можно прикрутить еще один монитор и клавиатуру — и приспособления для этого имелись. Однако более простым оказалось чисто программное решение. Так, в UNIX появилось понятие виртуальной консоли (или виртуального терминала).

Терминологическое отступление

Как уже было сказано ранее, консоль и терминал в историческом аспекте были понятиями разными. Однако различия между ними почти стерлись, и потому далее я буду использовать эти термины как синонимы. Есть и еще один вид виртуального терминала — так называемый X-терминал, реализуемый посредством специального класса программ, их договоримся называть графическими эмуляторами терминала, и речь о них будет в следующем разделе.

Виртуальный терминал — это все то же сочетание реальной клавиатуры и дисплея, которое при определенных условиях может выступать как самостоятельная машина. На каждом из наличных виртуальных терминалов может зарегистрироваться отдельный пользователь, и в его силах запускать там свои задачи, никак не влияющие на задачи пользователя, зарегистрировавшегося на другом виртуальном терминале.

Сам факт существования виртуальных терминалов, их максимальное количество и базовые свойства (такие как экранный буфер, возможность изменения экранных шрифтов, раскладок клавиатуры, цветовой гаммы и т. д.) определяются частью ядра системы, которая называется консольным драйвером. В Linux он так и называется: драйвер Linux-консоли. А ядра BSD-систем поддерживают возможность использования одного из нескольких консольных драйверов. Так, во FreeBSD и DragonFly по умолчанию (и сообразно здравому смыслу) в качестве драйвера системной консоли используется syscons, в Net- и OpenBSD эту роль нынче играет wscons. Однако все три системы могут работать также с консольным драйвером pcvt; другое дело, что это не дает никаких преимуществ (и даже наоборот, скажем, русификация pcvt представляет собой занятие для садомазохистов).

Непосредственное управление свойствами терминала (например, загрузкой конкретного шрифта или клавиатурной раскладки) осуществляет некий набор утилит, объединяемый в определенный (специфичный для данной ОС) программный пакет. Во FreeBSD (и DragonFly) он фактически безальтернативен и носит то же имя, что и консольный драйвер — syscons. В Linux практически равноправно можно использовать одну из альтернатив: пакеты kbd или console-tools. До некоторого времени последний считался более продвинутым, однако ныне они идентичны по своим возможностям. И приверженность разработчиков того или иного дистрибутива одному из этих пакетов обусловлена исключительно личными пристрастиями или историческими причинами.

Как уже неоднократно говорилось, все, что существует в UNIX-системе статически — суть файлы, в том числе физические или виртуальные устройства. И терминалы тут не исключение, каждому из них соответствует свой файл в каталоге /dev. Так, физической консоли соответствует файл /dev/console, номенклатура же файлов виртуальных терминалов различна в разных ОС. Во FreeBSD и DragonFly имена их файлов имеют вид /dev/ttyv#, где # — порядковый номер терминала, начиная с нуля. В Linux (без использования devfs) файлы виртуальных терминалов именуются как /dev/tty#, причем # начинается с единицы. Файл с именем /dev/tty0 тоже есть, но он зарезервирован за той самой системной консолью, которая представляет собой явление почти мифическое. При devfs файлы устройств виртуальных терминалов приобретут вид /dev/vc/#, где к номеру приложимо все сказанное. Хотя и в этом случае что-нибудь вроде tty# в каталоге /dev имеется на предмет обратной совместимости (это зависит от настроек демона devfsd).

Насколько мне известно, консольные драйверы POSIX-систем поддерживают до 63 виртуальных терминалов. Это связано с тем, что терминалы, как и всякие другие файлы устройств, характеризуются своими номерами: старшим (major) и младшими (minor). Старший номер класса терминальных устройств зависит от ОС, а под младшие номера зарезервированы числа с 1 до 63. Этим и определяется максимально возможное число консолей.

Принципиальная возможность существования 63 виртуальных терминалов не означает, однако, будто бы все они на самом деле доступны пользователю. Начать с того, что виртуальный терминал требует активизации. Для этого на нем должен быть запущен какой-либо процесс. При старте системы такие процессы (команды семейства `getty`) запускаются для некоторого количества терминалов. В большинстве дистрибутивов Linux традиционно активизируется 6 виртуальных консолей, в последних версиях FreeBSD и в DragonFly — 8. Эти числа определены в конфигурационных файлах /etc/inittab и /etc/ttys соответственно, и могут быть изменены в любую сторону. Консоли сверх умолчального количества могут быть активизированы и после загрузки системы. Для этого достаточно запустить на них какой-либо процесс. Им может быть:

- процесс на тему `getty`, вызывающий приглашение к авторизации;
- запуск программы автоматической регистрации пользователя типа qlogin;
- запуск сеанса работы оконной системы X (*подробности в следующем разделе*);
- запуск графической программы, обеспечиваемой библиотекой SVGAlib;
- непосредственный запуск (почти) произвольной программы с помощью команды `openvt`.

Любым из указанных способов в Linux можно открыть виртуальные терминалы с 7 по 63. Во FreeBSD есть еще одно ограничение — максимальное число консолей, поддерживаемое текущей конфигурацией ядра. По умолчанию оно равно 16, изменение требует реконфигурирования ядра и его пересборки.

Вообще-то, первое знакомство с механизмом виртуальных консолей Linux или FreeBSD обеспечивает незабываемое эмоциональное потрясение пользователю, немалую часть своей компьютерной жизни проведшему в окружении "черного DOS". С ним можно сравнить только восторг, испытываемый от возможностей командного интерпретатора, о нем пойдет речь в *главе 12*.

Потрясение это обусловлено рядом факторов. Во-первых, сама возможность открыть второй (третий, пятый) виртуальный терминал, авторизоваться в нем под своим пользовательским или иным именем (в том числе и root) и запустить любую программу, переключаясь по мере надобности между открытыми приложениями, ничуть не сложнее, чем между окнами в графической среде типа Windows.

Способ переключения между виртуальными терминалами зависит от настроек используемого консольного драйвера. В Linux и BSD syscons переключение выполняется комбинацией клавиш <Alt>+<F#>, где # — номер консоли. В wscons той же цели служит чуть более сложное сочетание <Ctrl>+<Alt>+<F#> (забегая вперед, отмечу, что она же используется во всех ОС для переключения из сеанса X в текстовую консоль).

Указанные комбинации (например, <Alt>+<F#>) для перехода между консолями зависят исключительно от текущей раскладки клавиатуры.

Легко сообразить, что означенным способом можно получить доступ к виртуальным терминалам с 1 по 12. А как быть, если вздумается установить большее их количество? Ведь более 12 функциональных клавиш на клавиатурах обычно не наблюдается...

Есть несколько способов доступа к виртуальным терминалам с произвольным номером. Например, в большинстве случаев комбинация клавиш <Alt>+<Shift>+<F#> обеспечивает переход к консолям с 13 по 24. Далее, в некоторых системах нажатием клавиши <PrintScreen> можно последовательно пролистать активные консоли, начиная с текущей (а в других случаях перейти в последнюю по счету из активизированных консолей). И, наконец, универсальный способ — команда

```
$ chvt #
```

где # — номер целевой консоли.

В Linux виртуальные терминалы абсолютно равноправны. В BSD-системах же сохранился рудимент представления о системной консоли, это первый виртуальный терминал. На него сыплются все сообщения о системных ошибках. И даже если это запретить соответствующей настройкой конфигурационных файлов, от кое-каких сообщений (например, о присоединении или отсоединении USB-устройств) избавиться нельзя никакими средствами (по крайней мере, я таких средств до сих пор не нашел).

Как говорилось в *главе 7*, каждый пользовательский интерактивный процесс (то есть, фигурально говоря, программа, запущенная пользователем с помощью командной директивы) связан с каким-либо виртуальным терминалом. При этом для программы различаются понятия управляющего и текущего терминала. Первый тот, с которого программа была запущена, и с которого можно при необходимости прервать ее исполнение. На текущий же терминал выводятся результаты ее работы. Обычно управляющий и текущий терминалы совпадают, однако это не обязательно. С помощью определенных средств (того же qlogin, например, или openvt), программу можно запустить так, чтобы она исполнялась на терминале, отличном от стартового. При этом последний сохраняет свои управляющие функции.

Еще одно различие выявляется между текущей консолью и всеми прочими при запуске какой-либо программы графического режима, например, Ик-

сов. При этом в первую очередь запускается вполне конкретная программа (X-сервер), и текущая консоль как бы блокируется вплоть до выхода из Иксов (или из SVGAlib-программы). А для запущенной с нее программы активизируется новая виртуальная консоль, которая и становится текущей.

Хотя на самом деле блокировки консоли, с которой запущены Иксы, не происходит. Просто она занимается процессом, который и выводит на нее массу своих служебных сообщений. Так что если запустить Иксы в фоновом режиме, а вывод сообщений куда-нибудь перенаправить (в log-файл или тот же /dev/null), то и эту консоль можно задействовать для нормальной работы.

X-сеанс, запущенный с какого-либо виртуального терминала, сохраняет его в качестве управляющего, то есть может быть с него же и прекращен, например, стандартной клавишной комбинацией <Ctrl>+<C>.

Отметим, что использующая графику программа, запущенная из консоли с поддержкой графического режима через frame buffer, занимает только текущий виртуальный терминал, который остается для нее и управляющим.

Второе открытие, подстерегающее пользователя, приобщающегося к POSIX'ивизму, — консольный буфер экрана (или, как он называется в BSD, буфер истории). Оказывается, если вывод данной консоли не умещается на один экран — его можно "пролистать" назад и, до текущего положения, вперед, как в текстовом редакторе. Правда, одних клавиш <PageUp>/<PageDown> для этого недостаточно. В консоли Linux (и wscons) пролистывание осуществляется одной из этих клавиш при нажатом <Shift>. А в syscons перевод в режим "листания" буфера истории осуществляется нажатием фиксируемого переключателя <ScrollLock>.

Есть и еще одно отличие буфера истории консолей BSD и Linux. В первой он полностью независим для каждого виртуального терминала. И, переключаясь между ними, можно листать страницы их истории, сколько вздумается (в некоторых пределах, установленных в текущей конфигурации ядра).

В Linux же в каждый момент времени доступен только буфер истории текущего виртуального терминала. Переключение в другую консоль активизирует буфер истории, но необратимо стирает историю консоли предыдущей (как любитель истории, не могу не отметить, что больший "историзм" BSD-систем проявляется и в таких мелочах).

Третий поражающий воображение фактор — возможность настроить для каждой консоли свою цветовую гамму: установить свой цвет фона и шрифта, а в syscons еще и цвет так называемого бордюра — это та самая черная каемка, памятная заставшим времена старых 14-дюймовиков без средств цифрового управления, на которых искоренить ее нельзя было никакими силами. Так вот и этот абсолютно, казалось бы, паразитный элемент можно приспособить для использования в мирных целях — для визуального разли-

чения консолей (например, консоль для регистрации root у меня всегда имеет бордюр тревожно-красного цвета).

И, наконец, четвертый: хотя виртуальные терминалы изображают собой (почти) самостоятельные машины, между ними возможен обмен данными. И служит этому любимое устройство "подоконников" — самая обычная мышь. Каковая в консоли служит не средством указательства или позиционирования курсора (в UNIX-консоли текстовый курсор и курсор мыши суть вещи, совершенно друг с другом не связанные, во FreeBSD это подчеркивается даже их разным представлением на экране), а предназначена для помещения выделенных экранных фрагментов в специальный буфер обмена. Из него они могут быть скопированы в любой активный виртуальный терминал.

Выделение мышью фрагмента осуществляется двояко: или обычным для Windows способом — перемещением мышиного курсора при нажатой левой кнопке, или двумя последовательными щелчками — сначала левой кнопкой на начале выделяемого фрагмента, затем правой на его конце.

А копирование из буфера происходит и того проще, переводом текстового (не мышиного!) курсора в нужную позицию того же виртуального терминала или переключением на другой виртуальный терминал и щелчком средней кнопкой. В наиболее распространенных нынче двухкнопочных мышах с колесом прокрутки последнее успешно выполняет роль средней кнопки. И, наконец, в совсем уж ущербных двухкнопочных мышах действие средней кнопки в большинстве случаев можно эмулировать одновременным нажатием двух наличных...

О несравненных достоинствах консоли можно говорить бесконечно. Но пора переходить к разговору о режимах.

О режимах

Когда машины были большие, и когда к ним начали прикручивать дисплеи в качестве устройств вывода, дисплеи эти были разными: текстовыми или графическими.

Текстовые дисплеи в терминальных комплектах использовались чаще. Их особенность в том, что они в принципе были способны выводить символы только из некоего предопределенного набора, жестко прошитого в ПЗУ знакогенератора. Конечно, в такие наборы входили не только текстовые (то есть алфавитно-числовые) символы, но и так называемая псевдографика (уголки, линейки и т. д.), с помощью которой можно было строить изображения (например, элементы меню). Но важно то, что выход за пределы предопределенного набора символов был невозможен (я помню времена, когда

для обеспечения вывода символов кириллицы требовалось перепрограммирование знакогенератора, что и проделывалось нашими умельцами).

Графические дисплеи работали по другому принципу: в них изображение строилось с помощью специальных алгоритмов, попиксельно. Что, соответственно, давало в принципе возможность вывода на экран совершенно произвольных вещей: любых наборов символов в различном шрифтовом оформлении, а также собственно изображений — векторных (то есть образованных линиями, описываемыми математическими уравнениями) или растровых (слагавшихся из наборов пикселов определенного цвета). Тем не менее, графические дисплеи по природе своей были устройствами растровыми, даже векторные кривые в конечном счете слагались из цепочек пикселов.

С появлением персоналок собственно текстовые дисплеи (MDA и ранние Hercules) быстро вышли из употребления. Видеосистемы IBM-совместимых машин, объединяющие в себе видеоадаптер (или, как часто говорят, видеокарту) и собственно монитор, стали графическими. Однако текстовый режим как таковой в них сохранился. Только теперь предопределенные наборы символов не прошивались в "железе", а загружались в видеокарту. Именно программирование видеокарт и обусловило возможность создания простых способов работы с нелатинскими наборами символов (в том числе и кириллицей).

Постепенно текстовый режим вытеснялся различными графическими режимами (а были системы, изначально ориентированные на использование графики, такие как Mac OS или Amiga). И лишь в UNIX-подобных ОС текстовый режим до недавнего времени прочно удерживал свои позиции, и в значительной мере именно благодаря поддержке виртуальных терминалов.

Часто неявным образом ставится знак равенства между текстовым режимом и режимом консольным (или режимом терминального доступа). В общем случае это неверно. Конечно, виртуальные консоли часто функционируют именно в текстовом режиме (в большинстве BSD-систем — почти исключительно в нем). Однако существует и понятие так называемой графической консоли, которое не следует смешивать с понятиями графических рабочих сред. Ибо это самая обычная консоль, но изображения на ней (в том числе и текстовые символы) выводятся графическими средствами, то есть прорисовкой, а не вытягиванием из предопределенного набора. Это возможно потому, что все современные (именуемые обычно VESA-совместимыми) видеокарты поддерживают линейный кадровый буфер (Frame Buffer), допускающий прямое воспроизведение графики без использования специальных программных средств.

В принципе, с распространением жидкокристаллических мониторов можно ожидать полного отмирания чисто текстовой консоли, не больно эстетично выглядит стандартный текстовый режим (80 колонок на 25 строк) на LCD-

матрицах с физическим разрешением 1280×1024; не говоря уже о неэффективном использовании экранного пространства. Однако не думаю, что важность консоли от этого уменьшится.

Таким образом, графическая консоль (или фрейм-консоль) знаменует собой плавный переход от текстового режима к режиму графическому. Последний в POSIX-системах может быть реализован различными способами.

Первый способ — только что упомянутая графическая консоль Linux, в которой возможно изменение экранного разрешения в широких пределах, вплоть до 1280×1024, глубины цвета, просмотр изображений и даже видео.

Ядра NetBSD и OpenBSD, насколько мне известно, режима Frame Buffer не поддерживают. А во FreeBSD такая возможность теоретически могла быть включена, но только для фиксированного разрешения (800×600), да и на всех видеокартах, с которыми мне довелось иметь дело, выглядит это убого.

Однако ныне в DragonFlyBSD режим графической консоли реализован ничуть не хуже, чем в Linux-консоли, позволяя выставлять произвольные (из числа поддерживаемых видеоадаптером) разрешения. И уже поддержка графической консоли обратно портирована также в ядро FreeBSD, начиная с шестой ветки (в пятой она достигается специальными патчами).

Однако не только ограничения ОС не позволяют считать фрейм-консоль полноценной графической системой. Главное — крайне малое количество собственно графических приложений, способных использовать возможности Frame Buffer. В их числе, фактически, просмотрщик графических файлов, средство для создания скриншотов, изначально текстовый браузер links — и все. Да, еще Mplayer, собранный должным образом, может прокручивать видео во фрейм-консоли. Впрочем, эта великая программа способна проигрывать видео и в текстовой консоли — передавая его ASCII-символами (весьма занятное зрелище, доложу я вам).

По всем указанным причинам графическая консоль не может рассматриваться в качестве самостоятельного режима: назначение ее — в расширении возможностей консоли текстовой. И потому далее, за исключением особо оговоренных случаев, различий между ними делаться не будет.

Второй способ реализации графического режима в свободных POSIX-совместимых ОС — с помощью специализированной библиотеки SVGAlib. Сама по себе она разработана для ОС Linux, однако скомпилированные с ее использованием бинарные приложения можно запустить на любой BSD-платформе в так называемом режиме Linux-совместимости (своего рода эмуляции, основанной на подмене системных вызовов).

Хотя запускать-то особенно и нечего: на SVGAlib основано едва ли с полдюжины приложений (в числе которых, правда, знаменитый Doom). А поскольку проект этот практически прекратил свое развитие, ожидать сущест-

венного роста программ не приходится. Так что роль SVGAlib оказывается еще более ограниченной, чем графической консоли.

Таким образом, практически единственным универсальным способом реализации графического режима в POSIX-совместимых ОС оказывается оконная система X (X Window System), или, в просторечии, просто Иксы. Ибо X — это ее имя собственное (возникшее потому, что исторически ей предшествовала другая графическая система, именовавшаяся W), а Window в ее названии — прилагательное, призванное подчеркнуть ее оконную природу, в противоположность полноэкранным графическим системам. Ведь в прежние времена таких существовало немало (примером чему та же SVGAlib).

Начинающими пользователями Linux Иксы воспринимаются как неотъемлемая часть этой операционной системы. Однако это не так: оконная система X создавалась совершенно независимо не только от Linux, но и от UNIX вообще. И назначением ее было — обеспечивать графический режим работы на любых машинах, как-то способных вообще поддерживать графику, и поверх любых ОС.

Свободные реализации оконной системы X (а есть еще и сугубо коммерческие ее варианты, используемые в проприетарных UNIX) — "правильная" Xorg и "неправильная" Xfree86 — абсолютно идентичны в Linux, Free-, Net- и прочих BSD. И все приложения, написанные в расчете на запуск в абстрактных Иксах, будут с равным успехом работать в любой из этих систем.

Главной составной частью Иксов вообще (и XFree86 или Xorg в частности) является программа, именуемая X-сервером (собственно, устройством X-сервера и его функциональностью и различаются разные варианты реализации оконной системы X). Она запускается непосредственно на данной машине и взаимодействует с ее устройствами ввода (мышью и клавиатурой) и вывода — видеоподсистемой. А уже поверх X-сервера могут быть запущены разнообразные клиентские приложения. Причем, вопреки обычному пониманию терминов "клиент" и "сервер", именно они могут иметь своим местопребыванием не локальную машину, а любую другую, доступную по сети (локальной или глобальной).

Один из важнейших X-клиентов — программа терминального доступа, позволяющая эмулировать обычную текстовую консоль для ввода команд (и запуска любых программ вообще), почему ее и именуют еще эмулятором терминала. Таких эмуляторов существует великое множество, но минимум один (xterm) всегда будет в распоряжении пользователя, так как входит в штатный комплект любой реализации Иксов.

Не менее важен и другой класс X-клиентов — программы управления элементами графического интерфейса, так называемые оконные менеджеры (Window Managers). Их тоже преизрядное количество, но в состав Иксов стандартно входит лишь один — весьма архаичный twm.

Кое-что об Иксах уже говорилось во вводной части, подробностям же ее устройства будет посвящена *глава 16*.

Об интерфейсах

Итак, пользователь получил доступ к ресурсам машины через терминал в текстовом или графическом режиме. А что дальше? А дальше ему требуется некая программа для взаимодействия между ним, любимым, и операционной системой с протекающими в ней процессами и входящими в нее файлами.

За всю историю человечества было придумано лишь два принципиально различных класса интерфейсов (между прочим, это относится не только к компьютерной сфере деятельности).

Исторически первым (и, замечу, наиболее естественным) способом взаимодействия пользователя с машиной был командный интерфейс, основанный на отдаче прямых директив. Обычно он осуществляется с помощью клавиатуры, однако теоретически можно использовать и любой другой способ ввода команд: скажем, щелчком средней кнопки мыши. Именно ему под именем CLI (Command Line Interface, интерфейс командной строки) суждено было на долгие годы стать традиционным интерфейсом UNIX (а затем и POSIX-совместимых систем вообще).

За вторым способом взаимодействия человека и машины прочно закрепилось название графического (GUI, Graphic User Interface, графический интерфейс пользователя). Название, впрочем, не строгое, а в общем случае и просто неверное: далее будут даны примеры такого рода интерфейсов, функционирующих в текстовом режиме, и напротив — командных интерфейсов, реализуемых в режиме графическом.

Основной принцип интерфейсов второго рода — отказ от использования прямых командных директив в пользу манипулирования объектами (файлами, каталогами, интерфейсными элементами и т. д.), выполняемого обычно с использованием мыши. Хотя, как мы видели ранее, с помощью мыши можно и вводить прямые команды. А манипулировать объектами с помощью клавиатурных комбинаций также никто не запрещает.

"Совершенно верное определение" подобрать затрудняюсь, но то, что именуется графическим интерфейсом, скорее, следовало бы называть интерфейсом объектным, манипуляционным или как-то в этом роде. Хотя все эти термины неудачны: первый из-за ассоциации с ООП в понимании Грэди Буча (к коему в общем случае иметь отношения вовсе не обязан), второй — просто из-за трудновыговариемости.

Не вполне адекватен также термин "сенсуальные" интерфейсы, предложенный в свое время Максимом Отставновым. На основании того, что звук стал их полноправной частью. До недавнего времени это было не более чем мечтой: почти никакой функциональной нагрузки (кроме более или менее мелодичных стонов и визгов) звук в компьютерной рабочей среде не нес. И дожить до полноценных голосовых (особенно русскоязычных) интерфейсов я не особенно рассчитывал. Хотя недавно в очередной раз был посрамлен в своем пророчестве: в KDE версии 3.4 управление голосом приобрело уже практический смысл. Однако использовать голосовые технологии для отдачи прямых командных директив будет ничуть не сложнее, чем для манипулирования объектами (на мой взгляд, даже легче).

В итоге я остановился на определении "визуальные интерфейсы", так как зрительный контроль действий в них абсолютно необходим. В отличие от сред командных, где теоретически вполне можно представить стучащего по клавиатуре вслепую оператора, обращающего взгляд на экран только для созерцания результата. Или не обращающего вовсе, если результат его не очень интересует.

Для примера: вы пытались когда-либо объяснять по телефону, как выполнить с помощью компьютера некое действие? Если да, знаете, что при использовании DOS это вполне проходило, в Windows же — получалось скверно...

Так что впредь я применительно к интерфейсам, основанным на манипулировании объектами, призываю употреблять термин "визуальные" (за неимением пока лучшего). Исключительно как антитезу интерфейсам командным, предполагающим управление прямыми директивами. Собственно же слово "графический" следует оставить за определением режимом, альтернативного режиму текстовому, как было сказано в предыдущем разделе.

Разумеется, между используемым режимом и типом интерфейса на практике есть некоторая, хотя и не однозначная, корреляция. Так, при консольном доступе (вне зависимости — в чисто текстовом же режиме или во фрейм-консоли) пользователь фактически обречен на использование той или иной разновидности CLI. Для чего ему служат программы класса командных оболочек (shells): одна из таких программ (login shell) запускается при авторизации в системе и берет на себя ответственность за интерпретацию и исполнение вводимых пользователем директив.

Говоря об обреченности, я не имею в виду фатального принуждения. Ибо пользователю в качестве login shell вольно использовать чуть ли не любую программу, в том числе и обеспечивающую ему нечто вроде визуального интерфейса. И программы такие имеются, знаменитый Midnight Commander тому примером. Поскольку, несмотря на свою сугубо текстовую ориентацию, в основе его лежит именно манипулирование объектами, осуществ-

ляемое комбинациями "горячих клавиш". Другое дело, что очень быстро для пользователя становится ясным, что mc не делает ничего такого, чего нельзя было бы выполнить (и обычно гораздо быстрее и проще) прямыми командами оболочки.

Тут-то и приходит осознание обреченности, в хорошем смысле этого слова, обреченности на более эффективные приемы работы. Вопреки мнению классиков советской фантастики, я не считаю ущемлением свободы обреченность на что-то хорошее — например, на любовь хорошей девушки. И к тому же, свободы выбора (обречь себя на любовь девушки нехорошей) отнять у нас никто не в силах. Или, паче того, на любовь Midnight Commander, впрочем, это отдает уже нетрадиционной ориентацией.

За своеобычное лирико-дидактическое отступление не извиняюсь. Потому что в графическом режиме корреляция с типами интерфейсов еще сложнее. Начать с того, что сама по себе оконная система X (а мы выяснили, что это единственный универсальный способ обеспечения графического режима) никакого пользовательского интерфейса вообще не обеспечивает, а служит лишь средой для запуска клиентских программ.

Среди клиентских Иксовых программ выделяются два важнейших их типа, которые и способны обеспечить пользовательский интерфейс внутри оконной системы. Это эмуляторы терминала и оконные менеджеры.

Так вот, пользователь, запустив совместно с Иксами эмулятор терминала, оказывается в самой что ни на есть командной среде — том же login shell, что и в консоли. Где он точно так же, как и в консоли, может вводить командные директивы, в том числе и на запуск истинно графических приложений. С той только разницей, что функционирует его командная оболочка в Иксовом окне, а не в полноэкранном (хотя и виртуальном) терминале.

Правда, воспользоваться прелестями оконного интерфейса он пока не в силах, ведь средств управления окнами (открытия, закрытия, масштабирования, минимизации и даже переключения между окнами) у него нет. Чтобы получить такие средства, ему требуется еще одна клиентская программа — менеджер окон. И вот это уже полноценный визуальный пользовательский интерфейс, он же GUI в традиционной терминологии.

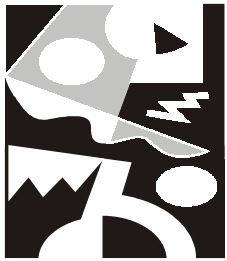
Существуют многие множества оконных менеджеров, представление о их количестве можно получить на сайте Window Managers for X (**http://xwinman.org/**). Они различаются своей функциональностью: одни обеспечивают только базовые средства управления окнами (масштабирования, перемещения, переключения) и соответствующие интерфейсные элементы (кнопки минимизации, развертывания и т. д.). Другие же имеют развитые средства запуска программ, управления запущенными приложениями.

Особый вид X-клиентов представлен интегрированными графическими средами, или, как их еще называют, десктопами. В их числе KDE, GNOME, XFce. От собственно оконных менеджеров они отличаются тем, что, помимо собственно средств управления интерфейсом, включают в себя еще некоторое количество пользовательских приложений: программы эмуляции терминала, файловые менеджеры, браузеры, почтовые клиенты и прочие коммуникационные программы, текстовые редакторы и даже офисные пакеты.

Традиционно интерфейсы командного стиля противопоставляются визуальным интерфейсам. Основанием служат абсолютно разные механически (даже, я сказал бы, физиологически) приемы работы в них. Однако обращение с автомобилем и, скажем, лошадью также требует совершенно разных приемов и навыков. Что не мешает в обоих случаях достигать одной и той же цели (транспортировки чего-либо или кого-либо). С различным, правда, успехом в разных условиях.

Аналогично и с интерфейсами. Множество задач целесообразно решать с помощью командных директив — примером чему пакетная обработка текстовых данных. В то же время, скажем, обработку изображений обычно легче выполнить в визуальной среде (хотя и здесь роль команд не следует недооценивать). И потому наиболее эффективным оказывается сочетание командных и визуальных методов. Благо, POSIX-системы предоставляют богатые возможности комбинирования тех и других.

Глава 12



Истина — в командах

- Введение в CLI.
- Командная строка.
- Опции.
- Аргументы.
- Кое-что об исключениях.
- Псевдонимы.
- Переменные.
- Навигация и редактирование.
- История команд.
- Регулярные выражения.
- Командные конструкции.
- Сценарии оболочки: первые представления.
- Понятие о функциях.
- Самая главная команда.

Нам уже приходилось говорить о командах, а некоторые даже использовать на практике. Потому что командный интерфейс (интерфейс командной строки, Command Line Interface, CLI) — это очередная вечная истина POSIX-мира, постижение которой позволит пользователю взаимодействовать с системой.

Введение в CLI

Как говорилось в предыдущей главе, POSIX-мир не обделен визуальными интерфейсами. Однако роль CLI от этого меньше не становится. Ибо CLI

универсален и представляет собой базу, для которой GUI всякого рода являют лишь оболочку.

Поясню последнюю мысль. Всякое действие в POSIX-системе может быть выполнено прямой командной директивой (это станет понятно чуть позже). И его же можно осуществить путем манипулирования объектами. Например, копирование файлов выполняется соответствующей командой `cp`, это первый способ. Но его же можно осуществить перетаскиванием мышью объекта, представляющего наш файл зрительно, из того места, где он находился ранее, туда, где мы хотим видеть его копию, а это уже второй способ. И так почти во всем. Так вот, манипуляция объектами в GUI — это обычно более или менее опосредованное выполнение соответствующих данному действию команд. Поэтому основные навыки работы с CLI не помешают даже тому пользователю, который не вылезает из графической среды.

А теперь пора перейти собственно к командам. Интерфейс пользователя с POSIX-системой обеспечивается в большинстве случаев классом программ, именуемых командными интерпретаторами, командными процессорами, командными оболочками, или `shell` (шеллами, этот термин будет предпочтительным в дальнейшем изложении).

Как легко догадаться по одному из определений, кроме предоставления пользовательского интерфейса, шеллы выполняют и еще одну функцию: служат интерпретаторами собственных языков программирования. Сам по себе этот момент для нас сейчас не существенен. Однако именно на нем основывается классификация шеллов. Ибо программ таких существует изрядное множество, которое можно разделить на две группы: Bourne-shell-совместимые и C-shell-совместимые. В силу ряда причин в качестве стандарта POSIX принята одна из оболочек первой группы — так называемый POSIX-шелл. Правда, он представляет собой чистую абстракцию, однако большинство используемых в UNIX оболочек декларируют ту или иную степень совместимости с ним.

Собственно описанию шеллов будет посвящена *глава 15*. В главе же настоящей я попробую совместить роскошь (понимание принципов командного интерфейса) с необходимым — изучением базовых команд. А заодно и с полезным, то есть описанием особенностей того самого мифического POSIX-шелла, наиболее последовательно воплощенных в стандартных оболочках Free- и NetBSD (/bin/sh и /bin/ash, соответственно, это практически одно и то же). И потому все примеры, иллюстрирующие принципиальные вопросы, будут базироваться на наиболее используемых командах (мы ведь их понемножку продолжаем запоминать, правда?), построенных в соответствии с правилами POSIX-шелла. Однако при описании интерактивных возможностей командной строки рамки POSIX-шелла окажутся тесными, и тут придется обращаться к более "продвинутым" представителям этого семейства, например, bash и zsh, а также к C-shell.

Командная строка

Основой командного интерфейса является командная строка, начинающаяся с приглашения для ввода (в настоящей книге обозначается милым сердцу символом длинного зеленого друга `$`, хотя вид приглашения может быть настроен в широких пределах). Это среда, в которой задаются основные элементы командного интерфейса — командные директивы с их аргументами и опциями.

Командная директива (или просто команда) — основная единица, посредством которой пользователь взаимодействует с шеллом. Она образуется по определенным правилам, именуемым синтаксисом. Синтаксис командной директивы определяется, в первую очередь, языком, принятым в данной командной оболочке. Кроме того, некоторые команды (не очень многочисленные, но весьма популярные) имеют собственный, нестандартный синтаксис.

Однако в целом базовые правила построения команд имеют много общего, по крайней мере, в POSIX-совместимом семействе. И именно эти базовые правила будут предметом данного раздела. Синтаксические особенности отдельных нестандартных команд будут оговариваться по ходу изложения.

Итак, командная директива образуется:

- ❒ именем команды, однозначно определяющим ее назначение;
- ❒ опциями, определяющими условия выполнения команды;
- ❒ аргументами — объектами, над которым осуществляются действия.

Очевидно, что имя команды является обязательным компонентом, тогда как опции и аргументы могут и отсутствовать (или подразумеваться в неявном виде по умолчанию).

Еще один непременный компонент командной директивы — это специальный невидимый символ конца строки, именно его ввод отправляет команду на исполнение. В обыденной жизни этот символ вводится нажатием и отпусканием клавиши <Enter>. Почему обычно и говорят: для исполнения команды нажмите клавишу <Enter>. Тот же эффект, как правило, достигается комбинацией клавиш <Ctrl>+<M>. Конца командной строки, знаменующего исполнение команды, мы на экране не видим. Однако важно, что это такой же символ, как и любой другой (хотя и имеющий специальное значение). Это знание нам понадобится, когда речь дойдет до специальных символов вообще (и даже немного раньше).

В подавляющем большинстве случаев опции (или их последовательности) задаются непосредственно за именем команды, а аргумент (или группа аргументов) команду завершает, хотя это правило имеет некоторые исключе-

ния. Вне зависимости от порядка опций и аргументов, принятых для данной команды, интерпретация их осуществляется слева направо.

Команды, опции и аргументы обязательно разделяются между собой пробелами. Кроме того, опции обычно предваряются (без пробела) символом дефиса или двойного дефиса. Впрочем, немногочисленные команды могут использоваться с опциями без всяких предваряющих символов.

Как уже говорилось, имя команды определяет выполняемые функции. Существуют команды, встроенные в оболочку, то есть не имеющие запускающих их исполняемых файлов, и команды внешние. В последнем случае имя команды однозначно указывает на имя исполняемого файла программы, выполняемой при отдаче соответствующей директивы. Часто встроенные и внешние команды одного назначения имеют одинаковые имена, в этом случае обычно предпочтительно использование встроенных команд (впрочем, они и вызываются в первую очередь).

Некоторые команды могут выступать под несколькими именами. Это связано с тем, что исторически в различных UNIX-системах команды, исполнявшие одинаковые функции, могли получать разные названия. В каждой конкретной системе обычно используется только одна из таких команд-дублеров. Но при этом имена дублирующих команд также могут присутствовать в системе — для совместимости. Не следует думать, что это две различные программы одного назначения: как правило, такая синонимичность команд реализуется посредством механизма ссылок, жестких или символических, о чем говорилось в *главе 8*, или же с помощью псевдонимов (alias), о которых речь пойдет позднее.

Иногда команда, вызванная через имя своего синонима, может отличаться по своей функциональности от самой же себя, вызванной под родным именем. В этом случае говорят об эмуляции одной команды другой. Типичный пример — командная оболочка /bin/bash в большинстве дистрибутивов Linux имеет своего дублера — /bin/sh; вызванная таким образом, она воспроизводит функциональность того самого пресловутого POSIX-шелла, о котором я недавно говорил как о явлении мифическом.

Для правильного применения команд, конечно же, нужно знать их имена и назначение. Однако нас никто не заставляет напрягать пальцы вводом имени команды полностью. Потому что нам на помощь приходит великий принцип автодополнения: для любой команды достаточно ввести первые несколько ее символов — и нажать клавишу табуляции (<Tab>). И, если введенных буковок достаточно для однозначной идентификации, полное имя команды волшебным образом возникнет в строке. Если же наш ввод допускает альтернативы продолжения имени — все они высветятся на экране (сразу или после повторного нажатия на табулятор), и из них можно будет выбрать подходящую.

Поясню на примере. Создание пустого файла выполняется командой `touch`. Чтобы ввести ее имя в строку, достаточно набрать первые три ее буквы (`tou`) и нажать клавишу <Tab>, остальные два символа будут добавлены автоматически. Если же мы, из естественной человеческой лени, ограничимся только двумя первыми символами, то после нажатия табулятора нам будет предложен список возможных дополнений:

```
$ to <Tab>
toc2cue toc2mp3 toe     top     touch
```

В данном случае достаточно набора еще одной буковки (`u`) и повторного нажатия на <Tab>. А вот если прибегнуть к табулятору в пустой командной строке, перед нами предстанет список имен всех команд, доступных в данной системе. Правда, перед этим обычно задается вопрос, а хотим ли мы созерцать эти 500—700, а то и больше, имен.

> ***Отступление***
>
> Автодополнение — стандартная возможность bash и всех других командных оболочек, относимых к категории развитых. Но как раз в стандарте POSIX эта возможность не предусмотрена, и потому POSIX-шелл ее лишен. А в современных представителях семейства C-shell (tcsh) автодополнение реализуется несколько иначе.

Опции

Как уже говорилось, указания имени достаточно для выполнения некоторых команд. Типичный пример — команда `ls` (от list), предназначенная для просмотра имен файлов (строго говоря, содержимого каталогов). Данная без аргументов, она выводит список имен файлов, составляющих текущий каталог, представленный в некоторой форме по умолчанию. Например, в домашнем каталоге пользователя это будет выглядеть примерно так:

```
$ ls
back/    Desktop/  Mail/   OpenOffice.org1.1.2/  soft/        work/
bin/     docs/     mnt/    other/                tmp/
daemon/  lena/     music/  signature             wallpapers/
```

Исполнение же многих других команд невозможно без указания опций и (или) аргументов. Для них в ответ на ввод одного имени часто следует не сообщение об ошибке (или не только оно), но и краткая справка по использованию команды. Например, в ответ на ввод команды для создания каталогов `mkdir` (от make directory) последует следующий вывод:

```
usage: mkdir [-pv] [-m mode] directory ...
```

Для одних опций достаточно факта их присутствия в командной директиве, другие же требуют указания их значений (даваемых после опции через знак равенства). В приведенном примере команды `mkdir` к первым относятся опции `-v` (или `--verbose`), предписывающая выводить информацию о ходе выполнения команды (запомним эту опцию, в том же смысле она используется чуть ли не во всех командах UNIX), и `-p`, которая позволяет создать любую цепочку промежуточных каталогов между текущим и новым (в случае их отсутствия).

А вот опция `-m`, определяющая атрибуты доступа к создаваемому каталогу, обязательно требует указания значения этих самых атрибутов, заданных в символьной форме.

Многие опции имеют две формы: краткую, односимвольную, и полную, или многосимвольную. Некоторые же опции могут быть даны только в многосимвольной форме. Общее правило здесь таково: если одного символа достаточно для однозначного определения опции, могут употребляться обе формы в качестве равноправных. Однако поскольку количество символов латинского алфавита ограничено (а человеческая фантазия, конструирующая опции, безгранична), при большом количестве опций одной команды некоторые из них приходится делать исключительно многосимвольными.

Продемонстрирую это на примере опций все той же команды `mkdir`. Полный их список будет следующим:

```
-m, --mode=MODE установить код доступа
       (как в chmod)
-p, --parents не выдавать ошибок,
       если существует, создавать
       родительские каталоги,
       если необходимо
-v, --verbose печатать сообщение
       о каждом созданном каталоге
--help показать помощь и выйти
--version  вывести информацию
       о версии и выйти
```

Очевидно, что для опции `--version` краткая форма совпала бы с краткой формой опции `--verbose`, и потому первая существует только в полной форме. А вот для опции `--help` краткая форма в большинстве команд возможна и выглядит как `-h`. Более того, во многих командах вызов помощи может быть вызван посредством опции `-?`. К слову сказать, приведенный список опций команды `mkdir` получен именно таким способом.

Раз уж зашла речь об опциях `--version` и `-h` (`--help`, `-?`), давайте и их запомним на будущее. Это так называемые стандартные опции GNU, в число их входит и опция `-v`, `--verbose`. Назначение полной формы (`--version`, `--help`, `--verbose`) одинаковое во всех командах, краткой — во многих.

Опять-таки, из того же примера видно, что опции в односимвольной форме предваряются единичным символом дефиса и могут быть даны единым блоком, без пробелов:

```
$ mkdir -vpm 777 dir/subdir
```

При этом, естественно, опция, требующая указания значений, ставится последней, и ее значение отделяется пробелом. Опции же в многосимвольной форме требуют предварения удвоенным дефисом, обязательно должны разделяться пробелами, и значения их, если таковые требуются, присваиваются через символ равенства (он называется еще оператором присваивания):

```
$ mkdir --parents --mode=777 dir/subdir
```

Загадочные семерки после опции `-m` (`--mode`) — это и есть те самые атрибуты доступа, данные в символьной нотации, о которых речь шла в *главе 8*.

Опции команды именуются также флагами (реже ключами) или параметрами. Однозначной трактовки этих терминов нет. Однако обычно под флагами подразумеваются опции, не требующие указания значений. Параметр же применяется к опции, такового требующей, и объединяет опцию и ее значение. Правда, мне встречалось определение параметра как совокупности опций и аргументов, но я буду придерживаться приведенных определений.

Опции определяют условия выполнения команды. Ранее был приведен пример команды `ls` без опций. Однако на самом деле отсутствием опций при ней определяется вид выводимого списка по умолчанию, как многоколоночного списка, состоящего из имен файлов без учета скрытых файлов (а таковыми являются файлы, имена которых начинаются с символа точки, почему они еще называются dot-файлами), без каких-либо их атрибутов и визуального различения файлов различных типов.

Различные же опции команды `ls` определяют состав и формат выводимого списка файлов. Так, в форме

```
$ ls -a
```

она обеспечивает вывод списка имен всех файлов текущего каталога, включая скрытые файлы вида

```
.*
```

(символ `*` здесь обозначает шаблон имени, соответствующий любому количеству любых символов, в том числе и нулевому, а также символы текущего и родительского каталогов). В форме

```
$ ls -l
```

дается вывод списка имен файлов в "длинном" формате (отсюда название опции `l`, от long) с указанием атрибутов доступа, принадлежности, времени модификации, размера и некоторых других характеристик:

```
drwxr-xr-x   5 alv  users  136 2004-08-04 10:58 back/
drwxr-xr-x   2 alv  users  232 2004-07-29 13:00 bin/
drwxr-xr-x  19 root root   720 2004-08-04 14:45 daemon/
drwx------   3 alv  users  136 2004-07-18 14:00 Desktop/
drwxr-xr-x  13 alv  users  328 2004-07-26 10:40 docs/
drwxr-xr-x   2 alv  users  176 2004-07-21 03:43 lena/
drwxr-xr-x  17 alv  users 1736 2004-08-04 17:51 Mail/
drwxr-xr-x   2 alv  users   48 2004-07-20 22:24 mnt/
drwxr-xr-x  18 alv  users  552 2004-08-05 09:03 music/
drwxr-xr-x   5 alv  users  416 2004-08-04 14:17 OpenOffice.org1.1.2/
drwxr-xr-x   5 alv  users  144 2004-07-22 01:22 other/
-rw-r--r--   1 alv  users   71 2004-07-22 12:20 signature
drwxr-xr-x  11 alv  users  272 2004-08-04 16:42 soft/
drwxr-xr-x   3 alv  users  360 2004-08-06 08:29 tmp/
drwxr-xr-x   2 alv  users  344 2004-07-20 22:22 wallpapers/
drwxr-xr-x  14 alv  users  376 2005-10-11 06:43 work/
```

Форма

```
$ ls -F
```

позволяет получить список файлов с символьным различением файлов различных типов. Например, имя каталога будет выглядеть как

```
dirname/,
```

имя исполнимого файла

```
filename*
```

(здесь звездочка — не шаблон имени, а символическое обозначение исполняемого файла), и т. д.:

```
back/    Desktop/  Mail/   OpenOffice.org1.1.2/  soft/        work/
bin/     docs/     mnt/    other/                tmp/
daemon/  lena/     music/  signature             wallpapers/
```

А форма

```
$ ls --color=auto
```

представит те же типы файлов в списке в различной цветовой гамме (впрочем, при некоторых условиях; и, добавлю, `auto` — лишь одно из воз-

можных значений опции `--color`). Правда, это относится только к Linux, во FreeBSD опция `--color` ни в одной оболочке не сработает.

Я столь подробно остановился на команде `ls` не только из-за многочисленности ее опций: это одна из самых популярных команд просмотра файловой системы. И должным образом настроенная (в том числе с помощью приведенных опций), она дает ничуть не менее информативную и зрительно выразительную картину, чем развитые файловые менеджеры типа Midnight Commander или даже konqueror, о котором будет рассказываться в *главе 17*.

Порядок опций, если их приводится более одной, для большинства команд не существен. Хотя, например, для команды `tar`, создающей файловые архивы, опция `-f`, значением которой является имя создаваемого или распаковываемого архива, традиционно указывается последней. И, к слову сказать, именно эта команда — одна из немногих, опции которой не обязаны предваряться символами дефиса. Так, директивы

```
$ tar cf filename.tar dir
```

и

```
$ tar -cf filename.tar dir
```

абсолютно равноценны: и та, и другая создаст единый архивный файл `filename.tar` из отдельных файлов каталога `dir`.

Особый смысл имеет символ удвоенного дефиса `--`, если после него не следует никакой опции: таким образом обозначается конец списка опций, и все последующие отделенные пробелом символы интерпретируются как аргументы (со временем я расскажу, когда это оказывается необходимым). Одинарный же дефис с последующим пробелом, напротив, подменяет аргументы команды, знание этого нам потребуется, когда речь дойдет до командных конвейеров.

Аргументы

Однако тема стандартного ввода (а равно и вывода) у нас еще на горизонте. А пока мы подобрались к понятию аргументов командной директивы. Аргументами определяется, как правило, объект (или объекты) действия команды. В большинстве случаев в качестве аргументов команд выступают имена файлов и (или) пути к ним. Ранее говорилось, что при отсутствии аргументов команда `ls` выводит список имен файлов текущего каталога. Это значит, что текущий каталог выступает как заданный неявным образом (по умолчанию) аргумент команды `ls`. Если же требуется вывести список имен файлов каталога, отличного от текущего, путь к нему должен быть указан в качестве аргумента команды явно, например:

```
$ ls /usr/local/bin
```

Для правильного построения аргументов команды требуется рассмотрение еще одного понятия — пути к файлу. Путь — это точное позиционирование файла в файловой системе относительно ее корня или нашего в ней положения — текущего каталога.

Так, если пользователь находится в своем домашнем каталоге (абсолютный путь к нему обычно выглядит как /home/username), то просмотреть содержимое каталога /usr/local/bin он может вот так:

```
$ ls ../../usr/local/bin
```

Первый путь в аргументе команды `ls` абсолютный, отсчитываемый от корневого каталога, второй задается относительно каталога текущего, ведь `../` — это родительский каталог к нему.

Пути в аргументах команд могут быть весьма длинными. Например, чтобы просмотреть доступные раскладки клавиатуры, в Linux нужно дать команду примерно следующего вида:

```
$ ls /usr/share/kbd/keymaps/i386/qwerty
```

И читатель вправе спросить: неужели мне все это вводить вручную? Отнюдь, — отвечу я ему. Потому что автодополнение действует также для путей в их аргументах. И в данном случае обязательный ввод будет выглядеть следующим образом:

```
$ ls /ushkki3qy
```

Все недостающие символы будут добавлены автоматически. А такая оболочка, как zsh, вообще позволяет (при соответствующих настройках, о чем будет говориться в *интермедии* после *главы 15*) обойтись следующей последовательностью:

```
$ ls /u/s/k/k/i/q
```

Большинство команд допускает указание не одного, а нескольких (и даже очень многих) аргументов. Так, единой директивой вида

```
$ cp file1 file2 ... fileN dir
```

можно скопировать (команда `cp` — от copy) сколько угодно файлов из текущего каталога в каталог `dir`.

Маленькое отступление

Упоминание команды `cp` — удобный случай чуть вернуться назад и рассмотреть одну очень важную опцию, почти универсальную для команд POSIX-систем. Для начала попробуем скопировать один каталог в другой:

```
$ cp dir1 dir2
```

Как вы думаете, что получится в результате? Правильно, сообщение о невозможности выполнения этой операции, примерно в таком виде:

```
cp: omitting directory 'dir1'
```

Команда `cp` в чистом виде для копирования каталогов не предназначена. Что делать? Очень просто — указать опцию `-R` (от Recursive; в большинстве систем проходит и опция `-r` с тем же смыслом, но первая форма работает абсолютно везде). В результате в каталог `dir2` не только будут скопированы сам каталог `dir1` и все входящие в него файлы, но и вложенные подкаталоги из `dir1`, если таковые имеются.

Вполне возможно, что в дистрибутиве, который имеется в вашем распоряжении, проходит и копирование каталогов просто через `cp`, без всяких дополнительных опций. Это потому, что команда `cp` часто определяется как псевдоним самой себя с опцией рекурсивного копирования, о чем скоро пойдет речь в этой главе.

Вообще рекурсия (определение некоего выражения через самого себя) — очень важное понятие в UNIX, послужившее даже базой для своеобразного хакерского юмора. Вспомним GNU, что, согласно авторскому определению, означает GNU is Not UNIX. Однако сейчас для нас важно только то, что рекурсия применима практически ко всем файловым операциям, позволяя распространить действие одной командной директивы не только на файлы данного каталога, но и на все вложенные подкаталоги и их содержимое.

Вернемся к аргументам. Действие некоторых команд неоднозначно в зависимости от аргументов, к которым она применяется. Например, команда `mv` служит как для переименования файлов, так и для их перемещения в другой каталог. Как же она узнает, что ей делать в данном конкретном случае? Да именно по аргументам. Если дать ее в форме

```
$ mv filename1 filename2
```

то следствием будет переименование `filename1` в `filename2`. А вот если первым аргументом указан файл, а вторым — каталог, например,

```
$ mv filename dir
```

то результатом будет перемещение `filename` из текущего каталога в каталог `dir`. К слову сказать, команды типа `mv` воспринимают разное количество аргументов в зависимости от того, какие они, эти аргументы. В первом примере аргументов может быть только два, имя исходного файла и имя файла целевого. Зато во втором примере в качестве аргументов можно задать сколько угодно файлов и каталогов (с учетом оговорки относительно "сколько угодно"), все они будут перемещены в тот каталог, который окажется последним в списке. То есть директивой:

```
$ mv file1 ... fileN dir1 ... dirM dirN
```

в каталог `dirN` будут перемещены файлы `file1` ... `fileN` и каталоги `dir1` ... `dirM`. Характерно, что для этого команде `mv`, в отличие от команды `cp`, не требуется каких-либо дополнительных опций — она рекурсивна по самой своей природе.

Кое-что об исключениях

Итак, типичная форма POSIX-команды в обобщенном виде выглядит следующим образом:

```
$ command -[options] [arguments]
```

Из этого правила выбиваются немногочисленные, но весьма полезные и часто используемые команды. Однако и для таких команд с нестандартным синтаксисом устанавливаются те же компоненты: имя, опции, аргументы, хотя по ряду причин (в том числе исторических) порядок их может меняться.

Это можно проиллюстрировать на примере полезнейшей команды `find`, предназначенной для поиска файлов (и не только для этого — она являет собой почти универсальное орудие в деле всякого рода файловых манипуляций, почему и сподобилась отдельного описания в интермедии о манипуляции файлами). В типичной своей форме она выглядит примерно следующим образом:

```
$ find dir -option1 value -option2 [value]
```

Здесь `dir` (каталог, в котором выполняется поиск) может рассматриваться в качестве аргумента команды. Опция `-option1` (обратим внимание, что здесь, несмотря на многосимвольность опций, они предваряются единичным символом дефиса) и ее значение `value` определяют критерий поиска (например, `-name filename` — поиск файла с указанным именем). Опция `-option2` предписывает, что же делать с найденным файлом (файлами), например, `-print` — вывести его имя на экран. Причем опция действия также может иметь значение. Так, значением опции `-exec` будет имя команды, вызываемой для обработки найденного файла (файлов). Директива вида

```
$ find ~/ -name *.tar -exec tar xf {} \;
```

требует отыскать в домашнем каталоге (`~/`), выступающем в качестве аргумента, файлы, имя которых (первая опция — критерия поиска) соответствует шаблону `*.tar` (значение первой опции), и выполнить (вторая опция — действия) в их отношении команду `tar` с собственными опциями, обеспечивающими распаковку архивов (значение второй опции). Интересно, что в этом контексте в качестве значений второй опции команды `find` выступает не только внешняя команда, но и все относящиеся к ней опции.

В последнем примере имеется несколько символов, смысл которых может показаться непонятным. Надеюсь, он прояснится достаточно скоро, в разделе о регулярных выражениях.

Псевдонимы

Вернемся на минуту к команде `ls`. У читателя может возникнуть вполне резонный вопрос: а если я всегда хочу видеть ее вывод в цвете, да еще с символическим различением типов файлов, да еще в "длинном" формате? Ну и без вывода скрытых файлов мне никак не прожить. И что же, мне каждый раз вводить кучу опций, чтобы получить элементарный эффект?

Этот вопрос задавали себе многие поколения не только пользователей, но и разработчиков. И ответили на него просто — введением понятия псевдонима команды (`alias`).

Что это такое? В большинстве случаев — просто некоторое условное имя, подменяющее определенную команду с теми ее опциями, которые мы используем чаще всего. Причем, что характерно, псевдоним команды может совпадать с ее именем. То есть, например, набирая просто `ls`, мы получаем список файлов не в формате по умолчанию, а в том, в каком угодно нам.

Устанавливаются псевдонимы очень просто — одноименной командой `alias`, в качестве аргументов которой выступают имя псевдонима и его значение, соединенные оператором присваивания (именуемым в просторечии знаком равенства). А именно: если мы хотим видеть вывод команды `ls` в цвете и с символьным различением типов файлов, в Linux нам достаточно дать команду вроде следующей:

```
$ alias ls='ls -F --color=auto'
```

В BSD-реализации команда `ls` не понимает опции `--color=auto`, и потому здесь аналогом приведенного псевдонима будет:

```
$ alias ls='ls -FG'
```

В обоих случаях следует обратить внимание на два момента:

- имя псевдонима совпадает с именем команды (что отнюдь не препятствует созданию псевдонима типа `ll='ls -l'` специально для вывода файловых списков в длинном формате);
- значение псевдонима заключено в одинарные кавычки.

Пока просто запомним, что кавычки (и именно одинарные) — обязательный атрибут команды установки псевдонима.

Маленькое отступление

Приведенная форма определения псевдонима действительна для всех shell-совместимых оболочек. В оболочках же семейства C-shell синтаксис команды будет иным:

```
$ alias ls ls -FG
```

То есть разделителем между именем псевдонима, командой и ее опциями выступают символы пробела (или табуляции). А заключение значения псевдонима в кавычки потребуется только в том случае, если командная конструкция включает более одного пробела (например, сопровождается несколькими опциями или аргументами).

Таким образом мы можем наделать себе псевдонимов на все случаи жизни. В разумных пределах, конечно, иначе вместо упрощения жизни мы создадим себе необходимость запоминания множества невнятных сочетаний символов. Однако на наиболее важных (и обычно определяемых) псевдонимах я остановлюсь.

Вспомним команды типа `cp` и `mv`, которыми мы, в частности, можем скопировать или переместить какие-то файлы из каталога в каталог. А что произойдет, если случайно в целевом каталоге уже имеются файлы, одноименные перемещаемым? Файлы в целевом каталоге будут заменены новыми, теми, что копируются туда или перемещаются. То есть исходное содержание этих файлов будет утрачено — и безвозвратно, восстановить его невозможно будет никакими силами.

Разумеется, иногда так и нужно, например, при резервном копировании старые версии файлов и должны быть заменены их более свежими вариантами. Однако такое приемлемо далеко не всегда. И потому в большинстве команд, связанных с необратимыми изменениями файловой системы, предусматривается специальная опция `-i` (или `--interactive`). Если задать эту опцию с командой `cp` или `mv`, то при совпадении имен исходного и целевого файлов будет запрошено подтверждение на выполнение соответствующего действия:

```
$ cp file1 file2
cp: overwrite  file2'?
```

И пользователь может решить, нужно ли ему затирать существующий файл, ответив `yes` (обычно достаточно `y`), или это нежелательно, и следует ответить `no` (а также просто `n`, или не отвечать ничего, это равноценно в данном случае отрицательному ответу).

Так вот, дабы не держать в голове необходимость опции `-i` (ведь, как я уже говорил, пропуск ее в неподходящий момент может привести к весьма печальным результатам), в подавляющем большинстве систем для команд `cp` и `mv` (а также для команды `rm`, служащей для удаления файлов, в POSIX-системах эта операция также практически необратима) определяются одноименные им псевдонимы такого вида:

```
$ alias cp='cp -i';
$ alias mv='mv -i';
$ alias rm='rm -i'
```

Все это, конечно, очень благородно, заметит внимательный читатель. Но что, если мне заведомо известно, что сотни, а то и тысячи файлов целевого каталога должны быть именно переписаны новыми своими версиями (как тут не вспомнить про случай с резервным копированием)? Что же, сидеть и жать на клавишу <Y>?

Не обязательно. Потому что все команды рассматриваемого класса имеют еще опцию `-f` (`--force`, она также практически универсальна для большинства команд). Отменяя действие опции `-i`, она предписывает принудительно переписать все файлы целевого каталога их обновленными тезками. И никто не мешает нам на этот случай создать еще один псевдоним для команды `cp`, например:

```
$ alias cpf='cp -f'
```

Правда, предварительно нужно убедиться, что в системе нет уже команды с именем, совпадающим с именем псевдонима, иначе эффект может быть весьма неожиданным (впрочем, это относится ко всем псевдонимам, не совпадающим с именами подменяемых команд).

Есть и другой способ обойти опции, установленные для команды-псевдонима: просто отменить псевдоним. Что делается командой обратного значения

```
$ unalias alias_name
```

То есть, дав директиву

```
$ unalias cp
```

мы вернем команде копирования ее первозданный смысл. Ну, а узнать, какие псевдонимы у нас определены в данный момент и каковы их значения, еще проще: команда

```
$ alias
```

без опций и аргументов выведет полный их список:

```
la='ls -A'
less='less -M'
li='ls -ial'
ll='ls -l'
ls='ls -F --color=auto'
```

В текущем сеансе пользователя заданные псевдонимы работают. Однако после рестарта системы (или просто после выхода из данного экземпляра командной оболочки) они исчезнут без следа. Чтобы увековечить, их нужно прописать в конфигурационном файле пользовательского шелла. Но этим мы займемся впоследствии — в *главе 15*. А пока обратимся к переменным.

Переменные

Переменные играют для аргументов команд примерно такую же роль, что и псевдонимы — для команд. То есть избавляют от необходимости мрачного ввода повторяющихся последовательностей символов. Конечно, это далеко не единственное (а может быть, и не главное) назначение переменных, однако на данном этапе для нас наиболее существенное.

Имена переменных в принципе могут быть любыми, хотя некоторые ограничения также существуют. Я уже вскользь упоминал о переменных в разделе про пользователей. В частности, там, помнится, фигурировали переменные `SHELL`, `USER`, `HOME`. Так вот, эти (и еще некоторые) имена зарезервированы за внутренними, или встроенными, переменными оболочки (некий минимальный их набор имеется в любом шелле). То есть значения их определены раз и навсегда и пользователем не изменяются. Таких встроенных переменных довольно много. И одна из первых по значению — переменная `PATH`. Это список каталогов, в которых оболочка, в ответ на ввод пользователя в командной строке, ищет исполняемые файлы, — просто команды. Вы, наверное, обратили внимание, что во всех приведенных примерах имена команд указывались без всяких путей к ним (в отличие от файлов-аргументов, путь к которым обязателен). Так вот, успех ее поисков и определяется списком значений переменной `PATH`. Каковые могут быть просмотрены командой:

```
$ echo $PATH
```

Обратим внимание на то, что в качества аргумента команды выступает не просто имя переменной, а оно же, но предваренное символом доллара. В данном случае никакого отношения к приглашению командной строки он не имеет, а предписывает команде `echo` подменить имя переменной ее значением (значениями). Вывод команды будет примерно таким:

```
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin
```

Тем временем вернемся к переменной `HOME`. Значение ее — полный абсолютный путь к домашнему каталогу пользователя. То есть, чтобы перейти в него, пользователю по имени `alv` вместо

```
$ cd /home/alv
```

достаточно набрать

```
$ cd $HOME
```

Он в своих владениях! Может показаться, что экономия грошовая (тем паче, что перейти в собственный каталог пользователь может просто командой `cd` без всяких аргументов), но минуту терпения — и выгоду от использования переменных вы увидите.

Кроме переменных, предопределенных в шелле, пользователю предоставляется почти полная свобода в определении переменных собственных. И вот тут-то и наступает ему обещанное облегчение при наборе аргументов команд.

Предположим, что у нас имеется глубоко вложенный подкаталог с данными, постоянно требующимися в работе. Чисто условно примем, что путь к нему следующий:

```
/home/alv/data/all.my.works/geology/plate-tectonics
```

Весьма удручающе для набора, даже если исправно работать табулятором для автодополнения, не так ли? Прекрасно, упрощаем себе жизнь определением переменной:

```
$ plate=/home/alv/data/all.my.works/geology/plate-tectonics
```

Дело в шляпе. Теперь, если нам нужно просмотреть состав этого каталога, достаточно будет команды

```
$ ls $plate
```

А вызвать из него любой файл для редактирования можно так:

```
$ joe $plate/filename
```

Подобно псевдонимам, переменные, определенные таким образом (то есть просто в командной строке), имеют силу только в текущем сеансе работы — при выходе из оболочки они утрачиваются. Для того чтобы они действовали перманентно, переменные должны быть прописаны в конфигурационном файле пользовательского шелла. Однако, в отличие от псевдонимов, и этого оказывается не всегда достаточно. Ибо переменная, определенная посредством

```
$ NAME=Value
```

работает не просто только в текущем сеансе, но еще и только в конкретном экземпляре шелла. Почему и называется переменной оболочки shell variable. В разделе о процессах мы уже видели, что практически любое действие в шелле (запуск команды или программы, например) начинается с того, что оболочка, в которой это действие совершается, запускает новый экземпляр самой себя — дочерний шелл, или, как иногда говорят, субшелл.

Так вот, этот самый субшелл не наследует переменные родительской оболочки. И в итоге запущенная из командной строки программа ничего не будет знать, например, о путях к исполняемым файлам. Что автоматически ведет к невозможности запуска из нее команд просто по имени, без указания точного пути.

Чтобы избежать такой неприятной ситуации, было придумано понятие переменных окружения, или переменных среды — environment variable. Это те переменные, которые наследуются от родительского шелла всеми дочерни-

ми программами. И чтобы сделать их таковыми, переменные следует экспортировать. Как? Командой `export`, которая может быть применена двояким образом. Можно сначала определить переменную:

```
$ NAME=Value
```

а затем применить к ней команду `export`:

```
$ export NAME
```

А можно сделать это в один прием:

```
$ export NAME=Value
```

Второй способ применяется, если нужно определить и экспортировать одну переменную. Если же за раз определяется несколько переменных:

```
$ NAME1=Value1;
$ NAME2=Value2;
...;
$ NAMEN=ValueN
```

то проще прибегнуть к первому способу, так как команда `export` может иметь сколько угодно аргументов:

```
$ export NAME1 NAME2 ... NAMEN
```

Такой способ задания переменных действует только в POSIX-совместимых шеллах. В оболочках семейства csh для определения переменных обоих видов существуют специальные команды. Так, переменная оболочки задается командой `set` с указанием имени переменной и ее значения, соединенных оператором присваивания:

```
set EDITOR=joe
```

А чтобы та же самая переменная имела силу для всего окружения (то есть была бы переменной среди), потребуется другая команда:

```
setenv EDITOR=joe
```

Традиционно имена переменных окружения задаются в верхнем регистре, переменных оболочки — в нижнем.

Каждая из этих команда, данная без аргументов, выведет список определенных переменных оболочки и окружения соответственно. А для вывода значений абсолютно всех переменных есть специальная встроенная команда — `printenv`.

Навигация и редактирование

Имя команды, ее опции и аргументы образуют командные "слова". В качестве разделителей выступают пробелы. Кроме того, как разделители "слов"

интерпретируется ряд специальных символов: прямой слэш (элемент пути к файлу), обратный слэш (\), служащий для экранирования специальных символов, операторы командных конструкций, о которых будет сказано далее.

В некоторых случаях имеет смысл различать большое "слово" и малое. Первое включает в себя разделитель (например, пробел), в качестве же второго интерпретируются символы, лежащие между разделителями. Понятно выразился? Надеюсь, станет яснее при рассмотрении примеров.

Подчеркнем, что командное "слово" прямо не соотносится ни с опциями, ни с аргументами команды. Введение этого понятия призвано облегчить навигацию в командной строке и ее редактирование.

Ибо одно из великих достижений командного интерфейса POSIX-систем — это возможность перемещения внутри командной строки и внесения необходимых изменений в имя команды, ее опции и аргументы. Делается это различными способами.

Самый привычный и очевидный способ — использование клавиш перемещения курсора: <Left> (стрелка влево), <Right> (стрелка вправо), <End> и <Home>, действующих (хотя и не всегда) в командной строке точно так же, как и в каком-нибудь word-процессоре для Windows. А если добавить сюда еще клавиши <Delete> и <Backspace>, позволяющие удалять символы в позиции курсора или перед ней, то, казалось бы, чего еще желать?

Оказывается — есть чего, и самый очевидный способ навигации и редактирования оказывается не самым эффективным. Для начала заметим, что в общем случае привычные клавиши перемещения курсора и редактирования в POSIX-системах не обязаны работать так же, как они делают это в DOS/Windows. Так, клавиши <Up> (стрелка вверх), <Down> (стрелка вниз), <PageUp>, <PageDown> зарезервированы для других целей. Это зависит от многих причин, в том числе и исторических. Ведь POSIX-системы по определению предназначены работать на любых практически машинах (в том числе и на тех, клавиатуры которых клавиш управления курсором просто не имели).

Однако это не главное, в большинстве Linux-дистрибутивов командная оболочка по умолчанию настраивается так, чтобы пользователь при желании мог использовать привычные ему клавиши. Однако тут-то и оказывается, что в дополнение к этому оболочка предоставляет ему много более эффективную систему навигации по командной строке. И это система управляющих последовательностей, так называемых keybindings. То есть сочетания специальных клавиш (именуемых управляющими) с обычными алфавитно-цифровыми.

Основные управляющиеся клавиши, которые используются в таких последовательностях, это клавиши <Ctrl> и <Meta>. Пардон, — возразит внима-

тельный читатель, — сколько я ни долблю по клавишам моей персоналки, но клавиши <Meta> не замечал. Возражение принято: на PC-клавиатурах функции клавиши <Meta> выполняют либо нажатие и отпускание клавиши <Esc>, либо нажатие и удерживание клавиши <Alt>.

Впрочем, к этой теме я еще вернусь. А пока несколько простых рецептов, практически универсальных для любых командных оболочек в терминалах любых типов.

Рецепт первый: большая часть управляющих последовательностей состоит из сочетания клавиши <Ctrl> и алфавитно-цифрового символа. Под сочетанием (или комбинацией, для чего я уже употреблял ранее символ плюса) понимается то, что, удерживая нажатой клавишу <Ctrl>, мы одновременно нажимаем и какую-нибудь литерную или цифровую.

Так, действие клавишной комбинации <Ctrl>+<F> (от Forward) эквивалентно нажатию клавиши <Right>, это перемещение на один символ вправо, действие комбинации <Ctrl>+<B> (от Back) — нажатию <Left> (перемещение на один символ влево). Комбинации <Ctrl>+<A> и <Ctrl>+<E> действуют аналогично <Home> и <End>, перемещая курсор в начало и конец командной строки соответственно, Ну, а с помощью комбинаций <Ctrl>+ +<D> и <Ctrl>+<H> можно удалить единичный символ в позиции курсора или перед ней (также, как и клавишами <Delete> и <Backspace> соответственно).

Предвижу резонный вопрос: а какие достоинства в комбинации клавиш <Ctrl> и еще какой-то по сравнению с элементарными <End> или <Left>? Конечно, одно достоинство очевидно: при массовом вводе команд (и любых иных наборов символов, от исходных текстов до романов) и использовании keybindings руки не отрываются от основной (алфавитно-цифровой) части клавиатуры. И в итоге, по приобретении некоторого минимального навыка, дело движется гораздо быстрее. Обосновать тестами не могу, но не верящим — предлагаю попробовать.

Главное же преимущество клавиатурных последовательностей перед стандартными навигационными клавишами — много более широкая их функциональность. Я не случайно начал этот параграф с упоминания командных "слов". Это наряду с единичными символами также навигационные (и редакционные) единицы командной строки. То есть управляющие последовательности позволяют при навигации и редактировании оперировать не только единичными символами, но и целыми словами. Например, комбинация <Meta>+<F> смещает курсор на одно "слово" вперед, та же <Meta> в сочетании с <B> — на одно слово назад, и т. д. Прошу обратить внимание: действие алфавитной клавиши в комбинации с <Meta> сходно по смыслу ее сочетанию с клавишей <Ctrl>, но как бы "усилено": последовательность <Meta>+<D> уничтожает не символ в позиции курсора, как это было бы для <D> в сочетании с <Ctrl>, а все командное "слово".

Рассматривать ключевые последовательности подробно здесь я не буду: детали их действия зависят от командной оболочки и ее настроек. Отмечу только два существенных обстоятельства:

- keybindings предоставляют пользователю полный комплекс приемов для любых действий в командной строке, вплоть до преобразования регистров уже введенных символов и "слов" (из нижнего в верхний и наоборот), "перетасовки" символов в команде или ее аргументах и т. д. И не только в командной строке, большинство популярных в POSIX-мире текстовых редакторов, от простых nano или joe до грандиозных vim и emacs, построены по тому же принципу. Так что навыки, полученные при работе с keybindings, например, в bash, весьма поспособствуют виртуозному освоению любого из этих инструментов;
- действие ключевых последовательностей, как правило, не зависит от типа терминала и физического устройства клавиатуры, а также от ее раскладки, при переключении на кириллицу они будут работать столь же справно, как и в латинице.

История команд

Возможности навигации и редактирования строки особенно ярко проявляются в сочетании с другой замечательной особенностью, предоставляемой командными оболочками, — доступом к истории команд. Раз введенная в строке команда не уходит в небытие после исполнения, а помещается в специальный буфер памяти (который, как и все в UNIX, именуется весьма незатейливо — буфер истории команд; однако смешивать его с экранным буфером консоли, о котором говорилось в предыдущей главе, не следует — это вещи совершенно разные). Оттуда команда (со всеми ее опциями и аргументами) может быть извлечена для повторного использования. Или для редактирования и исполнения в новой реинкарнации.

Буфер истории команд сохраняется в течение всего сеанса работы. Однако в большинстве случаев командные оболочки настраиваются так, что по выходе из сеанса буфер истории сохраняется в специальном файле в домашнем каталоге пользователя, и таким образом его содержимое оказывается доступным при следующем запуске шелла. Имя этого файла может быть различным в разных оболочках, но обычно включает компонент history (в bash это ~/.bash_history, в zsh — ~/.zhistory). Так что, можно сказать, что введенным нами командам суждена вечная жизнь.

Конечно, не совсем вечная. И размер буфера истории команд, и количество строк в файле истории — величины конечные. Так что, если установленный предел превышен, то старые команды вытесняются более новыми. Однако и величину буфера, и количество строк в файле истории можно установить

любыми (в разумных пределах — не знаю, существует ли принципиальное ограничение на них, за исключением объема памяти и дискового пространства). А если учесть, что и из буфера, и из памяти с помощью соответствующих настроек (со временем я расскажу, каких) можно исключить дубликаты и еще кое-какой мусор, то мое заявление о вечной жизни команд не выглядит столь уж преувеличенным.

Универсальное средство доступа к буферу истории команд — специальная команда `history` (в большинстве дистрибутивов Linux она по умолчанию имеет псевдоним `h`). Данная без опций, эта команда выводит полный список команд в их исторической последовательности или некоторое количество команд, определенных соответствующими настройками (о которых будет говориться позднее).

В качестве опции можно указать желаемое количество одновременно выведенных команд. Например, директива

```
$ history -2
```

выведет две последние команды из буфера истории вместе с их номерами:

```
1023  joe shell.html
1024  less ~/.zshrc
```

Любая из команд в буфере истории может быть повторно запущена на исполнение. Для этого достаточно набрать в командной строке символ `!` (восклицательный знак) и затем без пробела номер команды в списке буфера. Например,

```
$ !1023
```

для приведенного выше примера повторно откроет файл shell.html в текстовом редакторе joe.

Другой способ доступа к командам из буфера истории — комбинации клавиш <Ctrl>+<P> и <Ctrl>+<N>, служащие для последовательного его просмотра (как бы пролистывания) назад и вперед (разумеется, если есть куда).

Любая извлеченная (с помощью стрелок или управляющими последовательностями) из буфера истории в текущую строку команда может быть повторно запущена на исполнение нажатием клавиши <Enter> или дублирующей ее комбинацией <Ctrl>+<M>. Причем предварительно ее можно отредактировать: изменить опции или аргументы точно так же, как и только что введенную.

Во всех современных "развитых" шеллах предусмотрены средства поиска команды в буфере истории простым перебором (обычно <Meta>+<P> — назад и <Meta>+<N> — вперед). Впрочем, несмотря на громкое название, это ничем практически не отличается от обычного пролистывания исторического списка курсорными стрелками. Что при обширной истории команд

может быть весьма утомительным. И потому для ее облегчения предусмотрена такая интересная возможность, как наращиваемый поиск (incremental search) нужной команды в буфере истории по одному (или нескольким) из составляющих ее символов.

Выполняется инкрементный поиск так: после нажатия (при пустой командной строке) клавишной комбинации <Ctrl>+<R> появляется предложение ввести алфавитный символ (или последовательность символов произвольной длины), заведомо входящий в состав требуемой команды:

```
$ bck-i-search: _
```

Ввод символа _ выведет последнюю из команд, его содержащих. При этом введенный символ будет отмечен знаком курсора. Он не обязан входить в имя команды, но может быть составляющим ее опций или аргументов (имени файла или пути к нему, например). Следующее нажатие <Ctrl>+<R> зафиксирует курсор на предыдущем символе, в пределах этой же или более ранней по списку команды. Однако вместо этого в строке поиска можно вводить дополнительные символы, детализирующие условия поиска команды (или ее опций и аргументов).

Процедуру поиска можно продолжать вплоть до достижения требуемого результата — то есть нахождения той команды, которая нужна именно сейчас. Нажатие клавиши <Enter> в любой из этих моментов запускает найденную (то есть помещенную в командную строку) команду на исполнение, с завершением поиска. Поиск обрывается также и нажатием комбинации <Ctrl>+<C>. Перед запуском найденная команда может быть отредактирована стандартными средствами, с использованием управляющих последовательностей.

Некоторые шеллы допускают чрезвычайно изощренные средства обращения с буфером истории команд. Например, в командной оболочке zsh предусмотрены способы извлечения из него отдельных командных "слов", входящих в сложные конструкции, о чем я расскажу в свое время.

Регулярные выражения

В соответствии со своей леностью разработчики POSIX-систем придумывают способы, как бы им минимизировать свои усилия. А пользователи из лени изощряются в применении этих приемов на практике. В частности, в том, как свести к минимуму набор в командной строке.

Собственно говоря, этой цели служили почти все приемы, описанные ранее. Осталось осветить немногое. А именно, регулярные выражения, реализуемые с помощью специальных символов (или метасимволов).

Элементарная задача: из каталога dir1 требуется скопировать все файлы в каталог dir2. Так неужели все они должны быть перечислены в качестве аргументов команды `cp`? Нет, нет, и еще раз нет. Ибо для этой цели придуманы шаблоны имен файлов. Самый часто используемый из них — специальный символ `*` (вроде бы я о нем уже говорил?). Он подменяет собой любое количество любых символов (в том числе — и нулевое, то есть отсутствие символов вообще). То есть для решения предложенной задачи нам достаточно дать команду:

```
$ cp dir1/* dir2
```

Чуть усложним условия: к копированию из dir1 предназначены не все файлы, а только HTML-документы, традиционно имеющие расширение html (строго говоря, в POSIX-системах нет расширений в понимании DOS, но об этом мы уже говорили в *главе 8*). Решение от этого не становится сложнее:

```
$ cp dir1/*html dir2
```

Обращаем внимание: в Linux, в отличие от DOS/Windows, шаблон `*` подменяет действительно любые последовательности символов, в том числе и точки в середине имени, то есть необходимости указывать шаблон как `*.html` нет.

Однако тут можно вспомнить, что HTML-документы могут иметь и расширение htm. Не пропустим ли мы их таким образом при копировании? Таким — безусловно, пропустим. Однако нам на помощь придет другой шаблон, символ `?`. А соответствует он любому единичному символу (или его отсутствию, то есть символу `Null`). И значит команда из примера

```
$ cp dir1/*htm? dir2
```

гарантированно охватит все возможные маски HTML-документов.

Вроде все хорошо. Однако нет: из каталога dir1 нам нужно скопировать только три определенных файла: file1, file2, file3. Не придется ли каждый из них указывать в командной строке с полным путем (а ведь они могут быть и в глубоко вложенном подкаталоге типа dir1/dir11/dir111)? Все равно не придется, на столь хитрую... постановку задачи у нас есть прием: символы группировки аргументов, обозначаемые фигурными скобками. Это на практике выглядит так:

```
$ cp path/{file1,file2,file3} dir2
```

И приведет к одноразовому копированию всех трех файлов в каталог dir2. Заметим, что сгруппированные аргументы разделяются запятыми без пробелов. И еще: в оболочке bash группируемые аргументы придется полностью вводить руками. Но вот в zsh на них распространяется возможность автодополнения, да и запятая после каждого имени появляется автоматически (и столь же автоматически исчезает при закрытии фигурной скобки).

Группировка аргументов может быть сколь угодно глубоко вложенной. Так, команда

```
$ mkdir -p dir1/{dir11/{dir111,dir112},dir12/{dir121,dir122}}
```

в один заход создаст трехуровневую структуру каталогов внутри текущего, если только не забыть про опцию `-p`, которая предписывает создавать промежуточные подкаталоги в случае их отсутствия.

И еще несколько примеров. Регулярное выражение для диапазона вида `[...]` подменяет любой из символов, заключенных в квадратные скобки. Символы эти могут даваться списком без пробелов (например, выражение `[12345]` соответствует любому символу от 1 до 5) или определяться в диапазоне, крайние значения которого разделяются дефисом без пробелов (эквивалентное первому выражение `[1-5]`). Кроме того, символ `^`, предваряющий список или диапазон, означает отрицание: выражение `[^abc]` подменяет любой символ, исключая символы `a`, `b` и `c`.

Последние примеры регулярных выражений могут показаться надуманными. Однако представим, что в том же каталоге dir1, кроме HTML-документов содержатся также файлы изображений в различных форматах — GIF, JPEG, TIFF и т. д. (традиционно имеющие одноименные расширения). И все они должны быть скопированы в каталог dir2, а вот как раз HTML-файлы нам в данный момент без надобности:

```
$ cp dir1/*[^html] dir2
```

И в каталоге dir2 окажется все содержимое каталога dir1, за исключением HTML-файлов.

Из приведенных примеров можно видеть, что метасимволы, образующие регулярные выражения, интерпретируются командной оболочкой особым образом, не так, как обычные алфавитно-цифровые символы, составляющие, скажем, имена файлов. В то же время мы уже видели (*в главе 8*), что собственно POSIX-системы накладывают на имена файлов очень мало ограничений. И в принципе система не запретит вам создать файл с именем, содержащим метасимволы. Другое дело, что работать с такими файлами может быть сложно, командная оболочка будет пытаться интерпретировать их в соответствии с правилами для регулярных выражений.

Конечно, использовать метасимволы в именах файлов весьма не рекомендуется. Однако возможны элементарные ошибки при наборе, и файлы, полученные при обмене с другими операционными системами (сами знаете, какими), могут иметь довольно непривычный (и, я даже сказал бы, неприличный) вид. Вспомним, что MS Word в качестве имени файла спокойно берет первую фразу документа. А если это вопрос? И тогда завершающий имя символ `?` будет в шелле интерпретироваться как шаблон, а не как элемент имени. Думаю, не нужно обладать очень развитым воображением, что-

бы представить последствия. Что делать в таких ситуациях? Для их разрешения резонными людьми придумано было понятие экранирования.

Маленькое отступление

Командные директивы с многочисленными их опциями и аргументами могут оказаться весьма длинными, не укладывающимися в пределы экранной строки. Правда, обычно командная оболочка по умолчанию настраивается с разрешением так называемого word wrapping (то есть переноса "слов" команды без обрыва строки). Однако перенос "слов" при этом происходит, как бог на душу положит. И в результате командная директива теряет читабельность и становится сложной для понимания.

Тут-то и приходит на помощь понятие экранирования. Знак экранирования — обратный слэш (\) — превращает символ, имеющий специальное значение (а таковым является, например, упоминавшийся ранее шаблон в именах файлов *), в самую обычную звездочку. А раз конец строки тоже символ, хотя и специальный, то и он доступен для экранирования. Так что если завершить введенный фрагмент команды обратным слэшем (некоторые оболочки требуют предварить его пробелом, и лучше так и делать, хотя в bash или zsh пробел не обязателен), после чего нажать <Enter>, то вместо попытки исполнения будет образована новая строка, в которой можно продолжать ввод. Вид приглашения к вводу при этом изменится, это будет так называемое вторичное приглашение командной строки, и его представление настраиваемо.

Возвращаемся к экранированию обратным слэшем. Действие его распространяется только на непосредственно следующий за ним символ. Если символы, могущие быть воспринятыми как специальные, идут подряд, каждый из них должен предваряться обратным слэшем.

У обратного слэша есть еще одна интересная особенность, я назвал бы ее инвертированием специального значения символов. Для примера: некая последовательность цифр (например, 033), введенная в командной строке, будет воспринята как набор обычных символов. Однако она же может выступать как код какого-либо символа (в частности, 033 — код символа <Esc> в восьмеричной системе счисления). И подчас возникает необходимость ввода таких кодов (тот же код для <Esc>, скажем, затруднительно ввести каким-либо иным образом). И вот тут обратный слэш проявляет свое инвертирующее действие: последовательность \033 будет восприниматься уже не как набор символов, а как код символа <Esc> (тут достаточно единичного слэша). Непосредственно в командной строке такой способ инвертированного экранирования, по понятным причинам, обычно не используется, но находит широкое применение в сценариях. Поэтому и запомним этот прием, он со временем потребуется нам, в частности, для русификации системы.

Есть и экраны, распространяемые на все, что заключено внутри них. Это кавычки, двойные и одинарные: большая часть символов между ними утрачивает свое специальное значение, но не все. В двойных кавычках сохраня-

ют специальное значение метасимволы `$` и `\`, а также обратные кавычки (`` ` ``), о назначении которых я скажу чуть позже. То есть в них сохраняется возможность, с одной стороны, получения значений переменных. А с другой стороны, если нам требуется дать символ `$` в его привычном значении, у нас есть возможность экранировать его обратным слэшем. И если потребуется вывести на экран сообщение "с вас, уважаемый, пятьсот долларов", то это можно сделать таким образом:

```
$ echo "с вас, уважаемый, \$500"
```

Еще одно широко применяемое использование двойных кавычек — экранирование пробелов, предотвращающих разбиение аргументов команды на отдельные "слова". Правда, в случае с командой `echo` это, как правило, не требуется (хотя настоятельно рекомендуется экранировать ее аргумент таким образом). Однако представьте, что в качестве аргумента команды копирования и перемещения выступает файл, переписанный с Windows-машины. Ведь там пробелы в именах — вещь обычная. Тут-то экранирование двойными кавычками и придется к месту.

Из сказанного понятно, почему двойные кавычки именуются еще неполными, или нестрогими: они все же допускают внутри себя использование символов со специальными значениями. В противоположность им, кавычки одинарные носят имя строгих, или полных. Потому что между ними утрачивают специальное значение все метасимволы, кроме их самих, в том числе и символ единичного экранирования. В итоге они используются там, где точно требуется отсутствие специальных символов. Если вы помните, мы применили строгие кавычки при установке псевдонимов. Они же часто оказываются обязательными при определении переменных.

Завершая тему экранирования, осталось сказать только об обратных кавычках. Их функция очень узка: они служат для экранирования команд. То есть, скажем, команда

```
$ echo date
```

в полном соответствии со своим именем просто выведет нам собственный аргумент:

```
date
```

Однако если аргумент команды закрыть обратными кавычками, то `date` будет воспринято как имя команды, подлежащей исполнению. И результат этого исполнения (то есть текущая дата и время, а именно для их получения и предназначена команда `date`) будет замещать имя команды в выводе `echo`:

```
$ echo `date`
Втр Дек 16 11:45:12 MSK 2005
```

Если вспомнить, что обратные кавычки сохраняют свое специальное значение внутри кавычек двойных, становится ясной польза от их примене-

ния: они незаменимы в тех случаях, когда требуется вывод результатов работы одной команды внутри другой. Как в нашем примере с выводом даты, если его (вывод) следует включить в некое выдаваемое командой `echo` сообщение.

Конечно, в описанном случае добиться той же цели можно было бы гораздо легче, просто командой `date`. Однако представьте, что у нас возникло желание одновременно и получить сведения о количестве пользователей в системе (для чего предназначена команда `who`). Тут-то и выясняется, что проще всего это сделать командой:

```
$ echo "На момент `date` в системе \
        зарегистрированы `who`"
```

Ответом на это будет сообщение, подобное тому, что часто можно наблюдать на главной странице многих сайтов:

```
На момент Втр Дек 16 12:11:36 MSK 2005 \
        в системе зарегистрированы alv lis
```

А теперь последнее, чем и закроем тему регулярных выражений вообще. В этом разделе рассматривалось использование метасимволов в командной оболочке (в данном случае sh, bash и zsh). В других оболочках применение метасимволов и условия их экранирования могут несколько отличаться. И к тому же, многие запускаемые из строки шелла команды могут иметь свои правила построения регулярных выражений. Так что в итоге их форма определяется сочетанием особенностей конкретной оболочки и команды, из нее запущенной. Все это по возможности будет оговариваться в дальнейшем.

Командные конструкции

Пора вернуться к генеральной линии моего рассказа — основам командного интерфейса. Надеюсь, из предшествующего изложения читателю стало ясно, что подавляющее большинство команд в POSIX-системах очень просты по сути и предназначены для выполнения какого-либо одного элементарного действия. То есть команда `cp` умеет только копировать файлы, команда `rm` — только удалять их, но зато делают они это хорошо. Подчас чересчур хорошо, что мог ощутить на себе каждый, кому "посчастливилось" по ошибке выдать директиву вроде

```
$ rm -Rf *
```

Для тех, кто не испытал этого волнительного ощущения, поясню: результатом будет полное и безвозвратное уничтожение всех файлов от текущего каталога вниз (включая подкаталоги любой степени вложенности).

Собственно, разделение любой задачи на серию элементарных операций — это и есть основной принцип работы в POSIX-системах, тот самый пресловутый UNIX-way, о котором столько говорят его приверженцы (среди них с некоторых пор числит себя и автор этих строк). Однако вслед за этапом решительного размежевания должен наступить этап объединения, как за анализом явления следует синтез эмпирических данных. И целям такого объединения служат командные конструкции.

Командные конструкции — очень важный компонент интерфейса командной строки. Они позволяют объединять несколько команд воедино и выполнять различные команды последовательно или параллельно. Для этого служат специальные символы — операторы: фонового режима, объединения, перенаправления и конвейеризации.

Простейшая командная конструкция — это выполнение команды в фоновом режиме, что вызывается вводом символа амперсанда после списка опций и (или) аргументов:

```
$ command [options] [arguments] &
```

В bash и некоторых других оболочках пробел перед символом амперсанда (`&`) не обязателен, но в некоторых шеллах он требуется, и потому лучше возвести его ввод (как и во всех аналогичных случаях) в ранг привычки. После этого возвращается приглашение командной строки, и возможен ввод любых других команд (в том числе и фоновых). Команды для последующего исполнения можно задать и в той же строке:

```
$ command1 & command2 & ... & commandN
```

В результате все команды, кроме указанной последней, будут выполняться в фоновом режиме.

Существуют и конструкции для последовательного выполнения команд. Так, если ряд команд разделен в строке символом точки с запятой:

```
$ command1 ; command2 ; ... ; commandN
```

то сначала будет выполнена команда `command1`, затем — `command1` и т. д. (опять-таки, обрамление `;` пробелами не обязательно во многих командных оболочках). Сами по себе команды не обязаны быть связанными между собой каким-либо образом: в сущности, это просто эквивалент последовательного их ввода в командной строке:

```
$ command1
$ command2
...
```

и т. д. При этом первая команда может, например, копировать файлы, вторая будет осуществлять поиск, третья — выполнять сортировку или другие

действия. Очевидно, что в общем случае выполнение последующей команды не зависит от результатов работы предшествующей.

Однако возможна ситуация, когда результаты предыдущей команды из такой конструкции используются в команде последующей. В этом случае ошибка исполнения любой составляющей команды, кроме последней, делает невозможным продолжение работы всей конструкции. Это само по себе было бы еще полбеды, однако в некоторых ситуациях исполнение последующей команды возможно только при условии успешного завершения предыдущей.

Характерный пример — сборка программы из ее исходных текстов, включающая три стадии — конфигурирование, собственно компиляцию и установку собранных компонентов. Что выполняется (несколько забегу вперед) последовательностью из трех команд:

```
$ ./configure
$ make
$ make install
```

Ясно, что если конфигурирование завершилось ошибкой, то компиляция начаться не сможет и, соответственно, потом нечего будет устанавливать. И потому объединение их в последовательную конструкцию вида

```
$ ./configure ; make ; make install
```

может оказаться нецелесообразным.

Однако для предотвращения таких ситуаций в конструкции из взаимосвязанных команд существует другой оператор, обозначаемый удвоенным символом амперсанда — `&&`. Он указывает, что последующая команда конструкции должна исполняться только в том случае, если предыдущая завершилась успешно:

```
$ ./configure && make && make install
```

На практике обе приведенные в качестве примера конструкции дадут один и тот же результат. Однако в ряде иных случаев различие между этими конструкциями может быть существенным.

Впрочем, предусмотрена и командная конструкция, в которой последующей команде предписано исполняться в том и только в том случае, если предыдущая команда завершилась неудачно. Она имеет вид

```
$ command1 || command2
```

и может служить, в частности, для вывода сообщений об ошибках. Конечно, можно найти ему и другие применения (как, впрочем, и оператору `&&`), но это уже далеко выходит за рамки нашего элементарного введения.

Следующая командная конструкция — это так называемое перенаправление ввода/вывода. Тут тоже нам придется несколько забежать вперед, однако

перенаправление — слишком практически важный прием, чтобы отложить его рассмотрение его до выяснения прочих обстоятельств.

И потому вкратце: любая команда получает данные для своей работы (например, список опций и аргументов) со стандартного устройства ввода (которым в первом приближении будем считать клавиатуру), а результаты своей работы представляет на стандартном устройстве вывода (коим договоримся считать экран монитора). А совсем-совсем недавно (*из главы 8*) мы узнали, что в POSIX-системах любое устройство — не более чем имя специального файла, именуемого файлом устройства. И, таким образом, ничто не запрещает нам подменить специальный файл устройства ввода или устройства вывода любым иным файлом (например, обычным текстовым). Откуда и будут в этом случае браться входные данные или куда будет записываться вывод команды.

Перенаправление вывода команды обозначается следующим образом:

```
$ command > filename
```

или

```
$ command >> filename
```

В первом случае (одиночный символ `>`) вывод команды `command` образует содержимое нового файла с именем `filename`, не появляясь на экране. Или, если файл с этим именем существовал ранее, то его содержимое подменяется выходным потоком команды (точно так же, как при копировании одного файла в другой, уже существующий). Поэтому такое перенаправление называется замещающим (или перенаправлением в режиме замещения).

Во втором же случае (двойной символ `>>`) происходит добавление вывода команды `command` в конец существующего файла `filename` (при отсутствии его в большинстве случаев просто образуется новый файл). И потому это называется присоединяющим перенаправлением, или перенаправлением в режиме присоединения.

Перенаправление ввода выглядит так:

```
$ command < filename
```

Конечно, теоретически можно представить себе и присоединяющее перенаправление ввода, однако практически оно вряд ли может найти применение.

Простейший случай перенаправления — вывод результата исполнения команды не на экран, а в обычный текстовый файл. Например, конструкция

```
$ ls dir1 > list
```

создаст файл, содержанием которого будет список файлов каталога dir1. А в результате выполнения конструкции

```
$ ls dir2 >> list
```

к этому списку добавится и содержимое каталога dir2.

При перенаправлении ввода команда получает данные для своей работы из входящего в командную конструкцию файла. Например, конструкция

```
$ sort < list
```

выведет на экран строки файла list в порядке возрастания значения ASCII-кода первого символа, а конструкция

```
$ sort -r < list
```

осуществит сортировку строк того же файла в порядке, обратном алфавитному (вернее, обратном порядку кодов символов, но это нас в данном случае не волнует).

В одной конструкции могут сочетаться перенаправления ввода и вывода, как в режиме замещения, так и в режиме присоединения. Так, конструкция

```
$ sort -r < list > list_r
```

не только выполнит сортировку строк файла list в обратном алфавитному порядке (что предписывается опцией `-r`, происходящей в данном случае от reverce), но и запишет ее результаты в новый файл list_r, а конструкция

```
$ sort -r < list >> list
```

добавит по-новому отсортированный список в конец существующего файла list.

Возможности построения командных конструкций не ограничиваются перенаправлением ввода/вывода: результаты работы одной команды могут быть переданы для обработки другой команде. Это достигается благодаря механизму программных каналов (pipe, или конвейеров — последний термин лучше отражает существо дела).

При конвейеризации команд стандартный вывод первой команды передается не в файл, а на стандартный ввод следующей команды. Простой пример такой операции — просмотр списка файлов:

```
$ ls -l | less
```

Перенаправление вывода команды `ls`, то есть списка файлов, который при использовании полного формата записи (опция `-l`) может занимать многие экраны, на ввод команды `less` позволяет просматривать результат с ее помощью постранично или построчно в обоих направлениях.

Конвейеризация команд может быть сколь угодно длинной. Возможно также объединение конвейеризации команд и перенаправления в одной конструкции. Кроме того, команды в конструкции могут быть сгруппированы с тем, чтобы они выполнялись как единое целое. Для этого группа команд разделяется символами точки с запятой и пробелами, как при последовательном выполнении команд, и заключается в фигурные скобки. Так, если

нам требуется перенаправить вывод нескольких команд в один и тот же файл, вместо неуклюжей последовательности типа

```
$ command1 > file ; command2 >> file ; ... ; commandN >> file
```

можно прибегнут к более изящной конструкции:

```
$ { command1 ; command2 ; ... ; commandN } > file
```

Как и многие из ранее приведенных примеров, этот может показаться надуманным. Однако представьте, что вам нужно создать полный список файлов вашего домашнего каталога, разбитый по подкаталогам, да еще и с комментариями, в каком подкаталоге что находится. Конечно, можно вывести состав каждого подкаталога командой `ls`, слить их воедино командой `cat` (она предназначена, в частности, и для объединения файлов), загрузить получившееся хозяйство в текстовый, где добавить необходимые словеса. А можно обойтись единой конструкцией:

```
$ { echo "List of my files" ;  > echo "My text" ; \
       ls text/* ;  > echo "My images" ; \
       ls images/* ;  > echo "My audio" ; \
       ls audio/* ;  > echo "My video" ; \
       ls video/* } > my-filelist
```

И в результате получить файл такого (условно) содержания, которое мы для разнообразия просмотрим с помощью только что упомянутой команды `cat` (благо, и для просмотра содержимого файлов она также пригодна):

```
$ cat my-filelist
List of my files
My text
text/text1.txt text/text2.txt
My images
images/img1.tif images/img2.tif
My audio
audio/sing1.mp3 audio/sing2.mp3
My video
video/film1.avi video/film2.avi
```

С понятием командных конструкций тесно связано понятие программ-фильтров. Это команды, способные принимать на свой ввод данные с вывода других команд, производить над ними некоторые действия и перенаправлять вывод (то есть результат модификации полученных данных) в файлы или другой команде. Программы-фильтры — очень эффективное средство обработки текстов, и в свое время мы к ним вернемся для подробного изучения.

Сценарии оболочки: первые представления

Наш затянувшийся разговор о командах и командном интерфейсе подходит к концу. Но это тот самый случай, когда из песни слова не выкинешь. Напротив, очень многое осталось недосказанным или необъясненным. Что ж, тем больше поводов будет у нас возвращаться к теме команд вновь и вновь.

А в заключение этого раздела — еще немного терпения. Потому что было бы несправедливо не уделить чуть-чуть места тому, что придает командному интерфейсу POSIX-систем его несравненную гибкость и универсальность. Итак, слово о сценариях оболочки.

В самом начале я обмолвился, что шелл — это не просто среда для ввода единичных команд и командных конструкций, но и еще интерпретатор собственного языка программирования. Так вот, сценарии оболочки, именуемые также скриптами, это и есть программы, написанные на этом языке.

На протяжении последующих глав нам то и дело придется рассматривать кое-какие примеры готовых сценариев, а подчас и пытаться создавать их собственноручно. Ибо занятие это в элементарном исполнении навыков программирования не требует вообще.

В самом простом случае сценарий — это просто одна или несколько команд или командных конструкций с необходимыми опциями и аргументами в виде обычного именованного текстового файла.

Сценарии в UNIX-системах несколько напоминают batch-файлы, памятные многим по временам MS-DOS, или макросы, знакомые пользователям любых развитых прикладных пакетов. Подобно тем и другим, они предназначены в первую очередь для автоматизации часто исполняемых рутинных операций. В частности, именно они позволяют избежать ввода длинных последовательностей в командной строке.

Однако значение сценариев не исчерпывается удовлетворением естественной человеческой лени. Они позволяют решать очень широкий круг задач, от конфигурирования системы (в частности, файлы, отвечающие за инициацию любой POSIX-системы, суть сценарии оболочки, почему и называются сценариями инициализации, или стартовыми скриптами) до создания весьма сложных приложений (к классу шелл-скриптов относятся многие менеджеры пакетов) и даже интерактивных Web-страниц (даже CGI-скрипты никто не запрещает писать на языке командной оболочки). Именно практически неограниченным возможностям создания пользовательских сценариев интерфейс командной строки обязан своей эффективностью.

Создание пользовательского сценария просто. Для этого всего и нужно:

- ❒ создать командную конструкцию, достойную увековечивания;
- ❒ поместить ее в простой текстовый файл;
- ❒ по потребности и желанию снабдить комментариями;
- ❒ тем или иным способом запустить файл на исполнение.

С принципами создания команд и командных конструкций мы в первом приближении разобрались (или пытались разобраться) раньше. А вот способов помещения их в файл существует множество. Можно просто ввести (или вызвать из буфера истории) нужную команду и оформить ее как аргумент команды `echo`, вывод которой перенаправить в файл:

```
$ echo "cp -rf workdir backupdir" > mybackup
```

Таким образом, мы получили простейший скрипт для копирования файлов из рабочего каталога в каталог для резервного хранения данных, что впредь и будем проделывать регулярно (не так ли?).

Аналогичную процедуру можно выполнить с помощью команды `cat`: она, оказывается, способна не только к объединению файлов и выводу их содержимого, но и к вводу в файл каких-либо данных. Делается это так. Вызываем `cat` с перенаправлением ее вывода в файл:

```
$ cat > myarchive
```

и нажимаем <Enter>. После этого команда остается в ожидании ввода данных для помещения их в новообразованный файл. Не обманем ее ожиданий и проделаем это. Причем можно не жаться и выполнить ввод в несколько строк, например:

```
cd $HOME/archivedir tar cf archive.tar \
        ../workdir gzip archive.tar
```

Завершив ввод тела скрипта, все той же клавишей <Enter> открываем новую строку и набираем комбинацию <Ctrl>+<D>, выдающую символ окончания файла.

В результате получаем сценарий для архивирования в специально предназначенном для этого каталоге archivedir наших рабочих данных (командой `tar`), а заодно и их компрессии (командой `gzip`). В UNIX, в отличие от DOS/Windows, архивирование и компрессия обычно рассматриваются как разные процедуры.

Наконец, сценарий можно создать в любом текстовом редакторе, но это не так интересно (по крайней мере, пока). Да и стоит ли вызывать редактор ради двух-трех строк?

Комментариями в шелл-сценариях считаются любые строки, начинающиеся с символа решетки (#): они не учитываются интерпретатором и не прини-

маются к исполнению (хотя комментарий может быть начат и внутри строки, важно только, что между символом # и концом команд больше ничего не было бы). Ясно, что комментарии — элемент для скрипта не обязательный, но очень желательный, хотя бы для того, чтобы не забыть, ради чего этот сценарий создавался.

Одна строка, начинающаяся символом решетки, в сценарии практически обязательна. И должна она быть первой и выглядеть следующим образом:

```
#!/path/shell_name
```

В данном случае восклицательный знак подчеркивает, что предваряющий его символ решетки не комментарий, а указание (sha-bang) на точный абсолютный путь к исполняемому файлу оболочки, для которой наш сценарий предназначен, например,

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

для POSIX-шелла или

```
#!/bin/bash
```

для оболочки bash. Здесь следует подчеркнуть, что шелл, для которого предназначается сценарий, отнюдь не обязан совпадать с командной оболочкой пользователя. И для полноты картины замечу, что указание точного имени интерпретатора требуется не только для шелл-скриптов, но и для программ на любых языках сценариев (типа Perl или Python).

Так что по-хорошему в обоих приведенных примерах ввод команд сценария следовало бы предварить строкой sha-bang. Конечно, отсутствие имени командной оболочки в явном виде обычно не мешает исполнению шелл-сценария, для этого будет вызван командный интерпретатор по умолчанию, то есть /bin/sh во FreeBSD или /bin/bash в Linux (файл /bin/sh, который обязан присутствовать в любой POSIX-системе, в Linux представляет собой символическую ссылку на /bin/bash и предназначен для имитации POSIX-шелла).

Однако если сценарий предназначен для другой командной оболочки, то без sha-bang он может исполняться неправильно (а если оболочка еще и не из семейства совместимых с POSIX shell, — то не исполняться вообще).

Следует еще и воздержаться от комментария в первой строке: в некоторых случаях он может быть интерпретирован как вызов конкретной оболочки (в частности, csh или tcsh). Лучше оставить первую строку свободной или начать сценарий с "пустой" команды

```
:
```

не совершающей никакого действия.

Теперь остается только выполнить наш сценарий. Сделать это можно разными способами. Можно непосредственно вызвать требуемый шелл как

обычную команду, снабдив его аргументом — именем сценария (предположим, что он находится в текущем каталоге):

```
$ bash scriptname
```

Далее, для вызова скриптов существует специальная встроенная команда оболочки, обозначаемая символом точки. Используется она аналогично:

```
$ . ./scriptname
```

с тем только исключением, что тут требуется указание текущего каталога в явном виде (что и символизирует `./`).

Однако наиболее популярный способ запуска сценариев — это присвоение его файлу так называемого атрибута исполнения. Что это такое, с чем этот атрибут едят и как присваивают, об этом мы уже говорили в *главе 8*. И тут только вспомним, что присвоение атрибута (или, как еще говорят, бита) исполнения — та самая процедура, которая волшебным образом превращает невзрачный текстовый файл во всамделишную (хотя и очень простую) программу.

Так вот, после присвоения нашему сценарию бита исполнения запустить его можно точно так же, как любую другую команду — просто из командной строки:

```
$ ./scriptname
```

Опять же, в предположении, что сценарий находится в текущем каталоге (`./`), иначе потребуется указание полного пути к нему. Все сценарии помещаются в специально отведенный для этого каталог (например, $HOME/bin), который и добавляется в качестве еще одного значения переменной `PATH` данного пользователя.

Понятие о функциях

И уж совсем в заключение этой главы осталось сказать только пару слов о функциях командной оболочки. Это такая же последовательность команд (или даже просто одиночная команда), как и сценарий, но не (обязательно) вынесенная в отдельный исполняемый файл, а помещенная в тело другого скрипта. В нем она опознается по имени и может быть выполнена неоднократно в ходе работы.

Главное отличие функции от сценария в том, что она выполняется в том же процессе (и, соответственно, экземпляре шелла), что и заключающий сценарий. Тогда как для каждого скрипта, вызываемого из другого сценария, создается отдельный процесс, порождающий собственный экземпляр шелла. Это может быть важным, если в сценарии определяются некоторые переменные, которые имеют силу только в нем самом.

Функции не обязательно помещаются внутрь сценария, их можно собрать в некоторые отдельные файлы, которые именуются библиотеками функций и могут быть использованы по мере надобности.

Ранее на протяжении всего повествования неоднократно упоминались (и будут упоминаться впредь) системные библиотеки, в частности, главная библиотека glibc. Так вот, это точно такие же сборники функций, правда, не командной оболочки, а языка Си, хранящиеся не в виде текстовых файлов, а в бинарном, откомпилированном, виде.

Самая главная команда

Возможно, этот раздел следовало бы назвать главой и поместить в самое начало книги. Ибо содержание его — не информация о тех или иных командах, или свойствах системы, а метаинформация: информация о том, как получить нужную информацию. Раздел посвящен выработке некоторых навыков, которые у истинного POSIX'ивиста должны быть доведены до уровня рефлексов. Однако лучше поздно, чем никогда, так что приступим.

Как как можно догадаться по прочтении предшествующих разделов этой главы, команд в UNIX огромное количество. В только что установленной Linux-системе минималистского типа (вроде CRUX или Archlinux) их может быть штук 500—700, в минимальной установке BSD — около 800. И это все без учета Иксов и всяческих приложений.

К слову сказать, а как определить количество команд? Есть несколько способов, зависящих от используемой ОС, дистрибутива, командной оболочки. Например, во всех BSD-системах все команды из базового комплекта собраны в каталогах /bin, /sbin, /usr/bin и /usr/sbin, так что достаточно просто подсчитать количество входящих в них файлов, Например, вот так:

```
$ ls /bin /sbin /usr/bin /usr/sbin | wc
```

Как я уже говорил, во FreeBSD пятой ветки такой подсчет даст результат: 800—850 команд, в зависимости от версии и полноты установки. В Linux размещение исполняемых файлов базовой системы зависит от дистрибутива, однако в первом приближении такой способ подойдет и здесь.

Если же учесть, что каждая команда имеет опции, да подчас также в немалом числе, возникает естественный вопрос: как нормальный человек все это может запомнить? Да никак, — последует ответ. Потому что запоминать все это изобилие команд нет не только возможности, но и ни малейшей необходимости.

Во-первых, команд, которые нужны постоянно, ежедневно и по много раз на дню, не так уж и много. И практически все эти команды имеют прозрачную (правда, английскую) этимологию или представляют собой простую

аббревиатуру от слов, обозначающих соответствующее действие: `ls` — от list, `cp` — от copy, `mv` — от move, `rm` — от remove и т. д. Так что тут в запоминании может помочь не столько какой-либо специализированный источник, сколько элементарный англо-русский словарь.

Во-вторых, и главных, вовсе не нужно помнить все команды, и тем более все их опции: гораздо важнее понимать, каким образом соответствующую информацию можно получить в нужный момент. И вот тут возможны варианты.

Для начала, каким образом можно узнать, какие команды имеют место быть в нашей системе? В первом приближении этому послужит клавиша табуляции: нажав ее в пустой командной строке, мы (в большинстве случаев) получим сообщение вроде такого:

```
do you wish to see all XXX possibilities (YYY lines)?
```

И если согласимся с этим предложением (нажав клавишу <y>), то нашему взору предстанет все доступное изобилие команд.

Конечно, с некоторыми оговорками. Клавиша табуляции не извлекает волшебным образом имена команд откуда-то из загашников системы. Нет, она просто выводит список всех исполняемых файлов (для которых установлен атрибут исполнения, *см. главу 8*), расположенных в каталогах, перечисленных как значения переменной `PATH`.

Из чего следует, что, во-первых, список этот будет разным для обычного пользователя и для администратора. Потому что значения переменной `PATH` у них, как правило, по умолчанию разные (или должны быть таковыми в правильно настроенной системе). Для пользователя в ответ на команду

```
$ echo $PATH
```

обычно можно увидеть список вроде

```
/bin /usr/bin /usr/X11R6/bin /usr/local/bin
```

тогда как для root к нему добавятся пути вроде

```
/sbin /usr/sbin /usr/local/sbin
```

Во-вторых, как уже упоминалось, что одна и та же команда может иметь более чем одно имя — например, `/bin/bash` и `/bin/sh`. Эти синонимичные имена (их не следует смешивать с псевдонимами, те в списке по нажатию табулятора не встречаются) представляют собой жесткие или символические ссылки на одну и ту же программу. Создавая таким образом иллюзию, будто команд в системе больше, чем на самом деле.

В-третьих, получаемый по нажатию табулятора список включает только файлы из каталогов, стандартно перечисляемых в переменной `PATH`. И некоторые программы, которые во многих современных дистрибутивах Linux часто норовят установиться в каталоги вида /opt/pkg_name, вполне могут и

не попасть в список, если значения переменной PATH не скорректированы должным образом.

Однако первое представление о наличных командах, тем не менее, с помощью табулятора получить можно. На худой конец, всегда есть возможность просто просмотреть содержимое каталогов и исполняемыми файлами, как было сказано ранее:

```
$ ls /{bin,sbin} /usr/{bin,sbin} /usr/local/{bin,sbin}
```

Остается установить только, что каждая из наличествующих команд делает, не всегда же можно определить ее функции по англо-русскому словарю. И тут нам на помощь приходит самая главная команда — команда man.

Команда man предназначена для вызова экранной документации в одноименном формате (Manual Pages). А такая man-документация почти обязательно сопровождает любую уважающую себя программу для POSIX-систем. И устанавливается в принудительном порядке при инсталляции соответствующей программы в любом случае, развертывается ли она из бинарного тарбалла или собирается из исходников.

Для файлов man-документации предназначены специальные каталоги. Обычно это /usr/share/man (иногда /usr/man) для man-страниц базовой системы и штатных компонентов дистрибутива, usr/X11R6/man — для документации по оконной системе Икс, usr/local/man — для документации к программам, самостоятельно собираемых пользователем (или устанавливаемых из портов). Впрочем, местоположение man-страниц может быть различным. Однако в большинстве случаев его легко определить, просмотрев значения специальной переменной — MANPATH:

```
$ echo $MANPATH
```

Это даст примерно такую картину:

```
/usr/man:/usr/share/man:/usr/local/man:/usr/X11R6/man
```

Man-страницы в любой POSIX-системе разделены на восемь нумерованных групп, каждая из которых размещена в собственном подкаталоге: /usr/share/man1, /usr/share/man2 и т. д. Назначение этих групп следующее:

1) man1 — команды и прикладные программы пользователя;
2) man2 — системные вызовы;
3) man3 — библиотечные функции;
4) man4 — драйверы устройств;
5) man5 — форматы файлов;
6) man6 — игры;
7) man7 — различные документы, не попадающие в другие группы (в том числе относящиеся к национальной поддержке);
8) man8 — команды администрирования системы.

В BSD к ним добавляется еще и девятая группа man9 — man-страницы для разработчиков ядра системы.

Нас, как пользователей, в данный момент интересуют в первую очередь команды из первой и восьмой групп, хотя и об остальных категориях забывать не след, иногда позарез нужные сведения отыскиваются в самой неожиданной из них.

Внутри групповых подкаталогов можно увидеть многочисленные файлы вида filename.#.gz. Последний суффикс свидетельствует о том, что man-страница упакована компрессором gzip (что бывает не всегда, но как правило). Цифра после имени соответствует номеру группы (то есть в подкаталоге ~/man1 это всегда будет единица). Ну, а имя man-страницы совпадает с именем команды, которую она описывает. Если, конечно, речь идет о команде: в разделе 2 оно будет образовано от соответствующего системного вызова, в разделе 2 — от имени функции и т. д. Но пока нас интересует только информация о командах, так что дальше я этого оговаривать не буду.

Для вызова интересующей документации требуется дать команду `man` с аргументами (номером группы и именем man-страницы), например:

```
$ man 1 ls
```

Причем номер группы необходим только в том случае, если одноименные документы имеются в разных группах. Для пользовательских команд он обычно не нужен, так как все равно просмотр групповых каталогов идет сначала в man1, затем в man8, и только потом во всех остальных (в порядке возрастания номеров).

Например, в группе 1 имеется man-страница tty (1) (к слову сказать, ссылаться на man-страницы принято именно так — с указанием номера группы в скобках), посвященная одноименной команде, а в группе 4 — man-страница tty (4), описывающая драйвер для устройства /dev/tty. Поэтому пользователь, нуждающийся в помощи по указанной команде, может ограничиться формой

```
$ man tty
```

Если же ему требуется получить сведения о драйвере устройства /dev/tty, номер группы должен быть указан в явном виде:

```
$ man 4 tty
```

Однако повторяю, в данный момент нас интересуют только команды. Так что для получения информации, например, по команде `ls` достаточно ввести следующее:

```
$ man ls
```

После этого можно будет лицезреть примерно такую картину:

```
LS(1)  FSF    LS(1)

NAME

       ls — list directory contents

SYNOPSIS

       ls [OPTION]... [FILE]...

DESCRIPTION

       List  information  about

       the FILEs (the current directory by default).

       Sort entries alphabetically if none

       of -cftuSUX nor --sort.
```

То есть в начале man-страницы даются имя команды, которую она описывает (`ls`), ее групповая принадлежность (`1` — пользовательские команды) и авторство (в данном случае `FSF`, Free Software Foundations), или название системы. После чего обычно дается обобщенный формат вызова (SYNOPSIS — перечисление всех возможных опций и аргументов) и краткое описание.

Следующая, основная, часть man-страницы — описание опций команды и условия их применения. Далее иногда (но, к сожалению, не всегда) следуют примеры использования команды (Examples) в разных типичных ситуациях. В заключение, как правило, даются сведения о найденных ошибках (Bug Report) и приведен список man-страниц, тематически связанных с данной (See also), с указанием номера группы, к которой они принадлежат, иногда — историческая справка, а также указываются данные об авторе.

Большинство man-страниц занимают более одного экрана. В этом случае возникает необходимость перемещения по экранам и строкам — навигация. Впрочем, сама по себе команда `man` не отвечает не только за навигацию по странице, но даже за ее просмотр. Для этой цели она неявным образом вызывает специальную программу постраничного просмотра (pager). В Linux таковым по умолчанию выступает уже известная нам команда `less`, во FreeBSD — `more` (впрочем, здесь это жесткие ссылки на одну и ту же программу). Хотя пользователь может определить для себя какой-либо другой pager, это такая же переменная, как и пользовательская оболочка или редактор, и устанавливается точно так же:

```
export PAGER=more
```

в POSIX-совместимых шеллах, или

```
setenv PAGEmore
```

в C-shell и его производных.

Однако не будем оригинальничать, сохранив `less` в качестве средства просмотра man-страниц (лично я предпочитаю ее, вызываемую в формате `more`, для чего определяю `alias less='less -M'`). В этом случае с навигационными ее возможностями можно ознакомиться, нажав клавишу <h> — вызов встроенной помощи команды `less`. Из нее мы и узнаем, что перемещаться по man-странице можно с помощью управляющих последовательностей, сходных в принципе с теми, с которыми мы ознакомились в предыдущей главе.

Управляющие последовательности команды `less` для большинства навигационных действий весьма разнообразны, но в принципе разделяются на две группы: буквенные и состоящие из комбинаций <Ctrl> и литеры. Так, переместиться на одну строку вперед можно просто нажатием клавиши <j>, на одну строку назад — клавиши <k>, сместиться на экранную страницу — с помощью клавиш <f> (вперед) и <b> (назад). Однако того же результата можно добиться комбинациями клавиш <Ctrl>+<n> и <Ctrl>+<p> для построчного перемещения и <Ctrl>+<v> — для постраничного.

Аналогично и для большинства других действий (смещение на половину экранной страницы, например: <Ctrl>+<D> и <d> — вперед, <Ctrl>+<U> и <u> — назад) можно обнаружить минимум одну альтернативную пару управляющих последовательностей. Регистр символов обычно значения не имеет. Одно из исключений — нажатие клавиши <g> перемещает к первой строке man-страницы, клавиши <G> — к последней.

Наличие двух типов управляющих последовательностей может показаться излишним усложнением, однако имеет глубокое внутреннее обоснование. Правда, для объяснения его придется существенно забежать вперед.

Я надеюсь, что со временем мы доберемся до такой штуки, как текстовые редакторы. Тема эта большая и животрепещущая, служащая предметом священных войн, в которые мы, впрочем, вступать не будем. В рамках нынешнего разговора отметим только, что, за исключением некоторых отщепенцев (в числе коих и автор этих строк), подавляющее большинство записных юниксоидов пользуются одним из двух редакторов — vi (и его клонами типа Vim) или emacs (включая вариации типа Xemacs).

Оба эти редактора относятся к категории командных. То есть все действия по редактированию осуществляются в них обычно не выбором пунктов из меню, а прямыми командными директивами, примерно как в командной строке оболочки. Так вот, одно из кардинальных различий между линиями vi и emacs — различие управляющих последовательностей для навигации по тексту и его редактированию. Стиль навигации в vi основан на однобуквенных командных аббревиатурах (команды типа `j` или `k` пришли в `less` именно оттуда). Стиль emacs же подразумевает последовательности, образо-

ванные сочетанием клавиши <Ctrl> и различных алфавитно-цифровых комбинаций.

Поскольку эффективное использование любого редактора командного стиля подразумевает доведенное до автоматизма использование управляющих последовательностей, переключение с vi-стиля на стиль emacs в этом деле может быть просто мучительным. Вот и предусмотрели разработчики возможность использования и того, и другого стиля — кто к чему привык.

Раз уж зашла речь о стилях управляющих последовательностей... В большинстве командных оболочек такое переключение между стилями управления также возможно. Только не параллельное, а альтернативное. И устанавливается оно в конфигурационных файлах пользовательского шелла.

Возвратимся, однако, к нашей man-документации. Для навигации по странице можно использовать и обычные клавиши управления курсором, клавиши <PgUp>,<PgDown>, а также некоторые другие. Например, нажатие клавиши <Enter> приводит к смещению на одну строку вперед (аналогично клавише <Down>, а клавиши пробела — на один экран вперед (подобно <PgDown>).

Однако это не лучший способ навигации. Потому что управляющие последовательности (независимо, в стиле ли vi, или в стиле emacs) обладают дополнительной возможностью: они понимают числовые аргументы. То есть если мы нажмем клавишу с цифрой <5>, а вслед за ней — клавишу <J>, то мы сместимся на пять строк вперед, комбинация <3>+<K> — на три страницы назад и т. д.

Есть и возможность поиска внутри man-страницы. Для этого нажимается клавиша прямого слэша </>, после чего вводится искомое слово (последовательность символов). Для выхода из просмотра man-страницы предусмотрена клавиша <q>. Кроме того, можно использовать и почти универсальную комбинацию для прекращения выполнения программ <Ctrl>+<C>.

Заканчивая разговор об управляющих последовательностях, еще раз подчеркну: все они относятся не к самой команде `man`, а к той программе-пейджеру, которая ею вызывается для просмотра.

Обращение к man-страницам позволяет получить практически исчерпывающую информацию по любым командам, но только в том случае, если пользователь знает название той команды, которая требуется в данном случае. А если он только в общих чертах представляет, что эта команда должна делать? Что ж, тогда можно прибегнуть к поиску man-страниц по ключевым словам, отражающим требуемые функции. Чему призвана служить опция `-k` команды `man`. Например, директива

```
$ man -k print
```

выведет на экран список всех man-страниц, описывающих команды, имеющие отношение к печати (причем не только на принтере, но и к выводу на экран, по-английски подчас это тоже будет обозначаться как print).

Исчерпывающим руководством по использованию системы Manual Pages является ее собственная man-страница. Доступ к ней осуществляется по команде

```
$ man man
```

Она выводит на экран man-страницу, содержащую описание команды `man` (эка загнул, а?):

```
MAN(1) FreeBSD General Commands Manual     MAN(1)


NAME
       man -- format and display the on-line \
       manual pages


SYNOPSIS
       man [-adfhkotw] [-m machine] \
       [-p string] [-M path] [-P pager] \
       [-S list] [section] name ...


DESCRIPTION
     Man formats and displays the on-line manual pages.
       ...
```

Система man-страниц имеет три кардинальных недостатка. Первый, о котором я уже говорил: она даст ответ только в том случае, если пользователь знает, как и о чем ее спрашивать. К сожалению, недостаток этот не устраним. Вернее, устранить его можно только чтением всякого рода вводных статей и книг (например, этой). А также, конечно, тех же руководств — в попытках постичь заложенную в них сермяжную правду. Уверяю, что момент истины рано или поздно наступит...

Второй недостаток в том, что подавляющее большинство man-страниц входит в состав дистрибутивов (Linux ли, Free- или прочих BSD) только в английском варианте. Конечно, в составе отечественных Linux-дистрибутивов можно найти много переведенных man-страниц, однако часто они существенно устарели по сравнению с оригиналом. И потому неизбежно приходится читать их на языке Вильяма Шекспира.

Что, впрочем, не так уж и страшно даже для тех, кто не очень свободно читает Шекспира в подлиннике. В свое время я обнаружил, что переве-

денные man-страницы подчас менее понятны, чем оригинальные. И вообще, от общения с документацией на тех языках, из которых я знаком только с ненормативной лексикой (типа испанского), у меня сложилось впечатление, что абсолютно неважно, на каком языке написанное данное пособие — лишь бы алфавит был понятен. Иначе говоря, POSIX'ивист POSIX'ивиста всегда поймет. На крайний случай, Виктор Вислобоков ведет большой проект по полному переводу всех man-страниц Linux на русский язык (**http://linuxshare.ru/projects/trans/mans.html**). А специфические для FreeBSD страницы в переводе можно найти здесь: **http://ln.ua/~openxs/projects/man/freebsd/index.html**.

Третий же и, пожалуй, главный недостаток — сложность навигации внутри объемных man-страниц и невозможность ориентации внутри группы связанных по смыслу документов — призвана устранить система info. Она принята в качестве стандартной для программ проекта GNU. И многие из них в полном объеме документированы только в этом формате.

К сожалению, о системе info-страниц я мало чего могу сказать, так как сам ею почти не пользуюсь. И потому за более подробными сведениями вынужден послать читателя к другим руководствам. Из которых первейшим, разумеется, будет команда:

```
$ info info
```

Документация в форматах man и info — далеко не единственный источник информации о POSIX-системах и их приложениях. Так что, если в вашей системе не обнаружится дополнительных источников информации, стоит поискать их на сайте разработчиков интересующей вас программы...

Подводя итог, сформулируем, наконец, одну из главных истин POSIX'ивизма: читайте документацию, ибо сеет она разумное, доброе, вечное. Если же нужные документы не валяются под ногами, ищите и да обрящете (с вероятностью, близкой к 100%).

Интермедия: команды обработки текстов

- Вступление.
- Просмотр файлов.
- Сравнение, объединение и деление файлов.
- Поиск в файлах.
- Sed: средство потокового редактирования.

В одной из предшествующих интермедий (после *главы 8*) речь шла о командах, которые манипулируют файлами как целыми, не затрагивая их содержания (и в общем случае от такового не зависящими). Ныне же речь пойдет о командах, создающих и изменяющих внутреннее содержание файлов.

Вступление

Конечно, само по себе манипулирование файлами (копирование, перемещение и т. д.) также подразумевает изменение содержания некоторых файлов, но только одного-единственного типа (а именно — каталогов), однако собственно внутренняя сущность обычных файлов при этом не изменяется. Предметом же настоящей интермедии будут штатные средства POSIX-систем, позволяющие в той или иной мере учитывать контент файлов и манипулировать им. Разумеется, манипулирование контентом возможно только для регулярных файлов. При этом многие их разновидности (бинарные файлы, файлы графических форматов и word-процессоров) требуют для изменения своего содержания специальных средств — а именно, компиляторов и прикладных программ, в которых они создавались. Однако здесь о них разговора не будет — ибо целью моей было продемонстрировать мощь обычных команд для решения многих пользовательских задач. Правда, на самом деле команды модификации контента действенны преимущественно для файлов текстовых.

Однако круг объектов таких команд не столь уж узок, как может показаться. Ведь именно в виде обычных текстовых файлов в ОС POSIX-семейства хранится масса общесистемной информации, исполняемых сценариев, баз данных атрибутов самых разных объектов. Далее — собственно нарративные тексты любого содержания: ведь чисто текстовый формат для них куда роднее, чем всякого рода `*.doc` и `*rtf`. Ну и никем не возбраняется использо-

вать такие команды в отношении текстов с разметкой — HTML ли, XML, TeX или еще чего. Так что поле приложения рассматриваемых команд — достаточно обширно.

Просмотр файлов

Однако прежде чем как-то манипулировать контентом файлов, желательно этот самый контент некоторым образом просмотреть. И тут можно вспомнить о команде `cat`, посредством которой мы некогда создавали файлы. Данная с именем файла в качестве аргумента, она выведет его содержимое на экран. Можно использовать и конструкцию перенаправления:

```
$ cat < filename
```

Несмотря на то, что в принципе это разные вещи, результат будет тот же — вывод содержимого файла на экран.

Недостаток команды `cat` как средства просмотра — невозможность перемещения по телу файла: выведя содержимое, она завершает свою работу. Конечно, "пролистывание" выведенного возможно, но — средствами системной консоли, а не самой команды.

Поэтому обычно для просмотра содержимого файлов используются специальные программы постраничного просмотра — т. н. pager'ы, очередной пример того, что передача этого термина исконно русским словом "пейджер" (а мне попадалось и такое) может создать совершенно превратное представление о сути дела.

В UNIX-системах имеются две основные программы pager'а — `more` и `less`. Первая из них допускает только однонаправленный (вперед) просмотр и имеет слабые интерактивные возможности. Почему ныне и представляет лишь исторический интерес, так что о ней я говорить не буду. Тем более, что в современных свободных POSIX-системах она как таковая отсутствует: файл с именем `/usr/bin/more`, который можно обнаружить во FreeBSD и некоторых дистрибутивах Linux, на самом деле представляет собой жесткую или символическую ссылку на ту же самую программу, что и команда `less`. Хотя эта программа проявляет несколько различные свойства, в зависимости от того, какой из указанных команд она вызвана, функции ее от этого не меняются. Так что дальше я буду говорить только о команде `less`.

Самый простой способ вызова команды

```
$ less filename
```

после чего на экран выводится содержимое файла, указанного в качестве аргумента, по которому можно перемещаться в обоих направлениях, как построчно, так и постранично. В нижней строке экрана можно видеть сим-

вол двоеточия — приглашения для ввода команд различного назначения. В частности, нажатие клавиши `h` выводит справку по использованию `less`, а клавиши `q` — выход из программы просмотра (или из просмотра справочной системы, если она была перед этим вызвана). Если команда была вызвана как `more` (это достигается еще и специальной опцией — `less -m`), вместо символа двоеточия в нижней строке будет выведено имя файла с указанием процента просмотра:

```
command.txt 3%
```

что, однако, не воспрещает и здесь давать ее встроенные команды — вводом символа двоеточия (`:`) и закрепленной за командой литеры (или их сочетания).

Большинство встроенных команд `less` предназначено для навигации по телу файла. Осуществляется она несколькими способами:

- с помощью стандартных клавиш управления курсором;
- с помощью предопределенных клавишных комбинаций, сходных с управляющими клавиатурными последовательностями командных оболочек и таких текстовых редакторов, как emacs и joe;
- с помощью фиксированных символьных клавиш, иногда подобных таковым командного режима редактора vi: `z` и `w` (вперед и назад на один экран), `e` и `y` (вперед и назад на одну строку, можно использовать также привычные по vi клавиши `j` и `k` соответственно), `d` и `u` (вперед и назад на пол-экрана).

Последний способ интересен тем, что допускает численные аргументы перед символьной командой: так, нажатие `3e` приведет к перемещению на три строки вперед, а `2w` — на два экрана назад.

Помимо "плавной", так сказать, навигации, можно перемещаться по файлу и скачками (jumping): нажатие клавиши с символом `g` (или <Esc>-последовательности) позволяет переместиться к первой строке файла, клавиши `G` (регистр важен!) — к последней его строке, клавиши `p` — к началу файла.

Кроме навигации, имеется и возможность двустороннего поиска — в направлении как конца, так и начала файла. Для поиска вперед требуется ввести символ прямого слэша (`/`) и за ним — искомое сочетание символов. Поиск в обратном направлении предваряется символом вопроса (`?`). В обоих случаях в шаблоне поиска можно использовать стандартные регулярные выражения `*`, `?`, `[список_символов]` или `[диапазон_символов]`. Нажатие клавиши `n` (в нижнем регистре) приводит к повторному поиску в заданном ранее направлении, клавиши `N` (в верхнем регистре) — к поиску в направлении противоположном.

Управляющие комбинации команды `less` могут быть переопределены с помощью команды `lesskey`. Формат ее

```
$ lesskey -o output input
```

В качестве входных данных выступает простой текстовый файл (по умолчанию — `~/.lesskey`, однако его следует создать самостоятельно), описывающий клавишные последовательности в следующем, например, виде:

```
#command
\r        forw-line
\n        forw-line
...
k         back-line
...
```

Выходные данные — создаваемый из текстового двоичный файл, который собственно и используется командой `less`. Стандартное для него имя — `~/.less`.

Команда `less` допускает одновременный просмотр нескольких файлов. Для этого ее следует вызвать в форме

```
$ less file1 file2 ... file#
```

после чего между открытыми файлами можно переключаться посредством `:n` (к следующему файлу), `:p` (к предыдущему файлу), `:x` (к первому файлу). Путем нажатия `:d` текущий файл исключается из списка просмотра. Символ двоеточия во всех этих случаях вводится с клавиатуры в дополнение к приглашению на ввод команд.

Команда `less` имеет великое множество опций — описание их на странице экранной документации занимает более дюжины страниц, поэтому задерживаться на них я не буду. Следует заметить только, что опции эти могут использоваться не только в командной строке при запуске `less`, но и интерактивно — после символа дефиса в приглашении ввода. Так, указав там `-m`, можно включить т. н. промежуточный формат приглашения (с отображением процентов просмотренного объема файла), а с помощью `-M` — длинный (`more`-подобный) формат, при котором в приглашении дополнительно указываются имя файла, его положение в списке загруженных файлов, просматриваемые ныне строки:

```
command.html (file 2 of 10) lines 1-29/1364 2%
```

Значение команд постраничного просмотра файлов еще и в том, что именно с их помощью осуществляется доступ к экранной документации (man-страницам). Команда

```
$ man cmd_name
```

как было описано в предыдущей интермедии, на самом деле вызывает определенный по умолчанию pager для просмотра соответствующего файла /usr/share/man/man#/cmd_name.gz. Какой именно — определяется переменной `PAGER` в пользовательских настройках.

Кроме команд постраничного просмотра, существуют команды для просмотра фрагментарного. Это — `head` и `tail`, выводящие на экран некоторую фиксированную порцию файла, указанного в качестве их аргумента, с начала или с конца соответственно. По умолчанию эта порция для обеих команд составляет десять строк (включая пустые). Однако ее можно переопределить произвольным образом, указав опции `-n [число_линий]` или `-c [количество_байт]`. Например, команда

```
$ head -n 3 filename
```

выведет три первые строки файла filename, а команда

```
$ tail -c 100 filename
```

его последние 100 байтов. При определении выводимого фрагмента в строках название опции (`n`) может быть опущено — достаточно числа после знака дефиса.

Существуют и средства просмотра компрессированных файлов. Для файлов, сжатых программой `gzip`, можно использовать команды `zcat` и `zmore`, для спрессованных командой `bzip2` — команду `bzcat`. Использование их ничем не отличается от аналогов для несжатых файлов — в сущности, именно они и вызываются для обеспечения просмотра. В случае команды `zmore`, как нетрудно догадаться, на самом деле используется команда `less` (сама по себе она аналога для компрессированных файлов не имеет).

Сравнение, объединение и деление файлов

Следующая важная группа операций над контентом файлов — сравнение файлов по содержанию и различные формы объединения файлов и их фрагментов. Начнем со сравнения. Простейшая команда для этого — `cmp` в форме

```
$ cmp file1 fil2
```

производит построчное сравнение файлов, указанных как первый и второй аргументы (а более их и не предусмотрено, все указанное после второго аргумента игнорируется). В случае идентичности сравниваемых файлов не происходит ничего, кроме возврата приглашения командой строки. Если же между файлами имеются различия, выводится номер первого различающегося символа и номер строки, в которой он обнаружен:

```
file1 file2 differ: char 27, line 4
```

Это означает, что различия между файлами начинаются с 27-го от начала файла символа (включая пробелы, символы конца строк и т. д.), который имеет место быть в строке 4. С помощью опций `-l` и `-z` можно заставить команду `cmp` вывести номера всех различающихся символов в десятичном или шестнадцатеричном формате соответственно.

Более информативный вывод обеспечивает команда `diff`. Она также осуществляет построчное сравнение двух файлов, но выводит список строк, в которых обнаружены отличия. Например, для двух файлов вида

```
$ less file1
line 1
line 2
line 3
line 4
line 5
```

и

```
$less file2
line 1
line 2
line 3
line 3a
line 4
line 5
```

это будет выглядеть следующим образом:

```
$ diff file1 file2
3a4
> line 3a
```

Если различия будут выявлены более чем в одной строке, для каждого расхождения будет выведен аналогичный блок. Смысл его — в том, какие строки первого файла должны быть преобразованы, и как именно, для того, чтобы файлы стали идентичными. Первая линия блока вывода содержит номер строки первого файла, подлежащей преобразованию, номер соответствующей строки второго файла и обозначенное буквенным символом преобразование, во второй линии приведена собственно строка — предмет преобразования. Символы преобразования — следующие:

- `a` (от append) указывает на строку, отсутствующую в первом файле, но присутствующую во втором;

- `c` (от change) фиксирует строки с одинаковым номером, но разным содержанием;
- `d` (от delete) определяет строки, уникальные для первого файла.

То есть в данном примере для преобразования `file1` в `file2` в него после строки 3 должна быть вставлена строка 4 из второго файла, что символизирует вторая линия блока — `> line 3a`, где `>` означает строку из первого сравниваемого файла. Если же аргументы команды `diff` дать в обратном порядке, вывод ее будет выглядеть следующим образом:

```
$ diff file2 file1
4d3
< line 3a
```

показывающим, что для достижения идентичности из `file2` должна быть удалена четвертая строка (`< line 3a`, где `<` означает строку из второго файла). Если же произвести сравнение `file1` с `file3`, имеющим вид

```
$ less file3
line 1
line 2
line 3a
line 4
line 5
```

то вывод команды

```
$ diff file1 file3
3c3
< line 3
---
> line 3a
```

будет означать необходимость замены третьей строки из `file1` (символ `<`) на третью строку из `file3` (символ `>`).

Такая форма вывода команды `diff` называется стандартной. С помощью опции `-c` можно задать т. н. контекстную форму вывода, при которой на экран направляются не только различающиеся строки, но и строки, их окружающие (то есть контекст, в котором они заключены):

```
diff -c file1 file2
*** file1       Sun May 12 11:44:44 2002
--- file2       Mon May 13 15:17:27 2002
***************
*** 1,5 ****
```

```
--- 1,6 ----
  line 1
  line 2
  line 3
+ line 3a
  line 4
  line 5
```

Количество строк контента задается опцией `-C`:

```
diff -C 1 file1 file2                                    ttyv1
*** file1       Sun May 12 11:44:44 2002
--- file2       Mon May 13 15:17:27 2002
***************
*** 3,4 ****
--- 3,5 ----
  line 3
+ line 3a
  line 4
```

В этом примере значение опции `-C` (единица) предписывает вывод по одной строке контекстного окружения вокруг различающейся строки. Сами же различающиеся строки помечаются следующим образом: знаком `-` (минус, или дефис) — строки, подлежащие удалению из первого файла, знаком `+` (как в примере) — строки, которые должны быть добавлены, знаком `!` — просто различающиеся строки.

Кроме контекстного формата, используется еще и вывод в унифицированном формате, что предписывается опцией `-U [значение]`, в качестве значения указывается число строк. В нем для обозначения изменяемых строк используются только символы `+` и `-`, а сам вывод чуть короче, чем при использовании контекстного формата.

С помощью многочисленных опций команды `diff` сравнение файлов может быть детализовано и конкретизировано. Так, опция `-b` предписывает игнорировать "пустые" символы пробелов и табуляции в конце строк, а опция `-w` — вообще "лишние" пробелы (и те, и другие обычно имеют случайное происхождение). При указании опции `-B` игнорируются пустые строки, то есть не содержащие никаких иных символов, кроме перевода каретки; строки с символами табуляции или пробела как пустые не рассматриваются, для их игнорирования требуется опция `-w`. Благодаря опции `-i` при сравнении не принимается во внимание различие регистров символов, а опция `-I`

`regexp` определяет регулярные выражения, строки с которыми также игнорируются при сравнении.

В качестве аргументов команды `diff` (одного или обоих) могут выступать также каталоги. Если каталогом является только один из аргументов, для сравнения в нем отыскивается файл, одноименный второму аргументу. Если же оба аргумента суть каталоги, в них происходит сравнение всех одноименных файлов в алфавитном порядке (вернее, в порядке ASCII-кода первого символа имени, разумеется). Благодаря опции `-r` сравнение файлов может осуществляться и во вложенных подкаталогах.

Вывод команды `diff` может быть перенаправлен в файл. Такие файлы различия именуются diff-файлами или, применительно к исходным текстам программ, патчами (patches), о которых будет сказано несколько позже. Именно с помощью таких патчей обычно распространяются изменения к программам (дополнения, исправления ошибок и т. д.).

В принципе, команда `diff` и придумана была именно для сравнения файлов исходников, над которыми ведут работу несколько (в пределе — неограниченное количество, как в случае с Linux) человек. Однако невозбранно и ее использование в мирных целях — то есть для сравнения просто повествовательных текстов. Единственное, что следует понимать при этом абсолютно ясно — то, что `diff` выполняет именно построчное сравнение. То есть: сравнение последовательностей символов, ограниченных символами конца строки с обеих сторон. И, соответственно, непрерывная абзацная строка в стиле emacs и vi — совсем не то же самое, что строка, образуемая в редакторе joe на границе экрана. Впрочем, это — вопрос, к которому еще не раз придется возвращаться.

Как уже было отмечено, команда `diff` осуществляет сравнение двух файлов (или — попарное сравнение файлов из двух каталогов). Однако поскольку Бог, как известно, любит троицу, есть и команда `diff3`, позволяющая сравнить именно три файла, указываемые в качестве ее аргументов. По действию она не сильно отличается от двоичного аналога. И потому изучение ее особенностей предлагается в качестве самостоятельного упражнения приверженцам троичной идеологии.

Существуют и средства для сравнения сжатых файлов. Это — `zcmp` и `zdiff`. Подобно командам просмотра, ими просто вызываются соответствующие команды `cmp` и `diff`. И потому использование их не имеет никаких особенностей.

От вопроса сравнения файлов плавно перейдем к рассмотрению способов их объединения. Для этого существует немало команд, из которых по справедливости первой должна идти команда `cat`, поскольку именно сие есть ее титульная функция (`cat` — от concatenation, сиречь объединения). Ранее уже

упоминалось, что она способна добавлять информацию со стандартного ввода в конец существующего файла. Однако дело этим не ограничивается. В форме

```
$cat file1 file2 ... file# > file_all
```

она создает новый файл, включающий в себя содержимое всех файлов-аргументов (и именно в том порядке, в каком они приведены в командной строке). Операция, казалось бы, нехитрая — однако представьте, сколько действий потребовалось бы в текстовом процессоре (например, в Word) для того, чтобы создать синтетический вариант из полутора десятков фрагментов, раскиданных по разным каталогам?

А вот команда `patch` выступает в качестве диалектической пары для команды `diff`, именно она вносит в файл те изменения, которые документируются последней. Выглядит эта команда примерно так:

```
$patch file1 diff_file
```

в ответ на что последует нечто вроде следующего вывода:

```
Hmm...  Looks like a normal diff to me...
Patching file file1 using Plan A...
Hunk #1 succeeded at 4.
done
```

В результате исходная версия `file1` будет сохранена под именем `file1.orig`, а сам он преобразован в соответствии с описанием diff-файла. Возможна и форма

```
patch < diff_file
```

В этом случае команда `patch` попытается сама определить имя файла-оригинала, и, если это ей не удастся, даст запрос на его ввод. Обращаю внимание на символ перенаправления ввода, поскольку если его опустить, имя diff-файла будет воспринято как первый аргумент команды (то есть имя файла-оригинала).

В качестве второго аргумента команды `patch` могут использоваться diff-файлы не только в стандартном, но и в контекстном или унифицированном формате. Это следует указать посредством опции `-c` или `-u` соответственно.

Сочетание команд `diff` и `patch` очень широко используется при внесении изменений в исходные тексты программы. Так, если мы обратимся к коллекции портов FreeBSD, то в большинстве каталогов портированных программ можно обнаружить подкаталог files (например, /usr/ports/editors/joe-devel/files), содержимое которого как раз и есть те патчи, которые должны накладываться на исходные тексты программы для ее корректной сборки, установки и функционирования во FreeBSD. Сам же исходный текст при

этом в неизменном, то есть распространяемом разработчиком, виде получается с его сайта (или любого другого сервера, в том числе и каталога distfiles FTP-сервера проекта FreeBSD.

Не менее часто, чем в слиянии, возникает и необходимость в разделении файлов на части. Цели этой служит команда `split`. Формат ее:

```
$ split [options] filename [prefix]
```

В результате исходный файл будет разбит на несколько отдельных файлов вида `prefixaa`, `prefixab` и т. д. Значение аргумента `prefix` по умолчанию — `x` (то есть без его указания итоговые файлы получат имена `xaa`, `xab` и т. д.).

Опции команды `split` задают размер выходных файлов — в байтах (опция `-b`) или количестве строк (опция `-l`). Первой опцией в качестве единицы, кроме байтов, могут быть заданы также килобайты или мегабайты — добавлением после численного значения обозначения `k` или `m` соответственно.

Команда `split` может использоваться для разбиения файлового архива на фрагменты, соответствующие размеру резервных носителей. Так, в форме

```
$ split -b 1474560 arch_name
```

она обеспечит разбиение архива на части, каждая из которых может быть записана на стандартную трехдюймовую дискету. А посредством

```
$ split -b 650m arch_name
```

архив можно подготовить к записи на носители CD-R/RW. Напомню, что именно командой `split` создаются файлы дистрибутива FreeBSD. Легко догадаться, что обратное слияние таких фрагментированных файлов можно выполнить командой `cat`.

Однако этим возможности BSD-реализации команды `split` не исчерпываются. С помощью опции `-p` файл может быть разделен на фрагменты, ограниченные строками, содержащими текст, приведенный в качестве значения шаблона. Так, командой

```
$ split -p Глава filename chapter-
```

текст произвольной длины будет легко разбит на файлы, соответствующие отдельным главам, которым будут присвоены имена `chapter-aa` и далее. В качестве значения опции `-p` могут использоваться теги HTML или символы разметки TeX, а также регулярные выражения.

Linux-реализация команды `split` таким свойством не обладает. Однако взамен этому в Linux есть команда `csplit`, именно для разделения файла по шаблону и предназначенная.

Поиск в файлах

В одном из предыдущих разделов говорилось о поиске файлов посредством команды `find`. Ныне же речь пойдет о поиске внутри файлового контента, то есть поиске текстовых фрагментов. Для этого в POSIX-системах используется семейство утилит grep — собственно grep, egrep и fgrep, несколько различающихся функционально. Впрочем, в большинстве систем все это суть разные имена (жесткие ссылки) одной и той же программы, именуемой GNU-реализацией grep, включающей ряд функций, свойственных ее расширенному аналогу, egrep. Соответственно, поиск текстовых фрагментов в файлах может быть вызван любой из этих команд, хотя в каждом случае функциональность их будет несколько различаться.

Однако начнем по порядку. Самой простой формой команды `grep` является следующая:

```
$ grep pattern files
```

где `pattern` — искомая последовательность символов, а `files` — файлы, среди которых должен производиться поиск (или просто одиночный файл). В указании имен файлов допустимы обычные маски, например, командой

```
$ grep line ./*
```

будут найдены строки вида `line` во всех файлах текущего каталога. Шаблон для поиска не обязан быть односложным. Правда, если в нем используются последовательности символов, разделенные пробелами, последние должны тем или иным способом экранироваться, иначе в качестве шаблона будет воспринято только первое слово. Например, каждый пробел может предваряться символом обратного слэша (`\`), или просто все искомое выражение заключается в одинарные или двойные кавычки.

Шаблоны могут включать в себя регулярные выражения. Причем список таковых для команды `grep` существенно шире, чем для команд манипулирования файлами. Так, кроме маски любой последовательности символов (`*`), любого одиночного символа (?), списка и диапазона символов (`[a...z]` и `[a-z]`), могут встречаться:

- `.` (точка) — маска любого одиночного (но, в отличие от маски `?`, обязательно присутствующего) символа; то есть при задании шаблона `lin.` будут найдены строки, содержашие `line` или `lins`, но не `lin`;
- `^` и `$` — маски начала и конца строки, соответственно: по шаблону `^line`, будут найдены строки, начинающиеся с соответствующего слова, по шаблону же `line$` — им заканчивающиеся;

- маски повторения предыдущего шаблона, `\{n\}` — ровно `n` раз, `\{n,\}` — не менее `n` раз, `\{,m\}` — не более `m` раз, `\{n,m\}` — не менее `n` раз и не более `m` раз.

Маски повторения относятся к так называемым расширенным регулярным выражениям. Для их использования команда `grep` должна быть дана с опцией `-e` или в форме `egrep` — последняя часто определяется в общесистемном или пользовательском профильном файле как псевдоним команды `grep`:

```
alias grep='egrep -s'
```

или

```
alias grep egrep
```

в оболочках семейств `shell` и `csh` соответственно.

При этом становятся доступными и другие возможности поиска — например, нескольких текстовых фрагментов (соединенных логическим оператором ИЛИ) одновременно. Делается это двояко. Первый способ — просто перечисление искомых фрагментов, каждый из которых предваряется опцией `-e` (и при необходимости экранируется кавычками):

```
$ grep -e pattern1 -e pattern2 files
```

При втором способе оператор между искомыми шаблонами задается в явном виде:

```
$ egrep 'pattern1|pattern2' files
```

Таким способом очень легко, например, составить оглавление для длинного текста (при наличии некоторой системы рубрикации в нем, разумеется). Для этого достаточно дать команду вроде следующей:

```
$ egrep 'Часть|Глава|Раздел|Параграф' filename
```

Для текста, включающего HTML- или TeX-разметку, роль рубрик могут выполнять соответствующие ее элементы, например:

```
$ egrep ' <h1>|<h2>|<h3>|<h4>' filename
```

Вывод команды `grep` может быть перенаправлен в файл, а при необходимости и предварительно отсортирован с помощью соответствующих командных конструкций перенаправления и конвейеризации.

Разумеется, тем же способом можно создать общее оглавление для серии фрагментов, записанных в виде самостоятельных файлов, — для этого достаточно только перечислить их имена в качестве аргументов команды. Так, например, если есть необходимость составления детальной карты сайта, включающей ссылки на подрубрики внутри отдельных HTML-документов, следует применить конструкцию типа:

```
$ egrep '<h1>|<h2>|<h3>|<h4>' path/*.html > sitemap.html
```

Еще одно замечательное свойство команды grep (и egrep) — возможность получения шаблона не со стандартного ввода (то есть не путем набора его с клавиатуры), а из файла. Так, если для приведенного выше случая создать простой текстовый файл shablon, содержащий строку

```
Часть|Глава|Раздел|Параграф
```

выполнять операцию по сборке оглавления впредь (и в любом тексте, хоть частично совпадающем по структуре с рассмотренным) можно будет таким образом:

```
$ egrep -f shablon filename
```

Опция -f и указывает команде, что список параметров должен извлекаться из файла, указанного в качестве значения опции.

Список опций команды grep не исчерпывается указанными. Так, опция -i предписывает игнорировать различие регистров внутри искомого выражения, опция -h — подавляет вывод имен файлов (выводится только содержание найденных строк), тогда как опция -l, напротив, выводит только имена файлов, содержащих найденный шаблон, но не текстовый фрагмент, опция -n выводит номера найденных строк в соответствующих файлах. Весьма важной представляется опция -v, обеспечивающая инверсию поиска: при указании ее выводятся строки, не содержащие шаблона поиска.

Команда grep имеет и аналог для поиска в сжатых файлах — команду zgrep. Использование ее в целом аналогично, и останавливаться на нем я не буду.

Sed: средство потокового редактирования

Весьма часто при обработке текстов встает такая задача: заменить одно слово (или последовательность слов) на другое сразу во многих файлах. Как она решается "подоконными" средствами? Обычно открытием всех подлежащих изменению документов в word-процессоре и применением функции поиска/замены последовательно в каждом из них.

Таким же способом можно воспользоваться и в POSIX-мире. Это просто, но уж больно скучно. Тем паче, что здесь есть очень эффективная альтернатива — средства потокового (неинтерактивного) редактирования, примером которых является sed.

Потоковое редактирование не требует загрузки документа в память (то есть открытия), как в обычных текстовых редакторах и word-процессорах. Нет, при нем подлежащий изменению файл (или группа файлов) обрабатывается построчно с помощью соответствующих команд, задаваемых как опции единой командной директивы. В наши дни это выглядит анахронизмом, однако

в ряде случаев оказывается чрезвычайно эффективным. Каких? Ответ легко дать на нескольких конкретных примерах.

Так, при установке некоторых дистрибутивов Linux из числа Source Based вполне возможна ситуация, когда в какой-то момент времени в распоряжении пользователя не окажется никакого обычного текстового редактора. А необходимость внесения мелких изменений в конфигурационные файлы (например, в файл /etc/fstab) возникнет. Так что делать, бежать срочно устанавливать свой любимый vim или emacs? И то, и другое — дело отнюдь не пяти минут. И тут самое время вспомнить про sed, который обязательно будет присутствовать в системе (поскольку он используется во многих установочных сценариях базовых пакетов).

Другой случай — во многих десятках, а то и сотнях, файлов требуется изменить одну-единственную строку, причем — одинаковым образом (например, заменить копирайт Васи Пупкина на Петю Лавочкина). Неужто для этой цели нужно вызывать мощный текстовый редактор, грузить в него кучу документов, перемещаться тем или иным способом к нужному фрагменту, вносить требуемое изменение? Отнюдь, ибо sed поможет и здесь, позволив выполнить изменение любого количества файлов в один в пакетном режиме.

Во всем блеске sed показывает себя при редактировании очень больших файлов (одно пролистывание которых требует немалого времени). А также — при редактировании сложных символьных последовательностей в нескольких файлах. Однажды, после очередной реконструкции моего сайта, передо мной встала задача тотальной модификации всех внутренних ссылок. Долго я с ужасом размышлял, как буду делать это в текстовом редакторе, и сколько ошибок при этом насажаю. Пока, раскинув мозгами, не нашел, как сделать это с помощью sed — быстро и, главное, безошибочно.

В самом общем виде sed требует двух аргументов — указания встроенной его команды и имени файла, к которому она должны быть применена. Впрочем, в качестве аргумента можно задать только простую команду, мало-мальски сложное действие (а команды поиска/замены принадлежат к числу сложных) необходимо определить через значения опции `-e`, заключенные в кавычки (одинарные или двойные — по ситуации). Что же касается имен файлов — то их в качестве аргументов можно указать сколько угодно, в том числе и с помощью масок типа `*`, `*.txt` и т. д. Правда, sed не обрабатывает содержимое вложенных подкаталогов, но это — дело поправимое (как — скоро увидим). Так что поиск и замена слова или их последовательности выполняются такой конструкцией:

```
$ sed -e 's/Вася Пупкин/Петя Лавочкин/' *
```

Здесь `s` — это команда поиска, `Вася Пупкин` — искомый текст, а `Петя Лавочкин` — текст для замены. В приведенной форме команда выполнит поиск

и замену только первого вхождения искомого текста. Чтобы заменить текст по всему файлу, после последнего слэша (он обязателен в любом случае, без него sed не распознает конца заменяющего фрагмента) нужно указать флаг `g` (от global). Важно помнить, что если оставить заменяющее поле пустым, искомый текст будет просто удален.

По умолчанию sed выводит результаты своей работы на стандартный вывод, не внося изменений в файлы аргументы. Так где же здесь редактирование? Оно обеспечивается другой опцией `-i`, указание которой внесет изменения непосредственно в обрабатываемый файл. В результате команда для замены, например, всех вхождений html на shtml во всех файлах текущего каталога будет выглядеть так:

```
$ sed -i -e 's/html/shtml' *
```

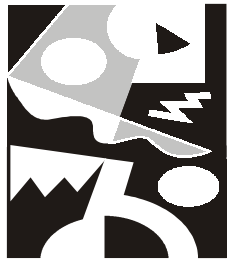
А что делать, если таким же образом нужно обработать файлы во всех вложенных подкаталогах? Придется вспомнить об универсальной команде `find`, описанной ранее. В форме

```
$ find . -name * -exec sed -i -e 's/html/shtml' * {} \
```

она с успехом справится с этой задачей.

Я привел лишь элементарные примеры использования sed. На самом деле возможности его много шире — представление о их применении в реальных ситуациях можно получить из специальной статьи Сергея Майкова (**http://posix.ru/shell/bash_and_sed/**).

Глава 13



Общесистемное конфигурирование

- Введение.
- О загрузке и загрузчиках.
- Особенности загрузчиков Lilo и GRUB.
- Задачи инициализации.
- Стили инициализации.

В *главе 4* говорилось о том, что понимание принципов устройства системы незаменимо при ее конфигурировании и, главное, в отличие от знания конкретных рецептов, имеет универсальный, не зависящий от дистрибутива и даже системы, характер. Настало время продемонстрировать это на практике. Тем более, что настройка общесистемных параметров загрузки и инициализации — необходимый этап в индивидуализации любой ОС POSIX-семейства. И для понимания этих принципов необходимо представлять, как эти самые операции, загрузка и инициализация, происходят.

Введение

Старт системы распадается на два этапа, лишь опосредованно связанных между собой: собственно загрузку ее и инициализацию.

Под загрузкой операционной системы понимается запуск на исполнение специальной программы, которая называется образом ядра системы (или просто ядром), а также, возможно, сопряженных с ним модулей. Как уже говорилось, образ ядра — почти обычный бинарный исполняемый файл. И специфика его запуска только в том, что, если любые другие программы запускаются под управлением какой-либо ОС, считываясь с файловой системы, которую эта ОС воспринимает в качестве родной, то ядро обязано запуститься как бы само собой и с носителя, о котором система ничего не

знает (поскольку сама она еще не загружена). Не случайно в англоязычной литературе для процесса загрузки общепринятая метафора звучит как bootstrapping, что столь же аллегорически можно перевести как "поднятие себя за шнурки своих ботинок" (остается только вспомнить барона Мюнхгаузена, вытащившего себя из болота за волосы). И хотя в реальной жизни такое не каждому удается, в мире POSIX-систем эта процедура осуществляется регулярно и, — как правило, успешно.

Образ ядра содержит все необходимое для чистого bootstrapping: загрузочный сектор, первичный загрузчик и собственно исполняемый код ядра. Однако загрузить его с нормальной файловой системы невозможно, ведь о существовании таковых ядро узнает только после того, как будет загружено. Следовательно, образ ядра должен лежать на raw-устройстве (дисковом разделе, дискете и т. д.), файловой системы лишенном. И потому такой способ применяется почти исключительно при использовании для старта Linux или BSD с загрузочной дискеты, например, для аварийно-спасательных работ. В прочих же случаях для загрузки ядра системы применяются специальные программы, именуемые, как и следовало бы ожидать, загрузчиками. Они загружают ядро системы, отвечают за определение оборудования, подгрузку соответствующих ему модулей и запуск первого процесса уже работающей ОС — процесса `init`.

После этого в действие вступает система инициализации. Ее роль обеспечить, посредством соответствующих стартовых скриптов (они же — сценарии инициализации), запуск основных системных процессов: сервисов или демонов. А также вызвать команды для получения терминала (процессы `getty`) и авторизации пользователей (`login`). Окончание старта и знаменуется приглашением к авторизации, все остальное относится уже к сфере влияния пользовательского окружения (userland). Зрительно этапы загрузки и инициализации отличаются тем или иным визуальным представлением выводимых сообщений (если вывод сообщений о ходе стартовых процессов, конечно, не отключен напрочь — в некоторых дистрибутивах Linux встречается и такое). Во FreeBSD и DragonFlyBSD, например, сообщения о ходе загрузки выводятся символами радикально белого цвета, сменяемыми на этапе инициализации обычным приглушенно-белым. В Linux картина внешне прямо противоположна: приглушенно-белая гамма в ходе старта и ярко-белая — при инициализации. А в NetBSD и OpenBSD сообщения о ходе загрузки даются на синем фоне...

Загрузка и инициализация — это первое, что в любой ОС видит пользователь. Правда, пользователю POSIX-совместимой системы такое удовольствие выпадает много реже, чем "подоконнику". Нормальный режим эксплуатации домашней UNIX-машины — это ее включение рано утром и выключение поздно вечером (представления о "рано" и "поздно" у всех свои). А служебная UNIX-машина по-хорошему выключаться вообще не должна, вплоть до полной физической амортизации. Ну, а необходимость в

рестарте системы по ходу работы в Linux или BSD возникает чрезвычайно редко. Собственно, только после пересборки и инсталляции нового ядра (или переносе корневого раздела, но это уже вообще исключительный случай), во всех прочих случаях реконфигурирования системы можно обойтись и без этого.

Так что, казалось бы, что за дело пользователю до того, как протекает старт системы и сколь долго он длится? Тем не менее, некоторые действия по настройке обоих этапов этого процесса бывают необходимыми. Ибо при старте системы не только выводится некоторая заставка и, возможно, меню с вариантами, но и подгружаются модули ядра, соответствующие наличному оборудованию, монтируются файловые системы, запускаются сценарии инициализации, открываются виртуальные терминалы, и т. д., и т. п. Конечно, безусловно обязательной из всех этих действий является только собственно загрузка ядра, прочие могут быть выполнены и впоследствии. Однако не лучше ли сразу по окончании процедуры старта получить полностью готовую к употреблению систему, нежели потом доводить ее вручную до этого состояния?

Так что давайте проследим основные стадии ее и посмотрим, где и как (а главное, зачем) в нее можно вмешаться.

О загрузке и загрузчиках

Начать изучение старта системы резонно с первого ее этапа, а именно, собственно загрузки. Как уже было сказано, управление этим этапом осуществляет специальная программа, которая по-русски так и называется — загрузчик, хотя в английском для нее употребляется два термина: loader и boot manager (что, как мы увидим со временем, немного разные вещи, но сейчас это не принципиально).

На самом деле любой загрузчик включает в себя две или даже три относительно независимые части, даже если он распространяется в виде единого пакета, как Lilo или GRUB. Чтобы убедиться в этом, представим себе, как происходит запуск машины на "железном", так сказать, уровне (имеется в виду intel-совместимой персоналки, на иных архитектурах все обстоит несколько иначе).

После включения питания запускается программа, прошитая в ПЗУ компьютера (BIOS). Она выполняет проверку железа, после чего отыскивает носитель, установленный в BIOS Setup в качестве первого загрузочного устройства (для определенности, винчестер), на нем первый физический блок, содержащий так называемую главную загрузочную запись (MBR, Master Boot Record).

Содержимое MBR — это, во-первых, таблица дисковых разделов, тех самых, о которых шла речь в *главе 9*. А во-вторых, некий код, принимающий на себя управление от BIOS по окончании его работы. В стандартном MBR (том, что записан на новом винчестере или восстанавливается после DOS-команды `FDISK /mbr`) этот код только и может, что отыскать первый физический раздел диска (primary partition) и передать управление на его загрузочный сектор. Этого вполне достаточно для загрузки систем вроде DOS или Windows 9*x*/ME с первого (или единственного) раздела. Но явно мало в любом другом случае, например, если на диске установлено несколько ОС, которые, естественно, не могут уместиться в одном разделе.

Поэтому в состав любого загрузчика должна входить программа, записываемая в MBR. Поскольку объем последнего всего 512 Кбайт (размер физического дискового блока), из которых часть уже занята под таблицу разделов, особо богатых функций в эту программу не вместить. Обычно она способна на то, чтобы опознать все задействованные (used) первичные разделы, вывести их список и позволить пользователю выбрать раздел для загрузки, после чего передать управление на загрузочный сектор выбранного раздела.

Подобно MBR, загрузочный сектор раздела (Boot Record, уже не Master!) содержит информацию о его разметке (Disk Label), зависящую от используемой в данной ОС схемы, и управляющий код, принимающий эстафету от программы, записанной в MBR. И этот код — вторая составная часть загрузчика. Правда, и ее возможности также не могут быть богатыми, ведь размер загрузочного сектора раздела составляет те же 512 Кбайт. И потому на нее возлагается одна-единственная функция: передать управление программе, лежащей за пределами загрузочного сектора. Которая, собственно, и должна опознать корневой раздел ОС и несомую им файловую систему, после чего, прямо или опосредованно, загрузить ее ядро.

Легко догадаться, что первые две составляющие загрузчика, в сущности, не имеют отношения ни к какой системе и не являются частями файловой системы вообще. А вот с третьей возможны варианты. Она может входить в файловую иерархию загружаемой ОС, как Lilo, или представлять собой нечто вроде самостоятельной мини-ОС, как GRUB (не случайно его настоятельно рекомендуют устанавливать в собственный дисковый раздел, не монтируемый по умолчанию к корню любой из загружаемых им ОС).

Стабиальность загрузки системы отчетливо выражена в BSD Loader — программе, используемой для загрузки систем этого семейства, но способной столь же успешно использоваться в качестве мультисистемного загрузчика. Однако о ней здесь речи не будет.

Особенности загрузчиков Lilo и GRUB

К тому же, BSD Loader — далеко не самый распространенный из мультисистемных загрузчиков в свободном POSIX-мире. Многие пользователи Linux применяют в этом качестве Lilo (Linux Loader). Тот же, кто по долгу или прихоти использует множество (более двух) операционных систем, да еще и меняет их, как перчатки, обычно отдает предпочтение загрузчику GRUB (GRand Unified Bootloader).

Загрузчик Lilo и его настройка — предмет подробного описания в любой толстой книге про Linux. В общем случае он записывается в MBR загрузочного диска, откуда при старте машины опознает дисковые разделы, как первичные, так и расширенные. И если раздел несет на себе метку Linux native, загрузит с него образ ядра Linux (изначально Lilo для этого и предназначался, его функции мультисистемного загрузчика вторичны). Если же тип раздела иной, Lilo в состоянии по цепочке передать управление на его загрузочный сектор, и дальнейшее будет определяться тем кодом, что записан в последнем.

Тип файловой системы для Lilo, по большому счету, безразличен: он способен загрузить ядро Linux с любой из ее многочисленных нативных файловых систем. Однако до недавнего времени он не мог выполнить эту операцию с файловой системы ReiserFS в случае, если в ней использовался так называемый tailing (*см. главу 9*). Хотя ныне это будто бы изжито, при размещении ReiserFS на корневом разделе обычно все равно рекомендуется выделять специальный раздел /boot с файловой системой Ext2fs под ядро и служебные его файлы, для страховки. Ведь, хотя использование тайлинга можно отключить, он автоматически восстанавливается при реинсталляции ядра.

Для разделов "чуждых" ОС тип их файловой системы для Lilo безразличен тем более, ведь, как уже было сказано, он просто передает управление на их загрузочные сектора. И поэтому, если озаботиться тем, чтобы в загрузочном секторе BSD-слайса был записан код `boot1` из BSD Loader, то Lilo благополучно справится с загрузкой любой ОС этого семейства. А возможность записи соответствующего кода при установке BSD-систем предоставляется. Точно таким же способом — по цепочке — Lilo может обеспечить загрузку DOS, Windows 3.*x*/9*x*/ME и Windows NT/2000/XP.

Каждая ОС, предусмотренная для загрузки через Lilo, представляет собой один из вариантов выбора в меню этой программы: это могут быть разные дистрибутивы Linux, лежащие на собственных разделах, и разные версии ядра одной и той же Linux-системы, и просто разные сборки ядра одной и той же версии. Поэтому Lilo очень часто применяют для тестирования ядер этой системы, скомпилированных с разными опциями, или ядер разрабатываемой ветки.

Варианты загрузки через Lilo описываются в его специальном конфигурационном файле /etc/lilo.conf. Это простой текстовый файл, доступный для изменения в текстовом редакторе (при наличии полномочий суперпользователя, разумеется). Он содержит несколько секций: общую, описывающую условия загрузки в целом, и отдельные — описание каждого варианта загрузки.

Общая секция в виде отдельных строк содержит:

- имя устройства, с которого выполняется запуск Lilo (именно Lilo, а не Linux или иной ОС), для случая первого диска это будет

  `boot=/dev/hda`

- время ожидания выбора вариантов загрузки

  `timeout=##`

  где `##` — время в миллисекундах;

- имя варианта, загружаемого по умолчанию (должно соответствовать тому, что далее будет указано в секции этого варианта);
- указание на режим (нынче это, как правило, строка вида `lba32`).

Содержание отдельных секций различается в зависимости от того, предназначен этот вариант для загрузки Linux или иной ОС. В общем случае обязательными здесь будут две строки. Первая — это метка варианта (label), его уникальный идентификатор в виде произвольной, но, желательно, мнемонически прозрачной последовательности символов, например: `linux`, `freebsd`, `windows`. Вторая — имя устройства, несущего загружаемую ОС или его корневую систему (вроде /dev/hda1, /dev/hda5).

Секция варианта, загружающего Linux, кроме обязательного идентификатора, должна включать:

- имя файла, содержащего образ ядра, с указанием пути относительно корневого каталога, заданного в одной из следующих строк, что обычно выглядит как `image=/boot/bzImage` или `image=/boot/vmlinuz`;
- имя устройства, несущего корневую файловую систему, для случая первого раздела на первом диске это будет

  `root=/dev/discs/disc0/part1`

  при использовании файловой системы устройств или

  `/dev/hda1`

  без оной (или при использовании udev, *см. интермедию* после *главы12*);

- указание монтировать корневую файловую систему при загрузке ядра в режиме "только для чтения" read-only; это не значит, что она станет недоступной для изменений — в процессе инициализации файловые системы будут перемонтированы в соответствии с их описанием в /etc/fstab.

Кроме того, в Linux-секции /etc/lilo.conf вносятся строки, определяющие видеорежим при загрузке (например,

```
vga=791
```

означает режим графической консоли) и строки, передающие ядру некоторые параметры, вроде

```
append="devfs=nomount"
```

(запрет на монтирование файловой системы устройств, требующийся при задействовании механизма udev), или

```
append="ide-scsi"
```

(включение эмуляции протокола SCSI через IDE-шину).

Секции для не-Linux-вариантов, как правило, не содержат ничего, кроме имени устройства несущего раздела и метки варианта:

```
other=/dev/discs/disc0/part1
```

```
label=dos
```

Lilo может использоваться и исключительно для загрузки единичной Linux-системы в сочетании с любым мультисистемным загрузчиком, отвечающим за загрузку систем прочих (BSD Loader, GRUB, вплоть до NT Loader). В этом случае в общей секции /etc/lilo.conf в качестве загрузочного устройства следует указать имя корневого раздела Linux-системы, ведь тогда Lilo будет стартовать уже с его загрузочного сектора (а не с MBR-диска):

```
boot=/dev/hda1
```

В общей секции же можно раз навсегда определить имя устройства корневой файловой системы. А далее возможны и отдельные секции для вариантов загрузки, но они будут описывать уже не самостоятельные ОС, а разные версии или сборки ядра, возможно — видеорежимы или параметры ядра.

Как уже было сказано, Lilo напрямую с файловыми системами не работает — даже файловыми системами Linux. Конфиг же этого загрузчика лежит в каталоге /etc, представляющем собой часть файлового древа инсталлированной Linux-системы, которая при старте машины еще не загружена. Как же загрузчик узнает о собственной конфигурации?

Очень просто, о существовании собственного конфига в момент запуска Lilo и не подозревает. Необходимые данные для его опознания прошиты в виде бинарного кода в загрузочном секторе диска или раздела. И делается это одноименной программой /sbin/lilo. При запуске она обращается к файлу /etc/lilo.conf, исходя из значения строки

```
boot=/dev/имя_устройства
```

в общей его секции, определяет, куда нужно записать данные — в MBR диска или загрузочный сектор раздела, и выполняет запись вариантов загрузки, завершаемую сообщением

```
Added имя_рек
```

где `имя_рек` — метка (label) добавленного или модифицированного пункта меню.

Из сказанного становится очевидным, что каждое изменение конфигурации загрузчика должно сопровождаться его переустановкой — перезапуском программы /sbin/lilo.

Сказанным в предыдущих абзацах определяются и два принципиальных ограничения на использование Lilo вообще. Первое вытекает из положения конфига этого загрузчика внутри файловой системы Linux. Так что для изменения его конфигурации из другой операционной системы необходимо иметь доступ к той файловой системе, на которой лежит корень инсталлированного Linux, причем в режиме записи. И если примонтировать Ext2fs/Ext3fs к файловой системе FreeBSD или DragonFlyBSD в режиме read/write можно без проблем (ныне это не потребует даже пересборки ядра и той, и другой ОС), то доступ на запись из любой ОС BSD-семейства к разделам с ReiserFS или XFS в настоящее время невозможен (и неизвестно, будет ли возможен в обозримом будущем).

Конечно, можно взять за правило размещать корневую систему Linux только в разделе с Ext2fs/Ext3fs. Однако второе ограничение (необходимость перезапуска /sbin/lilo) этим не обходится, ведь эта программа предназначена для работы в родной ОС, сиречь Linux (интересно, а пробовал ли кто-нибудь запустить /sbin/lilo под FreeBSD в режиме Linux Compability?). И, соответственно, непременное условие для переконфигурирования Lilo — возможность запуска Linux в каком-никаком виде, хотя бы с rescue-дискеты или LiveCD.

Есть и третье ограничение, не столь важное: относительно слабые (по сравнению с GRUB) интерактивные возможности. Конечно, Lilo позволяет в режиме командной строки вмешиваться руками в ход загрузки — но только загрузки Linux же (например, передавать параметры ядру). Да и то в ограниченном объеме. Возможности же вмешательства в ход загрузки чужой ОС вообще заканчиваются в момент выбора соответствующего ей варианта.

Раз уж речь зашла об ограничениях Lilo, то, пользуясь случаем, подчеркну: прочие ограничения, на которые можно натолкнуться в литературе, как то: невозможность загрузить ядро Linux из области диска, лежащей за пределами первых 1024 его цилиндров, или с логического раздела в разделе Extended DOS, давно потеряли силу. И пользователь любого современного дистрибутива о них может смело забыть.

Так что основная сфера применения Lilo — это работа преимущественно (или исключительно) в Linux, при эпизодическом использовании какой-либо другой ОС. Причем рискну предположить, что под другой ОС будет выступать, скорее всего, какая-либо из версий Windows. Экспериментирование же с многочисленными ОС разных семейств — это, по моему мнению, прерогатива GRUB.

Как и Lilo, GRUB неоднократно был предметом описания в различных (преимущественно онлайновых) документах. Наиболее полное (из числа мне известных) содержится в цикле статей Владимира Попова, который можно найти здесь: **http://posix.ru/system/grub/**. К которой я отсылаю любознательного читателя, затронув далее лишь основные особенности и возможности этой программы.

Если Lilo, как и следует из его названия, создавался в первую очередь для загрузки Linux при сохранении возможности старта Windows, то целью разработчиков GRUB изначально было создание универсального мультизагрузчика, максимально независимого от любой операционной системы. История его началась в 1995 г., когда операционная система GNU Hurd (микроядерная ОС светлого будущего, столь же отдаленного, как и коммунизм в мировом масштабе) достигла той степени развития, что могла уже быть загружена. И встал вопрос: каким же образом это сделать, так как существовавшие тогда загрузчики оказались на это неспособными.

Конечно, с точки зрения идеологии GNU, логично было бы изобрести для этой цели собственный (еще один) загрузчик, в полной мере отвечающий идеалам свободы и демократии. Однако, к счастью для всего прогрессивного человечества, был избран иной путь: Multiboot Specification — универсальный метод загрузки ядер, этой самой спецификации соответствующих. Причем сама спецификация бралась отнюдь не с потолка, в ее основу легли особенности существующих ядер Linux и BSD-систем. Ведь, как я уже говорил, файлы их образов самодостаточны для запуска и несут в себе всю необходимую информацию, и дело упирается только в то, что в обычной ситуации они расположены на носителе с файловой системой, поддержка которых начинается только после загрузки соответствующего ядра. Так что суть универсальности предложенного метода и заключалась в том, чтобы загрузчик получил средства доступа к файловой системе.

Зримым же, действительно универсальным воплощением Multiboot Specification и стал мультисистемный загрузчик GRUB. Собственно, единственным его ограничением для полноценного использования (далее станет ясно, что "неполноценное" использование GRUB вообще ничем не ограничено) была совместимость с файловой системой загружаемой ОС. Ибо, в отличие от BSD Loader и Lilo, GRUB способен работать с очень многими файловыми системами напрямую, без всякого использования ресурсов соответствующей среды: "монтировать", читать с них данные, подвергать их ре-

дактированию и записывать изменения. Что превращает его в своеобразную мини-ОС, имеющую к тому же свой собственный шелл-подобный интерфейс пользователя.

Можно выделить две степени совместимости GRUB с файловыми системами — полную и частичную. Полностью совместимые файловые системы — это те, что способны нести GRUB сам по себе. И в их числе абсолютно все нативные файловые системы Linux (включая JFS, имевшие некогда место проблемы с "монтированием" ReiserFS, насколько я знаю, в прошлом), Minix и FAT. То есть загрузчик GRUB может стартовать с любого носителя, отформатированного соответствующим образом: дискового раздела, дискеты Linux (на которых применяется файловая система Minix) или DOS (с файловой системой FAT). Хотя по причинам, которые скоро станут понятны, разработчики рекомендуют размещать GRUB в самостоятельном разделе с файловой системой Ext2fs.

Частично совместимые с GRUB файловые системы сами по себе нести его не могут. Но после старта он способен их идентифицировать, прочитать и загрузить с них ядро, если оно отвечает Multiboot Specification. И в эту группу попадают практически все файловые системы свободных ОС, внутренний формат которых общедоступен: кроме Linux, тут мы видим также все варианты FFS и UFS, используемых BSD-семейством. Если же какая-то из таких файловых систем (как правило, из числа только что созданных) кажется несовместимой с текущей версией GRUB — это исключительно временное явление. Потому что для любой открытой файловой системы нет никаких препятствий получить свою поддержку в GRUB.

Приведу пример: с появлением пятой ветки FreeBSD обрела новую нативную файловую систему, UFS2, несколько отличную от прототипа (UFS просто). В результате единовременная версия GRUB (0.93) оказалась неспособной работать с ней напрямую (я покажу, что это все равно было обходимо). Однако уже для версии 0.94 появился патч UFS2, а с версии 0.95 GRUB поддерживает эту файловую систему, что называется, "из тарбалла".

Можно видеть, что в списке частично совместимых с GRUB файловых систем нет вариантов NTFS. Что понятно, ведь их внутреннее устройство — тайна за семью печатями. ОС, файловые системы которых для GRUB недоступны, могут быть загружены "по цепочке" — передачей управления на загрузочный сектор соответствующего раздела. Тот же способ можно применить и для загрузки свободных ОС, файловые системы которых временно не поддерживаются. На моей памяти таковыми бывали периодически некоторые версии OpenBSD и FreeBSD пятой ветви. К слову сказать, даже несовместимость ядра с Multiboot Specification, имеющая место для DOS/Windows9*x*/ME, не оказывается помехой для GRUB. И указанные ОС могут быть столь же успешно загружены цепочечным методом.

Таким образом, универсализм GRUB в рабочем (то есть установленном и настроенном) состоянии сомнений не вызывает. Однако с точки зрения установки и настройки он предстает столь же независимым от любой ОС. Ибо устанавливается он с самодостаточной загрузочной дискеты. А с недавнего времени, благодаря усилиям Владимира Попова, это можно проделать и с мультизагрузочного LiveCD (который можно скачать отсюда: **http://posix.ru/download.shtml**). То есть наличия инсталляции какой-либо ОС (или дистрибутива Linux) не требуется.

Что же касается конфигурирования GRUB, то, если следовать приведенным ранее рекомендациям разработчиков (установке GRUB в отдельный дисковый раздел с файловой системой Ext2fs, и резоны к тому сейчас прояснятся), то и тут никаких сложностей не предвидится ни в одной из свободных ОС. В Linux этот раздел монтируется как каталог /boot, куда, вместе с файлами собственно загрузчика (в подкаталоге /boot/grub) помещаются также файлы ядра (/boot/vmlinuz и подобные ему). Причем разработчики опять же рекомендуют монтировать /boot не автоматически при старте Linux (после ее загрузки никакой необходимости обращаться к этому каталогу в обычных условиях нет), а руками, по мере надобности (установки нового ядра или реконфигурации GRUB).

За настройку загрузки GRUB отвечает специальный файл, доступный из Linux как /boot/grub/menu.lst. Это простой текстовый файл, который можно модифицировать в любом редакторе. Причем, что ценно, предварительно проверив работоспособность вносимых изменений в интерактивном режиме. Описывать его формат в деталях не буду, он очень прост и станет ясным из приводимого в заключение примера.

Так что для конфигурирования же GRUB из какой-либо иной ОС требуется только возможность чтения раздела с Ext2fs и записи в него. А, как уже говорилось, получить такую возможность из FreeBSD и DragonFlyBSD ныне никакого труда не составит (да и в случае Net- или OpenBSD в худшем случае потребуется только пересборка их ядер). Так что рекомендация разработчиков о выборе файловой системы загрузочного раздела становится понятной, прочие файловые системы Linux BSD-семейством поддерживаются ограниченно (или не поддерживаются вообще), а FAT... ну, он FAT и есть, всерьез говорить о нем не стоит.

Таковы основные особенности (и возможности) GRUB. О деталях его установки, настройки и интерактивной работы в ходе загрузки можно было бы написать еще много. Памятуя и о прекрасной штатной документации этой программы, правда, в не любимом мною формате (командой `info grub`, стандартный вариант `man grub` содержит лишь краткие о ней сведения). Так что в качестве завершающего штриха просто приведу прокомментирован-

ный (строки комментариев, как обычно, отмечены символом #) пример своего конфига, который долгое время верой и правдой служил мне при загрузке Archlinux.

```
# Config file for GRUB — The GNU GRand Unified Bootloader
# /boot/grub/menu.lst

# Общая конфигурация загрузчика
timeout   5
       # Время ожидания выбора загружаемой ОС в секундах
default   0
       # ОС, загружаемая по умолчанию
       # (в данном случае Linux)
color light-blue/black light-cyan/blue
       # Цветовая гамма меню (мне такая нравится)

# Секция, отвечающая за загрузку Archlinux
# (0) Arch Linux
title  Arch Linux  [/boot/vmlinuz]
       # Идентификатор ОС
       # он же — пункт меню загрузчика
root   (hd0,1)
       # Устройство, несущее корневую файловую систему:
       # 2-й раздел 1-го диска в нотации GRUB
       # В нотации Linux ему соответствует устройство
       # /dev/discs/disc0/part2 в следующей строке
kernel (hd0,0)/vmlinuz root=/dev/discs/disc0/part2 ro
       # где
       #  (hd0,0)/vmlinuz — положение файла образа ядра
       # на загрузочном устройстве в нотации GRUB -
       # 1-м разделе 1-го диска
       # root=/dev/discs/disc0/part2
       # имя устройства с корневой файловой системой
       # уже в нотации Linux (при использовании devfs)
       # ro — предписание монтировать его в режиме read-only
```

Задачи инициализации

Итак, тем или иным способом загрузка ядра и всего сопутствующего ему хозяйства успешно завершена. В дело вступает главный калибр любой POSIX-системы — процесс `init`. Это первый (в прямом и переносном смысле, его PID равен единице) пользовательский процесс, и запускается он исполнением одноименного файла /sbin/init.

В действительности это могут быть (и в разных системах действительно бывают) весьма разные программы. Более того, его можно подменить при интерактивном управлении процессом загрузки другой программой, например, командной оболочкой. Однако это сейчас не очень важно, рассмотрим только штатные задачи программы /sbin/init.

Первой из таких задач, как по времени исполнения, так и по значению, является проверка целостности наличных файловых систем. Для начала каждая из них проверяется на наличие бита "чистого размонтирования" (clean byte), который автоматически устанавливается в ходе правильного завершения предыдущего сеанса работы. Если такой бит обнаруживается в каждой файловой системе, все хорошо, дело движется дальше. Если нет, возможны варианты, о которых я скажу позднее.

Следует отметить, что сам по себе "бит чистого размонтирования" отнюдь не гарантирует сохранности файловой системы и, особенно, ее данных. Он лишь показывает, что файловая система была корректно размонтирована в предыдущем сеансе. В этом случае процесс `init` делает не лишенное резона допущение, что с метаданными и данными ее все в порядке, и переходит к выполнению следующей задачи.

А следующая задача процесса `init` — это вызов и отработка сценариев инициализации, или стартовых скриптов, собранных в каталоге /etc и его подкаталогах. Они столь сильно зависят от ОС (а внутри Linux еще и от конкретного дистрибутива), что дать их обобщенную характеристику практически невозможно. Можно только констатировать, что сценарии инициализации — это обычные сценарии оболочки, рассчитанные на исполнение стандартным POSIX-шеллом (/bin/sh в BSD-системах и /bin/bash — в Linux). Они включают в себя последовательности команд, призванные монтировать файловые системы, активизировать область свопинга, устанавливать системные часы, запускать те или иные службы и демоны, включая сетевые соединения.

Команды, образующие стартовые сценарии, получают свои опции, их значения и аргументы из специальных файлов конфигурации, также имеющих своим местопребыванием /etc и его подкаталоги. Конфигурационные файлы представляют собой либо простые базы данных опций и аргументов команд либо списки имен переменных (соответствующих опциям команд, исполь-

зуемых в скриптах) с присвоенными им значениями. Конфиги от скриптов легко отличимы при просмотре каталога /etc отсутствием у первых бита исполнения.

Сказанное может показаться не очень понятным, поэтому попытаюсь продемонстрировать на примере. Например, обязательная процедура на стадии отработки сценариев инициализации — монтирование необходимых файловых систем в режиме чтения/записи; и перемонтирование, ведь при загрузке ядра несущая его корневая файловая система монтируется в режиме "только для чтения". Это выполняется прямой директивой

```
$ mount -a
```

предписывающей смонтировать все файловые системы. А вот что понимается под словом "все" (имя опции `-a` от all), то есть список аргументов (устройств и точек монтирования), а также опций, с которыми должна быть смонтирована та или иная файловая система, и составляет содержание специальной базы данных, хранимой в файле /etc/fstab. Мы уже знакомились с ним в *интермедии* после *главы 10*, однако позволю себе напомнить обобщенный его формат (тем паче, что это один из немногих конфигов, формат которого одинаков во всех POSIX-системах).

Это очень простая база данных, каждая запись которой соответствует подлежащей монтированию файловой системе, а поля, разделителем которых являются символы пробела (пробелов) или табуляции, следующие:

```
имя файла устройства, несущего файловую систему;

точка монтирования — каталог в файловой иерархии;

тип файловой системы;

опции монтирования (часто имеет значение default).
```

Содержимое первых двух полей каждой записи передается команде `mount` из стартового сценария в качестве первого и второго ее аргументов, остальных двух — как опции, обязательные (тип файловой системы) и необязательные (все прочие).

Последовательное разделение стартовых сценариев и их конфигурационных файлов — один из краеугольных принципов общесистемного конфигурирования. В сущности, пользователю при нормальном ходе настройки практически нет необходимости ни знакомиться с содержимым скриптов (хотя это и не вредно), ни, тем паче, менять в них что-либо (последнее допустимо только в том случае, если этот самый пользователь точно знает, что делает, иначе систему легко довести до неработоспособного состояния). А вот вносить изменения в значения параметров конфигурационных файлов не только можно, но и нужно, разумеется, такое разрешение не избавляет пользователя от понимания смысла своих действий.

Наконец, третья непременная задача процесса `init`: в *главе 7* мы рассмотрели получение терминала (запуск процесса `getty`), установку его свойств и подготовку к авторизации — вытеснение его процессом `login`. Эта процедура также по-разному выполняется в разных системах, хотя тут многообразие не столь велико, как при отработке стартовых скриптов.

В ходе инициализации могут выполняться и некоторые другие задачи, скажем, конфигурирование приложений, не входящих в базовую систему, но, тем не менее, запускаемых в качестве стартовых сервисов (демонов). Поскольку такие программы устанавливаются отдельно от системы (и не в обязательном порядке), сценарии их запуска и сопряженные с ними конфиги могут лежать не в каталоге /etc, а в других, подчас весьма неожиданных, местах (/usr/etc, /usr/local/etc). Типичным примером здесь является httpd — демон, управляющий Web-сервером Apache. Однако все это уже необязательные составляющие этапа инициализации.

Описанная последовательность инициализации происходит, если ни на одной из стадий не происходит ошибок. Если же таковые случаются, по причине аппаратных ли сбоев или ошибок пользователя при настройке, все может протекать совершенно иначе.

Первый потенциальный источник ошибок инициализации — отсутствие "бита чистого размонтирования", выявляемое при начальной проверке файловых систем. В этом случае делается вывод о некорректности завершения предыдущего сеанса и возможности существования противоречий в ней. Таковыми могут быть, например, записи в каталогах, которым не соответствуют идентификаторы каких-либо файлов. Или, напротив, идентификаторов файлов, не приписанных ни к каким каталогам.

Для проверки, действительно ли такие противоречия имеют место быть (а отсутствие clean byte отнюдь не влечет их неизбежности), и исправления выявленного безобразия запускается утилита проверки файловой системы fsck (это разные программы, в зависимости от типа проверяемых файловых систем). Если такая проверка завершается успешно, то есть противоречий в структуре файловой системы нет, опять же все хорошо, процесс `init` возвращается к выполнению своих задач, если нет — снова возможны варианты, зависящие от ОС и "тяжести повреждений". Худший случай, когда серьезные противоречия обнаруживаются в корневой файловой системе, это знаменует окончание нормальной инициализации с переходом в однопользовательский режим, когда монтируется только корневая файловая система — и в режиме read only.

Ошибки при отработке скриптов инициализации также влекут разные последствия. Обычный (и наиболее легкий) случай: сервис, в сценарии запуска которого произошла ошибка (а она может быть вызвана, например, неправильным указанием опций или аргументов в соответствующем конфиге), просто не будет доступным после загрузки системы. Например, так будет с

демоном консольной мыши в Linux gpm, если в его конфиге неправильно был указан протокол или интерфейс мыши.

Более серьезные последствия будут иметь ошибки при монтировании файловых систем, самый частый источник которых — синтаксически неправильное их описание в файле /etc/fstab (то есть элементарные опечатки). Особенно серьезной проблемой будет невозможность монтирования корня файлового древа, в этом случае ядро впадет в панику (Panic mode), и продолжение загрузки окажется невозможным.

Наконец, ошибки в процессе получения терминала (возможные при экспериментах, например, с автоматической регистрацией в системе). Они также достаточно неприятны и, как правило, приводят к невозможности начала нормальной работы.

Впрочем, оснований для паники нет: даже при впадении в панику ядра все возможные при инициации системы ошибки исправимы тем или иным способом. Ибо легкий флирт, в том числе и с операционной системой, подобно насморку, переносится на ногах, и постельный режим необходим лишь в тяжелых случаях. В одних ситуациях достаточно завершить процесс загрузки и чуть подправить конфиги, в других — загрузиться в однопользовательском режиме и провести проверку и ремонтирование файловых систем вручную, в-третьих придется грузиться с rescue-носителя (например, LiveCD). А вот хирургического вмешательства (сиречь полной переустановки системы), скорее всего, не потребуется никогда.

Оборотная сторона инициализации системы — это ее останов или рестарт, различий между этими процессами практически нет. И отвечает за него команда `shutdown`, которая может быть дана от лица суперпользователя или члена группы operator. С опцией `-h` она вызывает останов машины, с опцией `-r` перезагрузку. И еще команде этой требуется аргумент — время, когда останов или рестарт должны произойти. Впрочем, есть способ и мгновенного останова или рестарта:

```
$ shutdown -h now
```

или

```
$ shutdown -r now
```

Во всех, насколько мне известно, POSIX-системах существуют также команды `halt` и `reboot` того же назначения. Однако самостоятельной роли они не играют, просто вызывая команду `shutdown` с опцией останова и перезагрузки соответственно.

Останов системы происходит в порядке, обратном ее инициализации. Сначала делается попытка корректного завершения всех пользовательских процессов отправкой им сигнала `TERM`. По истечении некоторого промежутка времени всем еще "живым" процессам отправляется сигнал `KILL` — для гарантированного их убиения. Затем стопорятся все стартовые сервисы и де-

моны. Наконец, содержимое дисковых кэшей записывается на диск (посредством команды `sync`), и размонтируются файловые системы. После этого обычно появляется сообщение о возможности безопасного отключения питания машины, или оно отключается автоматически. При рестарте все происходит точно так же, но после останова системы автоматически начинается перезагрузка машины. Весь процесс останова и рестарта определяется соответствующим сценарием, подобным сценариям инициализации.

Важно (и это одно из первых правил техники безопасности), что UNIX-машину не рекомендуется останавливать простым отключением питания. Это чревато разными неприятностями:

- потери данных в запущенных программах: неслучайно первое действие при останове — попытка их сохранения сигналом `TERM`;
- потери как бы сохраненных, но еще не записанных (то есть кэшированных) изменений в открытых файлах;
- отсутствие бита чистого размонтирования в нежурналирумых файловых системах, вызывающее их более или менее длительную проверку при рестарте;
- более или менее серьезные нарушения целостности файловых систем, вплоть до полного их разрушения (впрочем, последнее нынче случается крайне редко).

В современных версиях POSIX-систем в той или иной мере реализована поддержка управления питанием стандарта ACPI (Advanced Configuration and Power Interface). При ее включении в принципе становится допустим останов системы простым отключением питания машины, на нажатие кнопки питания на корпусе компьютера система реагирует точно так же, как и на команду

```
shutdown -h now.
```

Однако прибегать к этому следует только в случае уверенности в правильности настройки и функционирования модулей ACPI: и в Linux, и в BSD-системах эти опции экспериментальны, и их безошибочная работа не гарантируется.

Стили инициализации

Мое описание инициализации POSIX-системы поневоле получилось чрезвычайно обобщенным. Попробую перейти к конкретике, для чего нам потребуется представление о стилях инициации.

Стилей инициации два: BSD-стиль, повсеместно принятый в одноименных системах, и стиль System V, используемый в большинстве распространенных дистрибутивов Linux. В некоторых дистрибутивах Linux, при сохранении

родовых пятен System V (таких как уровни выполнения, о которых будет говориться далее), применяются BSD-подобные стартовые сценарии, и это вполне можно считать третьим, промежуточным, стилем.

Это размежевание лишь в первом приближении.

Особенности BSD-стиля

Начнем с чистого BSD-стиля, как более простого, прямолинейного и потому более доступного пониманию начинающего пользователя (да и, по моему глубокому убеждению, более ему подходящего). Начать с того, что он предусматривает всего два режима загрузки: однопользовательский, предназначенный для аварийно-восстановительных работ и решения некоторых задач администрирования, и многопользовательский, при котором осуществляется вся нормальная деятельность пользователя.

В однопользовательском режиме загрузка происходит при выборе соответствующего пункта **Boot in single user mode** в меню начального загрузчика, при задании команды `boot -s` в командной строке загрузчика (после выбора пункта его меню **Escape to loader prompt**) и при обнаружении серьезных (неустранимых автоматически) нарушений целостности файловой системы в ходе ее проверки на первой стадии инициализации).

В любом случае при загрузке в однопользовательском режиме не монтируется ни одна файловая система из /etc/fstab, кроме корневой, да и та остается лишь в режиме read only, в котором она была смонтирована при старте системы; не отрабатывается ни один сценарий инициализации и не активизируется ни один виртуальный терминал, кроме первого, исполняющего функции системной консоли. Очевидно, что никакая нормальная работа при этом невозможна, однопользовательский режим предназначен для аварийно-спасательных работ.

При загрузке в многопользовательском режиме все стадии инициации проходятся по полной программе: монтируются предназначенные к тому файловые системы из файла /etc/fstab (а корень ее ремонтируется в режиме чтения/записи), отрабатываются определенные стартовые скрипты (где и кем определенные — скоро увидим) и активизируются все описанные в файле /etc/ttys виртуальные терминалы (вплоть до графического приглашения к авторизации, если таковое определено). Авторизация возможна как для администратора, так и для любого пользователя, но о беспарольном входе придется забыть. Короче говоря, идет нормальная цивилизованная работа...

Между однопользовательским и многопользовательским режимами не лежит непреодолимой пропасти: переход из одного режима в другой возможен не только при рестарте машины, но и в ходе одного сеанса. Для немедленного перехода в однопользовательский режим служит команда

```
$ shutdown now
```

Возврат обратно в многопользовательский режим происходит по команде

```
$ exit
```

Такой способ широко применяется для реинициализации системы без полной ее перезагрузки после изменения каких-либо конфигурационных параметров. Нужно только помнить, что при этом не все сервисы и демоны обязаны работать правильно.

Загрузка в многопользовательском режиме (и это отличительная особенность BSD-стиля) потенциально влечет за собой доступность для запуска абсолютно всех системных служб и демонов: инициализирующие их сценарии (располагающиеся в каталоге /etc/rc.d) теоретически могут быть запущены из главного стартового сценария — файла /etc/rc. А вот какие из них будут запущены реально, определяется опциями его конфигурационного файла /etc/rc.conf.

При установке системы, использующей BSD-стиль инициализации, в каталог /etc на диске записывается некий /etc/rc.conf, строки имеют вид

```
servicename_enable="значение"
```

или

```
переменная="значение"
```

Значение строк первого вида: `"YES"` или `"NO"`. Легко догадаться, что они разрешают (или запрещают) запуск поименованного сервиса посредством соответствующего ему (и, как правило, одноименного) сценария из каталога /etc/rc.d (для определенности я рассматриваю случай FreeBSD и DragonFlyBSD, стартовые схемы их практически идентичны, но нечто подобное будет и в NetBSD, и в OpenBSD). Например, строка

```
moused_enable="YES"
```

разрешает запуск службы консольной мыши посредством скрипта

```
/etc/rc.d/moused
```

Значения же строк второго вида — это параметры, передаваемые командам, входящим в скрипты инициализации. Так, строки

```
moused_type="auto"
moused_port="/dev/ums0"
```

определяют, что значения опций `-t` (автоматическое определение протокола) и `-p` (имя файла мышиного устройства в данном случае с USB-интерфейсом) для команды /usr/sbin/moused, запускаемой из сценария /etc/rc.d/moused.

Как правило (и это тоже традиция BSD-систем), по умолчанию в файле /etc/rc.conf разрешен запуск лишь минимально необходимого для начала работы количества системных служб. Большая же их часть обычно запрещена, или явным образом, указанием значения `"NO"`, или по умолчанию (а от-

куда берутся умолчания — мы скоро увидим). Так что включение необходимых пользователю демонов (например, той же консольной мыши) — дело рук самого пользователя.

Откуда берутся стартовые умолчания? А берутся они из файла /etc/defaults/rc.conf, в котором описаны всевозможные (и все возможные) стартовые сервисы и их параметры. Файл этот не предназначен для прямого редактирования (хотя оно и не запрещено атрибутами его доступа). Вместо этого полагается отыскать в нем строки, относящиеся к нужным сервисам, перенести их в /etc/rc.conf и разрешить их запуск (или, напротив, запретить, если он разрешен по умолчанию, но в данном случае не нужен). Уточняющие опции сервисов также берутся из /etc/defaults/rc.conf, переносятся в /etc/rc.conf, и им приписываются нужные значения.

В общем виде это делается, например, так: в одной виртуальной консоли (на которой нужно зарегистрироваться как root или получить его права командой `su`) в текстовом редакторе открывается файл /etc/rc.conf, в другой (на ней можно авторизоваться и обычным пользователем) дается команда

```
$ less /etc/defaults/rc.conf
```

И нужные строки из последнего просто переносятся в первый, где и модифицируются должным образом. Не вредно при этом задействовать и третью пользовательскую консоль, для чтения

```
man (5) rc.conf.
```

Что же касается заключительной стадии BSD-инициации (процесса получения терминала), то она контролируется записями в файле /etc/ttys. Наличие специального файла для конфигурирования виртуальных терминалов — одно из важных отличий BSD-систем: в Linux, вне зависимости от стиля сценариев инициализации, принятых в конкретном дистрибутиве, настройка виртуальных терминалов описывается в общем конфиге процесса `init`.

Как уже говорилось, сценарии, отвечающие за запуск стартовых сервисов, находятся в каталоге /etc/rc.d (по крайней мере, в современных версиях FreeBSD, а также в DragonFlyBSD это именно так). Большинство запускаемых ими программ — это так называемые демоны (daemon — Disk And Execution MONitor), нечто вроде резидентных программ, работающих в фоновом режиме в ожидании запроса на исполнение их функции (печати, отправки почты, обращения к FTP- или HTTP-серверу и т. д.). В соответствии с этим, запускающие их сценарии устроены по принципу `start` — `stop`. И если при инициации системы выполняется первая функция, то при ее останове, как легко догадаться, вторая.

Ход процесса останова системы определяется сценарием /etc/rc.shutdown. Назначение его — выполнить функцию `stop` в сценариях всех сервисов, запущенных из скрипта /etc/rc в соответствии с описанием, данным в /etc/rc.conf и /etc/defaults/rc.conf.

Стили System V

Основу инициализации в стиле System V, в более или менее чистой форме, принятой в Linux, составляет понятие runlevels. Это один из тех терминов, любой русский перевод которого способен только сбить с толку начинающего пользователя. Потому что часто используемые переводы типа "уровни запуска" или "уровни загрузки" создают впечатление, что система в процессе инициализации последовательно проходит некие, соответствующие им, стадии. На самом деле, как мы видели, стадии инициализации действительно имеют место быть, вот только к runlevels они имеют отношение косвенное.

Понять, что же такое runlevels в System V проще всего в сравнении с BSD-инициализацией. В ней, как мы только что видели, существуют два режима: однопользовательский, при котором не исполняются никакие стартовые сценарии, и многопользовательский, при котором исполняются все сценарии, разрешенные в конфигах /etc/rc.conf и /etc/defaults/rc.conf. Так вот, runlevels — это нечто вроде таких же режимов, только их больше, и каждому поставлен в соответствие набор исполняемых при его вызове сценариев.

Вызов runlevels, то есть метасценария для группы сценариев, осуществляется при инициализации системы первым ее пользовательским процессом — процессом `init`. А сами эти метасценарии описаны в главном конфигурационном файле этого процесса /etc/inittab. Конфиг этот определяет весь ход процесса `init`, от проверки файловых систем до получения терминала. И является непременным атрибутом всех (по крайней мере, известных мне) Linux-систем, отличающим их от систем BSD-клана (где, как мы видели, единого конфигурационного файла для процесса `init` нет).

Теоретически процесс `init` предусматривает вызов семи runlevels: от нулевого до шестого; за тремя из них зафиксированы метасценарии определенного назначения, использование остальных отдано на откуп создателям дистрибутивов. Зарезервированы следующие runlevels:

- 0 — halt, останов системы;
- 1 — single user mode, однопользовательский режим;
- 6 — reboot, перезагрузка системы.

Почему останов и перезагрузка системы выступают в одном ряду с режимами ее запуска, понять вообще невозможно, даже отвлекаясь от перевода, но это уже фирменный стиль System V, который остается только запомнить.

Опять, однако, отвлекся — вернусь к прочим runlevels. Их использование свободно. Как правило, один из свободных номеров (обычно 3 или 4) задействуется под нормальный многопользовательский режим работы, другой же (4 или 5) — под режим запуска Иксового сеанса (то есть графический). В некоторых дистрибутивах предусматривается еще и многопользователь-

ский режим без поддержки сети. Вот как это выглядит в наиболее распространенном из дистрибутивов Linux — Red Hat (Fedora Core):

- 0 — останов системы;
- 1 — однопользовательский режим;
- 2 — многопользовательский режим без поддержки сети;
- 3 — полный многопользовательский режим;
- 4 — не используется;
- 5 — полный многопользовательский режим X-сессии;
- 6 — перезагрузка системы.

В других дистрибутивах Linux количество задействованных runlevels в интервале со 2 по 5 и их соответствие режимам может быть иным: для конкретного дистрибутива это можно посмотреть в файле /etc/inittab, в начале которого в виде комментария всегда содержится список.

А вообще описание действий по инициализации в файле /etc/inittab начинается с вызова первого исполняемого в ее ходе сценария. Имя его и расположение очень зависят от дистрибутива. В Red Hat, например, это будет `/etc/rc.d/rc.sysinit`. Функциональность также варьирует от системы к системе, но обязательны такие функции, как проверка файловых систем, их монтирование, активизация swap-раздела.

Следующее действие из предписанных `/etc/inittab` — определение режима по умолчанию, то есть все того же runlevels, что воплощается в строке вида

```
id:3:initdefault:
```

и автоматически влечет за собой исполнение набора скриптов, за режимом закрепленных.

Скрипты эти сгруппированы в определенных подкаталогах каталога /etc. В Red Hat это будут /etc/rc.d/rc0.d, /etc/rc.d/rc1.d и так до /etc/rc.d/rc6.d, в Debian — /etc/rc0.d, /etc/rc1.d и до /etc/rc6.d, в произвольном дистрибутиве Linux — даже и гадать не берусь. Однако общая закономерность в том, что цифра в имени подкаталога прямо соответствует номеру runlevels. И именно скрипты из подкаталога вида /etc*0* будут отрабатываться при останове системы, /etc*1* — при загрузке в однопользовательском режиме и т. д.

Возникает вопрос: а не повторяются ли сценарии в подкаталогах разных runlevels. Ведь очевидно, что для запуска многопользовательского режима без поддержки сети, полного многопользовательского режима и режима графического нужно выполнить существенно пересекающиеся множества действий. И различия лишь в том, что во втором случае добавляется запуск сетевых служб, а в третьем — еще и менеджера графического входа в систему (например, xdm).

Ответ, как ни странно, будет отрицательным. Потому что на самом деле реальных сценариев, соответствующих runlevels, в указанных каталогах нет. Все стартовые скрипты собраны в отдельном подкаталоге с именем вроде /etc/init.d (не поручусь, что оно будет одинаковым во всех дистрибутивах) без всякого разделения, а в подкаталоги вида /etc*#* помещаются лишь символические ссылки на них. Кстати, и упоминавшийся ранее первый инициализационный скрипт `/etc/rc.d/rc.sysinit` — тоже симлинк на `/etc/init.d/rc.sysinit`.

Более того, в каждом runlevel-каталоге ссылки эти имеются в двух экземплярах: одна ссылка, маркированная буквой `S` в имени, отвечает за старт сервиса, другая же, имеющая в имени литеру `K` (от kill), за его остановку. Это достигается за счет того, что в первом случае скрипт вызывается с опцией `start`, во втором же — с опцией `stop`. Исключение составляют подкаталоги, соответствующие runlevels 0 и 6: стартовых ссылок в них быть не может, а есть только стоповые.

Порядок запуска стартовых сценариев при рассматриваемой их схеме может быть важен: отдельные сетевые службы могут быть запущены только после включения общей поддержки сети. И обеспечивается этот порядок двузначными цифрами в именах ссылок (но не самих скриптов) в каждом runlevels-каталоге: чем меньше цифра, тем раньше запускается сценарий, которому эта ссылка соответствует.

Осталось определить, откуда берутся значения опций и аргументов для команд, образующих сценарии инициализации. А берутся они из собственных конфигурационных файлов, ведь принцип разделения скриптов и конфигов неукоснительно проводится и в схеме System V. Конфиги эти могут быть собраны в едином подкаталоге типа /etc/conf.d или разбросаны по всему каталогу /etc.

Наконец, с запуском сценариев инициализации покончено. Остается последнее из предписанных процессу `init` действий — получение терминалов. В схеме System V оно также описывается в файле /etc/inittab и выглядит примерно следующим образом (дано на примере Red Hat/ASPLinux):

```
# 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
```

В данном примере 6 виртуальных терминалов активизируются при всех runlevels (кроме однопользовательского режима, разумеется, останова и пе-

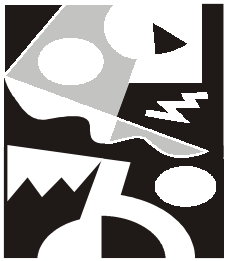
резагрузки). А при вызове runlevels 5 (напомню, здесь это X-сессия) на первой же свободной (то есть неактивизированной) консоли запускается менеджер графического входа в систему, за что отвечает такая строка:

```
x:5:respawn:/etc/X11/prefdm -nodaemon
```

Я попытался дать максимально обобщенное описание процесса инициализации в стиле System V. Более ясное представление о ней можно получить, только рассматривая реализации в конкретных дистрибутивах. Так что заинтересованным читателям рекомендую обратиться к дополнительным источникам информации, из которых я выделил бы Linux Startup Manual (**http://ctdp.tripod.com/index.html**), с той только оговоркой, что слово Linux в его заглавии следовало бы заменить на Red Hat, ибо в нем описана схема инициализации именно этого дистрибутива. А за более обобщенным описанием схемы System V и принципов построения собственной схемы загрузки в этом стиле можно обратиться к LFS Book Герарда Бикманса (**http://hints.linuxfromscratch.org/**) или какому-либо из ее русских переводов: **http://multilinux.sakh.com/lfs/** или **http://linux.yaroslavl.ru/docs/book/lfsbook/**.

Схема инициализации в стиле System V может показаться излишне усложненной. Хотя, по словам лично мне знакомых администраторов сетей, позволяет очень гибко управлять различными сервисами. Для конечного пользователя такое усложнение неоправданно. И, как выяснилось, это мнение не только мое. Иначе чем объяснить стремление прикрутить сценарии инициализации в BSD-стиле в ряде современных дистрибутивов Linux, таких как Gentoo, CRUX, Archlinux, не говоря уже о дедушке дистрибутивов Slackware. Разумеется, и в этом случае никуда не деваются ни runlevels, ни /etc/inittab. BSD-подобный облик схемы их загрузки приобретают за счет минимизации задействованных значений runlevels и наличия главного конфигурационного сценария (типа /etc/rc), конфигурируемого посредством единого /etc/rc.conf. Наиболее последовательно эта схема претворяется в жизнь создателями дистрибутива Archlinux, в котором понятие runlevels практически не используется.

Глава 14



Принципы сборки и установки пакетов

- Очень элементарное введение.
- Правила сборки.
- Три волшебных слова.
- Особенности сборки ядра.
- Вопросы оптимизации.
- Средства управления пакетами.

Как уже говорилось в *главе 2*, дистрибутивы Linux организованы по пакетному принципу. Точно так же, в виде пакетов, распространяются и любые программы, создаваемые независимыми разработчиками (из которых в основном и собираются дистрибутивы Linux). А в BSD-системы попакетно включаются все приложения, не входящие в состав базового комплекта. И потому одна из важных задач пользователя — это интеграция пакетов в свою систему.

Очень элементарное введение

В большинстве случаев эта задача решается за пользователя разработчиками: системы управления пакетами (менеджеры пакетов) составляют неотъемлемую часть любого дистрибутива Linux, Free- и прочих BSD. Однако в ряде случаев пользователю приходится устанавливать пакеты и самостоятельно. К тому же понимание сути этого процесса способствует уяснению того, что же делают пакетные менеджеры, и помогает принять правильное решение в нештатных ситуациях. Так что вопрос этот заслуживает подробного рассмотрения.

Само по себе выражение "сборка программы" часто повергает начинающего пользователя в некий священный трепет — по себе помню. Однако в этой

главе я постараюсь показать, что ничего сверхъестественного в этом процессе нет, и выполнение его по силам любому пользователю, вне зависимости от чисто программистской квалификации.

Однако сначала маленькое введение для тех, чье компьютерное образование начиналось не с книжек Брябрина и Фигурнова, а с руководств типа "Word за 5 минут" или "Quark Press для полных идиотов" (это не в обиду читателям, но в упрек писателям). Все прочие могут смело пропустить следующие элементарные рассуждения.

Так вот, программы, то есть наборы инструкций, предписывающие машине выполнить то или иное действие, представляют собой самые обычные тексты (называемые исходными текстами, исходниками, в которых необходимые инструкции описываются в соответствие с принятыми в данном языке правилами (синтаксисом языка). Собственно говоря, именно это мы и проделывали в разделе о сценариях из *главы 12*.

Однако не следует чересчур уж очеловечивать компьютеры и ожидать от них способности понимать какой-либо "человеческий" язык, даже такой формализованный, как язык программирования. Нет, они способны воспринять только собственные инструкции (зависящие от центрального процессора), представляемые в бинарном виде, то есть последовательностей нулей и единиц. Собственно, и сами эти цифры выше их понимания, каковое не выходит за пределы наличия/отсутствия электрического сигнала, но в такие глубины мы лезть уже не будем.

Так что для выполнения программы она, в конечном счете, должны быть транслирована в соответствующие, зависящие от архитектуры процессора, машинные инструкции. И выполняется этот процесс двояко, посредством интерпретации или компиляции.

Интерпретация — это последовательный перевод языковых конструкций из символов латинского алфавита (и прочих, специальных) в машинные инструкции по мере их ввода. Простейший процесс интерпретации — это ввод директив в командной оболочке (почему она часто называется также командным интерпретатором), а также обработка ее сценариев — наборов элементарных команд. Сколь бы ни был сложен такой сценарий, выполняется он последовательно: сначала интерпретируется команда 1, потом команда 2 и т. д.

При интерпретации никакого изменения исходного текстового файла не происходит. А сам он отличается от простого списка команд только тем, что имеет бит исполнения. Однако интерпретируемая программа не может исполняться сама по себе, для ее запуска требуется соответствующая среда — программа-интерпретатор (например, та же командная оболочка).

Процесс интерпретации выполняется каждый раз при исполнении программы. И, соответственно, каждый раз время затрачивается на выполнение одних и тех же процедур. Так что возникает резонный вопрос: а нельзя ли оттранслировать исходник программы в машинные инструкции раз и навсегда, и в дальнейшем запускать на исполнение уже набор таких инструкций, не затрачивая время на их преобразование?

Ответ столь же резонен: ну, конечно же, можно. И процедура эта называется компиляцией, а выполняющие ее программы — компиляторами. В ходе этой процедуры из исходного текстового файла по определенным правилам создается файл бинарный, образованный, если просмотреть его в текстовом редакторе, последовательностью неудобопонятных символов.

Предварительно оттранслированные (прекомпилированные) программы, по вполне очевидным причинам, выполняются много быстрее, чем программы интерпретированные, причем разница в скорости нарастает с объемом. И потому все масштабные программы, как правило, пишутся в расчете на использование в откомпилированном виде. Хотя и роль интерпретируемых программ (сценариев разного рода) в POSIX-системах не стоит недооценивать.

В соответствии с дальнейшим предназначением программ, при их написании выбираются инструментальные средства для этого, то есть, в первую очередь, языки программирования: интерпретируемые и компилируемые.

К первым принадлежат постоянно упоминавшиеся ранее языки командных оболочек. Однако ими список интерпретируемых языков не исчерпывается. В POSIX-системах широко используются такие мощные средства, как Perl, Python, Ruby, Tcl/Tk. Они предоставляют большие возможности, вплоть до создания графических пользовательских инструментов. Однако принципиально написанные на них программы ничем не отличаются от сценариев командной оболочки.

Компилируемых языков также великое множество, начиная с пресловутого Basic и заканчивая специализированными средствами типа FORTRAN. Однако в POSIX-системах наибольшее значение имеют программы на языке C, на котором написана большая часть ядра всех POSIX-совместимых систем (а ядро ОС — это почти такая же компилируемая программа, как и любая другая), а также большая часть их приложений.

Так что далее речь пойдет о сборке преимущественно C-программ. Однако для нас это существенного значения не имеет, ведь собственно программированием заниматься мы не будем, а принципы сборки практически не зависят от используемого языка. В частности, точно по той же схеме собираются и графические приложения, в которых широко используется язык C++.

Кроме собственно программ, предназначенных для непосредственного исполнения, существуют еще так называемые разделяемые библиотеки, или библиотеки функций (соответствующего языка программирования).

Все программы, вне зависимости от их назначения, неизбежно должны выполнять некоторые однотипные действия, как-то: открыть файл, закрыть его, вывести на экран и т. д. Сущность их не меняется, что бы программа ни делала. И потому нет никакого смысла программировать такие манипуляции каждый раз заново.

Вот их, как правило, и не программируют. А объединяют соответствующие директивы в отдельные программные комплексы, именуемые библиотеками. Сами по себе они к автономному исполнению не пригодны. Однако любая программа, при необходимости совершить одно из типовых действий, вызывает из такой библиотеки некий фрагмент кода, содержащий требуемую последовательность директив.

Библиотеки привязаны к определенным языкам программирования, синтаксису которого подчиняются описания директив (функции, о них говорилось в *главе 12*). Поскольку наиболее употребимым в POSIX-системах и их приложениях является язык C, то его функции и требуются чаще всего. Они собираются в главную системную библиотеку, которая именуется обычно libc (Library C), хотя реально это разные комплексы, отличающиеся полнотой функций и их описанием и зависящие от конкретной операционной системы.

В подавляющем большинстве дистрибутивов Linux используется реализация главной системной библиотеки, именуемая glibc (GNU Library C); специализированные дистрибутивы могут использовать и другие библиотеки (например, uclibc, менее функциональную, но более компактную). Главная системная библиотека FreeBSD называется просто libc и функционально близка к glibc.

В POSIX-системах используются библиотеки свойств терминала (например, ncurces) для консольных программ и библиотеки, описывающие процедуры управления окнами: для графических программ системы X (xlib), библиотеки интерфейсных элементов и графических примитивов (Motif, Qt, Gtk), библиотеки описания графических и мультимедийных форматов. Короче говоря, существует тенденция к вынесению в разделяемые библиотеки всех повторяющихся действий и элементов. И в этом одна из причин компактности большинства классических UNIX-программ, в том числе и предназначенных для работы в графическом режиме.

Правила сборки

Ну, все, с элементарным введением покончено. Переходим собственно к пакетам и их сборке.

Все открытые и свободные программы и разделяемые библиотеки распространяются их разработчиками в исходных текстах. Конечно, никто не запрещает им создавать и прекомпилированные версии своих творений (и многие создатели программ так и поступают, предлагая один или несколько вариантов таковых, рассчитанные обычно на наиболее распространенные дистрибутивы Linux, иногда FreeBSD, реже — другие BSD-системы). Однако это скорее исключение, чем правило.

Наборы исходников объединяются разработчиками в пакеты. Пакет — понятие очень широкое и многогранное. Это может быть и простая монофункциональная утилита (например, строчный текстовый редактор ed или архиватор tar), более или менее обширный набор функционально связанных программ (скажем, coreutils) или огромный программный комплекс (примером чему XFree86 или Xorg).

Следует оговориться, что термин "пакет" (английское package) постоянно употребляется в двух смыслах: как набор исходных текстов и как комплект скомпилированных из него программ и всех их служебных файлов. Обычно различие между ними ясно из контекста, в случаях же неоднозначности тот или иной смысл будет оговариваться явно.

Пакеты принято распространять в виде компрессированных архивов: файлов вида *.tar.gz (*.tgz) или *.tar.bz2 (*.tbz2, *.tbz), так называемых тарбаллов. Обычно действует правило: один тарбалл — один пакет. Очень большие пакеты могут быть поделены на несколько тарбаллов (те же XFree86 или Xorg), но делается это исключительно для удобства скачивания, все равно такой набор тарбаллов исходников сохраняет свою целостность.

Прекомпилированные пакеты подчас также распространяются разработчиками в виде тарбаллов. Так, XFree86, кроме исходников, доступен также в виде серии скомпилированных пакетов для нескольких дистрибутивов Linux, Free- и OpenBSD. Но это уже самостоятельные пакеты, и не все они обязательны к установке. А сборщики дистрибутивов могут и далее дробить изначально единый пакет.

И еще. В предыдущих главах я говорил, что BSD-системы, в отличие от Linux, не имеют пакетной организации. Это не совсем точно. Конечно, например, из FreeBSD Distributions нельзя выделить ядро системы или наборы базовых утилит в виде отдельных пакетов. Однако сам по себе он, в сущности, единый пакет, только очень большой. А то, что перед пользователем (на диске или FTP-сервере) он предстает в виде кучи мелких (по

1,44 Мбайт) пакетиков — просто наследие тех времен, когда системы еще устанавливались с дискет. В NetBSD и OpenBSD базовая система собрана в виде единого тарбалла (base.tgz) и уже совсем ничем не отличается от обычных пакетов.

В последнее время и в некоторых дистрибутивах Linux прослеживается тенденция отказа от квантования базовой системы. Так, в Gentoo она вся собрана в три тарбалла (stage1, stage2, stage3) и может быть развернута (с различной полнотой, в зависимости от схемы инсталляции) из любого из них. А в Sorcerer и его клонах базовый тарбалл вообще единственный.

Вернемся к нашим исходниками и посмотрим, что с ними нужно делать: ведь ясно, что в том виде, в каком они распространяются, использование их невозможно.

Для того чтобы программа, распространяемая в исходниках, могла выполнять свои функции, она должна быть собрана. Процесс этот в общем случае разбивается на три стадии:

- конфигурирование;
- собственно сборку;
- инсталляцию.

Конфигурирование — это приведение программы в соответствие с реалиями конкретной системы. Как неоднократно говорилось, подавляющее большинство свободного софта пишется в расчете на некую абстрактную POSIX-совместимую ОС. Конкретные же их представители отличаются друг от друга многими деталями, в частности, функциональностью библиотек, их названиями и расположением. Что и проверяется в процессе конфигурирования. То есть основное назначение этого процесса — проверка так называемых зависимостей пакетов.

Понятие зависимостей — одно из основных при сборке программ. Суть его в том, что пакет pkgname1 для установки и функционирования требует наличия в системе пакета pkgname2; тот, в свою очередь, может потребовать пакета pkgname3 и т. д. В качестве зависимостей выступают часто (хотя и не всегда) те самые системные библиотеки, о которых говорилось ранее.

Зависимости пакетов бывают разными. С одной стороны, различают зависимости при сборке (depends) и зависимости при запуске (run depends).

В большинстве случаев depends и run depends эквивалентны, однако это правило имеет многочисленные исключения. Так, при статической сборке библиотеки, которые использует данный пакет, требуются только в момент компиляции, в дальнейшем необходимости в них не возникает.

С другой стороны, следует различать зависимости жесткие и мягкие. Удовлетворение первых абсолютно необходимо для сборки данного пакета. Так, практически любая программа использует (статически или динамически) главную системную библиотеку glibc (или libc), любое приложение для системы X: главную Иксовую библиотеку xlib, все приложения для интегрированной среды KDE — библиотеки qt и kdelibs.

Мягкие зависимости данного пакета не критичны для его функционирования, удовлетворение их лишь добавляет ему дополнительные функции (которые могут оказаться и лишними).

Понятие зависимостей пронизывает насквозь POSIX-совместимые системы и особенно важно для свободных их представителей. В то же время пользователи Windows с ним сталкиваются очень редко, и потому постижение его вызывает определенные трудности у недавнего "подоконника". Это связано с двумя факторами.

Во-первых, традиционная модель разработки UNIX-программ характеризуется ярко выраженным стремлением не множить сущности без крайней необходимости. Или, говоря попросту, не изобретать велосипеды. Если требуемая разработчику данной программы функция уже реализована и включена в какую-либо распространенную библиотеку, то наш разработчик этой библиотекой и воспользуется, а не будет переписывать требующийся код с нуля. Благо, поскольку все распространенные библиотеки открыты, он имеет полную возможность это сделать.

За конфигурирование обычно отвечает сценарий, расположенный в корне дерева исходников данной программы и, по соглашению, носящий имя `configure`. Как минимум, он обязан проверять жесткие зависимости устанавливаемого пакета и при их нарушении выдавать соответствующие сообщения. Кроме того, он может обеспечивать также подключение дополнительных функций — при наличии определенных условий, то есть удовлетворении зависимостей "мягких".

Так, для консольных программ в Linux существует возможность использования мыши в качестве позиционирующего устройства. Обеспечивается эта функция специальным сервисом gpm. И во многих программах (таких как файловый менеджер Midnight Commander или текстовый браузер links) конфигурационный скрипт проверяет, установлен ли пакет gpm, и при наличии его автоматически задействует использование мыши.

Если процесс конфигурирования завершается успешно, в корне дерева каталогов создается специальный файл Makefile, в котором и фиксируются все предусмотренные разработчиком настройки, выступающие в качестве директив на следующем этапе.

Если конфигурирование прошло с ошибками, выдается соответствующее сообщение, форма которого также целиком определяется разработчиком. Ошибки эти могут быть связаны с нарушением жестких зависимостей пакета, и в этом случае никакие дальнейшие действия до их разрешения невозможны. Если же конфигурационный сценарий выявил нарушение мягких зависимостей, то пользователь обычно может отказаться от них, просто потеряв некоторую дополнительную функциональность. Так, например, я всегда отказываюсь от поддержки мыши (через gpm) в консольных Linux-программах. Правда, это может потребовать указания некоторых опций конфигурирования (о чем я скажу чуть позже).

Образцово-показательный отчет о выполнении сценария `configure` выдают, по моему мнению, пакеты, штатно входящие в состав интегрированной среды KDE. Во-первых, их конфигурирование не обрывается сразу же после нахождения первой ошибки (первого нарушения зависимостей, например, отсутствия каких-либо мультимедийных или графических библиотек), как это бывает в большинстве других программ, а в любом случае доводится до конца. После этого сообщается, что такие-то компоненты необходимы для сборки данного пакета (то есть связаны с ним жесткими зависимостями), другие же требуются для получения определенных функций (например, наличие пакета cups — для обеспечения печати на принтере, пакета sane — для сканирования и т. д.). И пользователю вольно решить, устанавливать ли ему пакеты, или он, за отсутствием сканера или принтера, вполне может обойтись без них.

Наконец, конфигурирование завершилось успешно. Наступает время следующего этапа — собственно сборки, то есть претворения исходных текстов программы в исполняемый машинный код. Этап этот распадается на несколько стадий.

Первая стадия — собственно компиляция исходного текста в бинарный код, завершающаяся формированием объектного модуля. Это, как правило, еще не готовая к запуску программа. Почему? Да потому, что в его скомпилированном коде может не быть многих стандартных функций, тех самых, которые разработчик предполагал заимствовать из разделяемых библиотек.

И потому вторая стадия — это связывание (linking) сгенерированного кода с необходимыми библиотечными фрагментами. Линковка может быть двух видов: статическая и динамическая. В первом случае требуемый код из библиотеки встраивается внутрь собираемой программы, после чего получается готовый к исполнению бинарный файл, более в библиотеке не нуждающийся. Это именно тот случай, когда понятия depends и run depends приобретают разное значение: первое оказывается шире.

Второй случай (динамической линковки) предполагает, что библиотечный код не встраивается в программу: вместо него устанавливается только ссылка на файл библиотеки и требуемую функцию (имя которой получается из заголовочного файла — header-файла). И в дальнейшем при запуске исполняемого модуля программы соответствующие библиотечные фрагменты извлекаются из библиотеки и присоединяются к коду программы уже только в оперативной памяти. При этом сущности depends и run depends совпадают: библиотека, с которой программа связывается при сборке, столь же необходима и для ее запуска.

Динамическая линковка приводит как к сокращению размера исполняемого файла, так и уменьшению объема оперативной памяти при запуске программ (особенно если они используют одни и те же библиотечные функции, а в большинстве случаев так оно и есть). И потому именно она преимущественно используется при сборке программ для свободных POSIX-систем (повторю еще раз, что разделяемые библиотеки в них открыты и могут применяться без ограничений).

Однако бывают ситуации, когда приходится прибегать к линковке статической. Так, во FreeBSD статически линкуются с главной системной библиотекой жизненно важные для запуска и восстановления системы утилиты (в предыдущих версиях этой ОС они располагались в каталогах /bin и /sbin, во FreeBSD, начиная с пятой ветки, для них отведен специальный каталог /restore). В результате они оказываются доступными даже в случае аварийной загрузки, когда все файловые системы, кроме корневой, не монтируются.

Третий этап процесса сборки — инсталляция. Это включение всех компонентов программы в структуру файловой системы данной машины, копирование в соответствующие каталоги файлового древа. Как правило, для разных компонентов пакета существуют традиционно предопределенные имена каталогов, в которых они должны размещаться: bin или sbin — для исполняемых модулей, lib — для библиотек, etc — для конфигурационных файлов, share — для всякого рода документации и примеров.

Предопределенные имена каталогов не обязательно будут ветвями корня файловой системы (типа /bin, /sbin и т. д.): более вероятно, что соответствующие компоненты собираемого пакета помещаются в каталоги /usr/bin, /usr/local/bin. Впрочем, к обсуждению этого вопроса мы скоро вернемся.

Так вот, процесс инсталляции и сводится к тому, что исполняемый файл (файлы) собранного пакета копируется в файл ~/bin (или, для программ системного назначения, в ~/sbin), ее конфиг в ~/etc, страницы документации в ~/man (~/ в данном случае символизирует не домашний каталог пользователя, а некий условный префикс).

По завершении всего сборочного цикла из пакета исходников должна получиться полнофункциональная, готовая к употреблению программа, требующая лишь некоторой пользовательской настройки.

Три волшебных слова

Теперь, разобравшись с принципами, посмотрим, как сборка пакетов осуществляется на практике.

Понятное дело, что тарбалл исходников следует декомпрессировать и развернуть в подходящем каталоге. Для самостоятельно собираемых исходников я использую обычно каталоги вроде $HOME/src или /usr/local/src (разумеется, для этого нужно обладать правами на запись в тот или иной). Как станет ясным из дальнейшего, удаление развернутого дерева исходников установленных программ очень нежелательно, поэтому следует озаботиться наличием достаточного количества свободного места в той файловой системе, в которой выполняется распаковка.

Сама по себе распаковка делается обычным образом, например, командой `tar`:

```
$ tar xzpvf /path_to_src/tarball.tar.gz
```

или

```
$ tar xjpvf /path_to_src/tarball.tar.bz2
```

в зависимости от использовавшейся для компрессии программы. Кратко остановлюсь на смысле опций (команда `tar` будет предметом отдельного рассмотрения).

Опция `x` (от extract) предписывает развертывание архива. Однако поскольку он был ранее сжат утилитой компрессии, его предварительно нужно декомпрессировать, этому служит опция `z` при команде `gzip` или `j` при `bzip2`. Опция `f` имеет своим значением имя подвергающегося развертыванию/декомпрессии файла, в примере это `tarball.tar.*`. Опция `v` не обязательна, она заставляет выводить на экран сообщения о ходе распаковки.

А вот опция `p` может быть важной: она предписывает сохранять атрибуты доступа и принадлежности теми же, что были у оригинальных файлов до их упаковки в тарбалл. Без нее хозяином всех распакованных файлов оказался бы пользователь, выполняющий процедуру распаковки. Обычно это не имеет значения, но в некоторых случаях нежелательно или просто не должно быть (например, при распаковке прекомпилированных тарбаллов stage1–3 в Gentoo). Так что лучше взять себе за правило не забывать про эту опцию никогда.

Если пакет состоит из нескольких тарбаллов, все они должны быть распакованы. Повторять несколько раз какую-либо из приведенных команд было бы скучно, однако эту процедуру можно выполнить в один присест. Самый простой способ для этого — прибегнуть к универсальной утилите `find`, что в данном случае будет выглядеть примерно так:

```
$ find /path_to_src -name *.tar.gz -exec tar xzpvf {} \;
```

В результате любой из описанных процедур в текущем каталоге должен образоваться подкаталог вида package_name-version, то есть соответствующий имени пакета с указанием номера его версии. Но это только в том случае, если исходный тарбалл был сформирован корректно, с включением корня дерева исходников. Редко, но бывает так, что разработчик забывает о такой мелочи. И потому, дабы не получить в текущем каталоге неудобопонятной мешанины файлов, перед собственно распаковкой лучше выполнить проверку командой

```
$ tar -tzvf tarball.tar.gz
```

где опция `t` (от list) и предписывает вывести список файлов тарбалла вместо его развертывания.

Дальнейшие действия по сборке пакета в большинстве случаев осуществляются путем последовательной отдачи трех команд:

- `./configure`;
- `make`;
- `make install`.

Первую из этих трех команд следует давать, перейдя предварительно в корень дерева исходников нужного пакета:

```
$ cd /path2srcpkg
$ ./configure
```

Это запуск того самого конфигурационного скрипта, о котором давеча говорилось. Обращаю внимание на `./`, эти символы являются указателями на текущий каталог (/path_to_srcpkg), в котором расположен файл сценария (а мы помним, что текущий каталог, как правило, не включается в число значений переменной `$PATH`).

Сценарий `configure` имеет некоторое количество опций. Число их и назначение определяются разработчиком, однако некоторые встречаются практически всегда. И важнейшая из них опция `--help` выводит полный список всех других опций. Строго говоря, именно с команды

```
$ ./configure --help
```

и следует начинать самостоятельную сборку любого пакета, особенно незнакомого. Прочитав предварительно файлы README и INSTALL (минимум

один их таковых, скорее всего, имеется в корне дерева исходников), которые содержат более или менее подробную информацию о программе, в том числе и о порядке ее сборки. Однако полного перечня опций конфигурирования там, скорее всего, не будет, так что ознакомимся с наиболее обычными из них посредством указанной команды.

В аккуратно написанных программах вывод команды

```
./configure --help
```

обычно распадается на несколько секций. Первой, как правило, идет секция

```
Installation directories:
```

В ней указываются каталоги, куда в дальнейшем будут устанавливаться отдельные компоненты собранного пакета. Важнейшей опцией является

```
--prefix=PREFIX.
```

Значением `PREFIX` будет выступать ветвь корневого каталога, в подкаталоги которого запишутся исполняемые файлы, конфиги, библиотеки и т. д. По умолчанию эта опция в большинстве случаев имеет значение `/usr/local`. То есть в случае, если значение опции `--prefix` при запуске скрипта `./configure` не задано, то исполняемые файлы пакета будут инсталлированы в /usr/local/bin, конфиги — в /usr/local/etc и т. д.

Так что если желательно размещение компонентов собираемого пакета в другом каталоге (например, в /usr), значение префикса следует задать в явном виде, скажем, так:

```
./configure --prefix=/usr
```

В последнее время некоторые пакеты предполагают установку по умолчанию в подкаталоги каталога /opt (/opt/pkgname/bin, /opt/pkgname/lib). А для KDE-приложений последних версий это обычно /opt/kde (/opt/kde/bin, /opt/kde/lib). Для таких программных комплексов лучше его не менять во избежание осложнений при поиске библиотек.

А вообще, значение опции `--prefix` может быть любым. В частности, если предполагается сборка пакета для дальнейшего автономного его распространения в бинарном виде, целесообразно сосредоточить все его компоненты в отдельном подкаталоге, например, вида $HOME/my_pkg/pkg_name. Аналогично следует поступать и при тестировании пакета, предшествующем его инсталляции.

Далее в той же секции обычно имеет место быть опция

```
--bindir=DIR.
```

И здесь значением `DIR` выступает обычно `PREFIX/bin`. Однако в некоторых случаях исполняемые файлы пакета целесообразно поместить в иные ветви файловой системы. Например, если вручную собирается командная оболоч-

ка, которая будет выступать в дальнейшем как login shell, или любимый (общесистемный) текстовый редактор, очень желательно, чтобы их исполняемые бинарные файлы находились непосредственно в каталоге корневой файловой системы (иначе они могут быть недоступны в аварийных случаях или при старте в однопользовательском режиме). И тут, вне зависимости от того, задано ли значение опции `--prefix` или нет, конфигурационный скрипт следует запускать в такой форме

```
$ ./configure --bindir=/bin
```

или, для программ административного назначения:

```
$ ./configure --bindir=/sbin
```

Нередко в секции

```
Installation directories:
```

можно обнаружить и другие опции, предписывающие размещение библиотек, страниц документации и тому подобных компонентов пакета.

Следующая почти непременная секция вывода помощи конфигурационного скрипта

```
Optional Features:
```

Именно в ней, как легко понять из названия, перечисляются опции, позволяющие подключить/отключить дополнительные возможности собираемого пакета. Они имеют вид

```
--enable-FEATURE
```

и

```
--disable-FEATURE
```

Первая, естественно, подключает возможность с именем `FEATURE`, а вторая отключает. Причем одна из этих опций может быть принята по умолчанию. Например, большинство свободных программ ныне собирается с поддержкой национальных языков (National Language Support — NLS), обеспечивающих вывод сообщений, пунктов меню, помощи на языках, отличных от английского. Однако в ряде случаев это может показаться нежелательным, и тогда при конфигурировании программы нужно задать соответствующую опцию:

```
$ ./configure --disable-nls
```

Обычно допустима и иная форма этой опции:

```
$ ./configure --enable-nls=no
```

Функции, подключаемые (или отключаемые) посредством опции

```
--enable(disable)-FEATURE
```

берутся из библиотечных пакетов. В частности, за поддержку NLS отвечает библиотека gettext (подчеркну, что сама по себе эта библиотека не дает возможности волшебным образом получать сообщения на русском или там эскимосском языке, а только обеспечивает принципиальную возможность вывода таковых).

Сходный смысл имеют опции, общий вид которых таков:

```
with-PACKAGE
```

или

```
without-PACKAGE
```

Отличие опций этой секции в том, что в качестве значений `PACKAGE` выступают не некие абстрактные функции, а имена конкретных пакетов, возможности которых добавляются к собираемому (или отнимаются от него).

Опции вида `with-PACKAGE` могут также иметь значения: `yes`, `no` или `auto`. Последняя обычно принята по умолчанию. То есть, если в ходе выполнения конфигурационного скрипта пакет с именем PACKAGE будет обнаружен в данной системе, его возможности будут подключены автоматически, если нет — проигнорированы.

Так, в приводимом ранее примере с поддержкой позиционирующих свойств мыши при сборке консольных программ типа links или mc они будут автоматически задействованы по умолчанию, если в Linux-системе (к BSD это не относится) обнаружится установленный пакет gpm. Если же он имеется, но поддержка мыши для данного пакета представляется нежелательной, это следует указать в явном виде:

```
$ ./configure --without-gpm
```

Последняя секция, практически всегда присутствующая в выводе помощи конфигурационного скрипта:

```
Some influential environment variables:
```

Как следует из названия, здесь перечисляются различные переменные окружения, способные оказывать влияние на процесс компиляции. Наиболее часто в качестве таких переменных предусматриваются флаги компилятора gcc типа `CFLAGS` и `CXXFLAGS` (для программ на языке Си и Си++ соответственно). Обычное употребление таких флагов — задание всякого рода оптимизаций: общего ее уровня, архитектуры целевого процессора, конкретных наборов его команд и т. д. Например, задание опции

```
$ ./configure CFLAGS="-O3 -march=pentium4"
```

обеспечит максимальный уровень оптимизации (значение `-O3`) для процессора Pentium 4 (значение `-march=pentium4`). Опять же, заострим внимание на кавычках, в которые эти значения заключены (дабы восприниматься как

единый аргумент). Впрочем, тема оптимизации будет предметом особого рассмотрения.

Ранее я говорил, что при сборке пакеты могут связываться с библиотечными функциями как статически, так и динамически. Так вот, характер связи также определяется на стадии начального конфигурирования. По умолчанию во всех известных мне случаях используется динамическая линковка. Чтоб связать исполняемый модуль с библиотекой статически, требуется специальный флаг `LDFLAGS`. Значениями его в данном случае будут

```
$ ./configure LDFLAGS="-static"
```

или, в случае линковки с несколькими библиотеками,

```
$ ./configure LDFLAGS="-all-static"
```

Вообще говоря, опции конфигурирования пакета могут быть очень разнообразны, я перечислил лишь наиболее часто встречающиеся. Общие рецепты эффективного их использования: внимательное чтение вывода помощи конфигурационного скрипта, знание особенностей своей собственной системы (в первую очередь — мест размещения разделяемых библиотек) и, конечно же, здравый смысл.

Надо отметить, что некоторые пакеты не имеют конфигурационного сценария в дереве исходников. Это не обязательно следствие лени разработчика: может быть, что программа настолько проста, что в предварительном конфигурировании не нуждается (например, не использует функций никакой разделяемой библиотеки).

Другая возможная причина отсутствия скрипта `configure` — предварительное конфигурирование уже выполнено разработчиком. В этом случае в дереве исходников можно обнаружить уже готовый Makefile, рассчитанный на некоторые типовые ситуации, или несколько его вариантов для разных архитектур, операционных систем и т. д. Если же ни один из предложенных автором вариантов не отвечает в полной мере реалиям пользователя, у него остается последний выход: ручная правка Makefile (обычно такие случаи документируются в файлах типа README или INSTALL).

Предварительное конфигурирование пакета — очень важный момент в его сборке: можно сказать, что успех ее на 90% определяется именно в результате исполнения скрипта `configure`. Однако рано или поздно оно завершается удачно (о случаях фатального невезения я скажу несколько позже). И наступает время собственно сборки, для чего предназначено второе из наших магических заклинаний — команда `make`.

Сама по себе команда `make` не выполняет ни компиляции (это дело компилятора gcc), ни линковки (с этой ролью справляется редактор связей ld), ни иных действий по превращению исходного текста в машинный код. Задача

ее — интеграция всех требуемых средств (а в процессе сборки могут задействоваться и препроцессоры, и языковые анализаторы, и иные инструменты), чтобы автоматически получить (почти) готовые к употреблению бинарные компоненты пакета. Собственно говоря, от пользователя требуется только дать директивное указание — набрать в командной строке `make` и нажать <Enter>, все остальное произойдет как бы само собой. С другой стороны, у него нет и возможности вмешаться в процесс сборки (разве что прервать его комбинацией клавиш <Ctrl>+<C>).

Конечно, и команда `make` способна воспринимать многие параметры командной строки. Правда, в большинстве случаев они дублируют опции, заданные при конфигурировании. Так, при отдаче директивы `make` можно задать флаги оптимизации

```
$ make CFLAGS="-O3 -march=pentium4"
```

и предписать статическую линковку с разделяемыми библиотеками

```
$ make LDFLAGS="-static"
```

или предписать последующую установку компонентов пакета в другой каталог. Далее остается только дожидаться успешного (надеюсь) окончания сборки.

У команды `make` есть еще один важный вид аргументов командной строки — цели (target). Собственно для сборки по умолчанию они обычно не требуются. Хотя некоторые пакеты требуют их задания в явном виде. Так, оконная система X штатным образом собирается с указанием цели `world`:

```
$ make world
```

В других случаях для достижения того же результата может применяться и иная цель, например,

```
$ make all
```

или указание на конкретную архитектуру, операционную систему и т. д. Все такого рода исключения, как правило, описаны в сопроводительной документации. На худой конец, допустимые для данного пакета цели команды `make` можно подсмотреть в Makefile, там они будут определены обязательно.

Однако есть у команды `make` и практически непременная цель `install`, предписывающая выполнить установку компонентов пакета. И это третье (и последнее) из наших шаманских заклинаний:

```
$ make install
```

После него мы, наконец, получаем готовую к употреблению программу.

В некоторых программах цель `install` разработчиком не предусматривается. И тут приходится вручную скопировать скомпилированные модули в подходящие каталоги. Правда, такое бывает очень редко и, как правило, для очень простых по устройству программ.

В принципе команду `make install` можно было бы дать и сразу по исполнении сценария `./configure`. В этом случае сначала будет исполнена цель `make` — компиляция и линковка пакета, а затем его инсталляция. Однако делать это не всегда целесообразно. Во-первых, такое совмещение целей затрудняет отслеживание ошибок. Во-вторых, их совместному исполнению может помешать отсутствие должных прав доступа.

Дело в том, что стадии конфигурирования и сборки обычно могут быть выполнены от имени обычного пользователя, это определяется правами доступа к каталогу, в который было распаковано дерево исходных текстов пакета. А вот установка собранных компонентов почти наверняка потребует административных привилегий. Ведь исполняемые файлы пакета после директивы `make install` копируются (если придерживаться схемы) в каталог /usr/local/bin, документация — в /usr/local/share и т. д. А все они закрыты для изменения кем бы то ни было, кроме root.

Так что прежде чем начинать установку собранного пакета, следует озаботиться получением соответствующих полномочий командой

```
$ su
```

или в некоторых случаях даже

```
$ su
```

которая приведет среду исполнения команды в соответствие с конфигурацией суперпользователя.

Впрочем, иногда полномочия администратора могут оказаться необходимыми и при сборке программы или ее конфигурировании. Типичный тому пример — сборка ядра Linux штатными средствами, в ходе которой действуют скрипты, требующие root-доступа. Впрочем, это тема отдельной беседы.

Все сказанное относилось к случаю сборки без ошибок на любом этапе. От них, однако же, никто не гарантирован. И что делать, если ошибки появляются?

В случае ошибки при сборке пакета перед пользователем, как обычно, появляются два выхода:

- бросить это занятие, попробовав отыскать и установить прекомпилированный вариант пакета;
- разобраться в причинах ошибки и попытаться ее устранить.

Наиболее часто сообщение об ошибке возникает в ходе предварительного конфигурирования. И в большинстве случаев связано с нарушением зависимостей, жестких или мягких, обычно это можно понять, внимательно читая экранный вывод.

С нарушением жестких зависимостей все ясно: нужно установить пакет, от которого зависит собираемый. Нарушения же мягких, но принятых разра-

ботчиком по умолчанию зависимостей обычно можно избежать посредством явного указания опций конфигурирования (типа `disable-FEATURE` и `--without-PACKAGE`).

Встречаются и ситуации кажущегося нарушения зависимостей, когда в ходе конфигурирования следует сообщение об отсутствии пакета, хотя пользователь точно знает, что таковой был установлен. В одних случаях это может быть связано с тем, что собираемый пакет ссылается не просто на некую библиотеку, но на конкретную ее версию. При этом даже если в системе установлена более новая ее реализация, заведомо перекрывающая функциональность предыдущей (а совместимость сверху вниз — один из краеугольных камней программирования вообще и POSIX-программирования в особенности), в ходе конфигурирования будет отмечено нарушение зависимостей. Разрешение такой коллизии очень простое: создание символической ссылки вида

```
$ ln -s libname.xxx libname.yyy
```

где `yyy` — номер старой версии библиотеки, а `xxx` — актуальный ее вариант.

Другой случай, когда сценарий конфигурирования пакета ищет библиотеку, от которой он зависит, не в том каталоге, где она реально располагается. Так, старые приложения KDE могут ожидать требуемых им библиотек в каталогах типа /usr/local/qt и /usr/local/kde, тогда как ныне они, скорее всего, будут располагаться в ветвях каталога /opt.

И тут выход из положения прост. Во-первых, можно задать переменную окружения

```
LDPATH="/opt/qt:/opt/kde"
```

значения которой точно определяют каталоги соответствующих программных комплексов.

Во-вторых, эти значения можно просто указать в качестве опций конфигурационного скрипта:

```
$ ./configure --with-qt-dir=/opt/qt \
       --with-kde-dir=/opt/kde
```

Реже бывают ошибки при исполнении команды `make`. Однако бороться с ними труднее. Общий рецепт тут дать очень трудно. Можно только посоветовать читать вывод сообщений о ходе компиляции, непосредственно предшествующих ее обрыву.

Ошибки при инсталляции связаны почти всегда с отсутствием прав на запись в каталоги, в которые помещаются устанавливаемые компоненты пакета. Иногда же они возникают вследствие того, что целевой каталог просто не существует. Например, в таких дистрибутивах Linux, как CRUX и

Archlinux, из пакетов штатного комплекта изъята вся документация, кроме man-страниц. И, соответственно, отсутствуют каталоги для помещения документации в форматах info и html. А поскольку практически любая программа проекта GNU сопровождается info-документацией, попытка инсталляции ее вызывает ошибку. Побороть ее очень просто: нужно только уничтожить в дереве исходников соответствующие подкаталоги.

На какой бы стадии ни возникла ошибка, перед повторной сборкой пакета дерево исходников следует очистить от побочных продуктов сборки предыдущей. Резонные люди рекомендуют просто стереть каталог с исходными текстами и развернуть его из тарбалла заново. Это, конечно, самый радикальный способ, но не всегда приемлемый. Обычно можно обойтись и терапевтическими мерами. Потому что в большинстве пакетов предусматриваются специальные цели для таких процедур. Первая из них:

```
$ make clean
```

Она вычистит дерево исходников от объектных модулей, собранных при предыдущей компиляции, сохранив, тем не менее, созданные при конфигурировании Makefile. Если нужно избавиться и от них, существует цель

```
S make distclean
```

По исполнении команды дерево исходников должно приобрести первозданный вид.

Пакеты, как правило, устанавливаются для того, чтобы запускать входящие в них программы. Однако не исключено, что первый же запуск новой программы показывает, что она делает что-то не то или не так, нежели это нужно пользователю. Или просто ему не нравится. И возникает вопрос: а как удалить такой неподходящий пакет?

Можно, конечно, последовательно пройтись по всем каталогам, в которые записывались компоненты пакета, выявить их по каким-либо признакам (например, по атрибуту `ctime`, в данном случае отвечающему времени создания файла) и удалить вручную. Однако способ этот трудоемкий и чреват ошибками.

К счастью, большинство разработчиков, не страдающих манией величия, предусматривают такую ситуацию, определяя специальную цель для удаления программы. Обычно это

```
$ make uninstall
```

и реже

```
$ make deinstall
```

Любую из этих команд нужно дать в корне дерева исходников, именно поэтому его желательно сохранять и после сборки, несмотря на непроизводительный расход дискового пространства, иначе единственным способом

удаления пакета останется ручной. При деинсталляции команда `make` отыскивает установленные компоненты пакета в дереве файловой системы и удаляет их.

Важно, что `make uninstall` не затрагивает пользовательских настроечных файлов — dot-файлов в домашнем каталоге, которые часто генерируются автоматически при первом запуске программы. Такие файлы при необходимости в любом случае придется удалять вручную. Казалось бы, неудобство, однако сейчас мы увидим, что это не лишено резонов.

Дело в том, что самостоятельная сборка пакетов из исходников не предусматривает никакого механизма обновления их версий. Правда, на самом деле обладателю настольной машины обычно нет надобности гнаться за самыми актуальными версиями большинства пользовательских программ: это целесообразно делать только при существенном расширении их функциональности или обнаружении серьезных ошибок. Однако администратор системы вынужден отслеживать все обновления пакетов, связанные исправлением ошибок в безопасности системы. А поскольку, как я все время повторяю, каждый пользователь — это немного администратор своего десктопа, необходимость в обновлении версий возникает достаточно часто.

Так вот, единственный способ обновления собственноручно собранной программы — это собрать и инсталлировать ее заново. При этом старые компоненты должны затереться новыми. Однако на практике от старой версии могу остаться хвосты, не только бесполезные, но, возможно, и конфликтующие с файлами новой версии.

И потому при обновлении версий вручную обычно рекомендуется сначала удалить старую версию и лишь затем инсталлировать новую. Предварительно убедившись, конечно, в успешности сборки (то есть выполнив стадии `./configure` и `make`, это еще одна причина для обособления цели `make install`) и работоспособности (для этого можно пробно запустить исполняемый файл пакета прямо из каталога исходников).

В то же время при смене версий, как правило, желательно сохранить все выполненные ранее настройки пакета, подчас это весьма трудоемкая процедура. И вот тут-то и оказывается, что пользовательские настройки удаленной версии остались в неприкосновенности в домашнем каталоге.

Особенности сборки ядра

Ядро свободной POSIX-системы — это (почти) такая же программа, как и любая другая, распространяемая в исходных текстах. И потому оно также должно быть приведено в пригодное к употреблению состояние посредством

сборки. Однако ядро все же не пользовательское приложение и даже не системная утилита, и потому процесс сборки ядра имеет свою специфику.

Кроме того, ядро — это тот компонент, который и отличает одну POSIX-систему от другой, например, Linux от FreeBSD. Тогда как все уже сказанное о сборке пакетов имеет силу (с очень незначительными оговорками) для любой POSIX-системы, причем даже не обязательно свободной: открытые приложения и утилиты для проприетарной Solaris или AIX в принципе собираются точно так же, как для Linux или какой-либо BSD.

Тем не менее, сборка ядра любой из рассматриваемых ОС включает те же стадии, что и сборка иных приложений: предварительное конфигурирование, собственно сборку и инсталляцию.

Конфигурирование ядра — это включение или выключение поддержки различных устройств (драйверов, хотя здесь это понятие существенно отличается от принятого в Windows), файловых систем, сетевых протоколов. Включение поддержки какой-либо опции подразумевает, что в ядро будет встроен код, обеспечивающий работу с неким устройством или файловой системой. Кроме того, многие опции могут быть включены как модули. То есть соответствующий им код компилируется в бинарный вид, но непосредственно в ядро не встраивается, а подгружается в виде отдельной программы по мере необходимости: вручную, соответствующими командами, или автоматически.

Заметим, что, когда речь заходит о драйверах устройств, распространяемых отдельно от ядра операционной системы (например, производителями оборудования), имеются в виду именно загружаемые модули ядра. Подобно другим программам, они могут существовать в виде исходников или в бинарном виде. В первом случае их теоретически можно собрать для любой версии ядра (или, по крайней мере, для диапазона близких версий), разумеется, только данной ОС (драйверы для Linux, как можно догадаться, не будут работать во FreeBSD, и наоборот). Бинарные же, прекомпилированные драйверы обычно жестко привязаны к версии ядра, хотя иногда могут както работать и при смене оной.

Процесс конфигурирования является наиболее ОС-специфичным во всей процедуре сборки ядра. Само собой разумеется, что ядра разных систем имеют разные функции и, соответственно, разные опции конфигурирования (еще бы, ведь это все-таки разные программы). Но даже внешне, на пользовательском уровне, процесс конфигурирования ядра Linux и, скажем, FreeBSD (или DragonFlyBSD) существенно отличаются.

Во FreeBSD традиционным инструментом конфигурирования ядра выступает обычный текстовый редактор. С его помощью конфиг умолчального ядра (ядра `GENERIC`, устанавливаемого при инсталляции системы) приводится в

соответствие с потребностями пользователя: одни опции отключаются, другие включаются.

С отключаемыми опциями все понятно, они просто изымаются из конфига (путем установки символа комментария на соответствующих строках). А вот откуда взять опции недостающие? Они отыскиваются в некоем образцово-показательном конфигурационном файле. Во FreeBSD четвертой ветки (и это унаследовано в DragonFlyBSD) такой файл носил имя LINT (для конфигов ядра принято употреблять символы верхнего регистра, этим подчеркивается величие сей программы) и был похож на настоящий, хотя и не работал (скомпилировать из него работоспособное ядро было невозможно). В пятой ветке на смену ему пришел файл NOTES — уже без претензий, это просто список всех теоретически доступных опций конфигурирования, снабженных достаточно подробными комментариями.

Специального включения/выключения модулей в конфигурационном файле ядра FreeBSD не предусмотрено, в качестве таковых по умолчанию собираются все выключенные опции, для которых модульная поддержка в принципе возможна (а она реализована еще не для всех опций). Правда, это положение можно изменить правкой соответствующих конфигурационных файлов.

В первозданном (или каноническом, что на **http://www.kernel.org**) ядре Linux никакого конфигурационного файла по умолчанию не предусмотрено: он генерируется при первом же запуске штатного инструмента для ядерной настройки, базируемого все на той же утилите `make`. Для изначального конфигурирования ядра предусмотрено четыре цели: `make config`, `make menuconfig`, `make xconfig` и `make gconfig`. Все они делают одно дело, но каждая по-своему.

- Команда `make config` вызывает текстовый конфигуратор ядра, работающий в диалоговом режиме. То есть он требует ответа на множество вопросов, для которых предусмотрены варианты ответов `Yes` и `No`, а для многих опций — еще и `M` (Module), который и обеспечивает подключение модульной поддержки.
- Использование `make config` представляется не очень удобным: в случае малейшей ошибки имеется только одна возможность для исправления — оборвать программу (например, через <Ctrl>+<C>) и начать все заново. И вообще, этот метод конфигурирования считается устаревшим, и в ядрах ветки 2.6.X цель `make config` штатно не документирована. Хотя и может использоваться при желании, ее преимущество (по-моему, несколько сомнительное) в том, что, в отличие от прочих средств конфигурирования, она не требует прав суперпользователя.

- Команда `make menuconfig`, напротив, загружает весьма наглядный меню-ориентированный конфигуратор, богато оформленный псевдографикой. В каждом из подпунктов меню достаточно отметить нужные опции для их включения или, напротив, снять отметки для отключения. Опции, для которых реализована модульная поддержка, могут быть отмечены соответствующим образом.
- Использование `make menuconfig`, пожалуй, наиболее употребимый (и удобный) способ конфигурирования ядра Linux. Нужно только помнить, что запуск соответствующей команды требует полномочий root.

Команды `make xconfig` и `make gconfig` вызывают конфигураторы ядра, работающие в графическом режиме и, соответственно, могут быть запущены только в терминальном окне сеанса Иксов. В версиях ядра до 2.4.X включительно предусматривалась только первая из указанных целей, которая вызывала достаточно простую меню-ориентированную программу. В ядрах ветки 2.6.X цель `make xconfig` претерпела существенные изменения: теперь ею вызывается весьма "тяжелая", основанная на библиотеке Qt, программа, требующая также наличия в системе установленной среды KDE и пакета разработки `kdevelop`, что делает ее практически недоступной для многих пользователей. Ну, а `make gconfig` влечет загрузку аналогичного по функциональности конфигуратора, но основанного на библиотеке Gtk. По причине стойкой нелюбви к последней я этим методом никогда не пользовался и ничего сказать о нем не могу.

Есть в Linux и еще несколько целей, обеспечивающих конфигурирование ядра. Так, командой `make defconfig` можно сгенерировать некий ядерный конфиг, в него будут включены именно те опции и модули, которые отмечены по умолчанию в меню `make menuconfig`.

При смене версии ядра можно прибегнуть к цели `make oldconfig`. Она восстановит конфигурацию ядра прежнего, запросив ответы только на вопросы, касающиеся вновь появившихся опций.

И, наконец, почти полный список доступных для конфигурирования ядра и его сборки целей можно получить посредством

```
$ make help
```

Собственно сборка ядра и в Linux, и в BSD-системах осуществляется командой `make`, возможно, с указанием неких конкретных целей. Останавливаться на их описании я не буду. Замечу только, что в Linux дополнительно нужно собрать и модули командой

```
make modules,
```

тогда как во FreeBSD это произойдет по умолчанию одновременно с компиляцией ядра.

В результате по завершении сборки у нас в подкаталогах дерева исходников ядра образуется собственно образ ядра и набор скомпилированных модулей. Это такие же объектные файлы, которые возникают на промежуточной стадии компиляции других программ. Так как к самостоятельному использованию они не предназначены, то и в линковке они не нуждаются.

Файл образа ядра во FreeBSD всегда носит имя kernel. В Linux же существует несколько видов образов, за которыми традициями закреплены фиксированные имена (типа vmlinuz, bzImage).

Как и компоненты любого прикладного пакета, образ ядра и сопровождающие его модули должны быть инсталлированы в те подкаталоги, где система ожидает обнаружить их при загрузке. В Linux таких мест традиционно два: каталог /boot или, реже, корень файловой системы, тогда как для модулей предназначается каталог /lib/modules/X.Y.Z, где X.Y.Z — номер версии ядра (например, 2.6.7). Во FreeBSD, начиная с пятой ветки, под ядро и модули отведен каталог /boot/kernel. В DragonFlyBSD (и остальных BSD-системах) ядро по-прежнему размещается в корне файловой системы, модули же — в каталоге /modules.

Так вот, процесс инсталляции и сводится к тому, что файл образа ядра вместе с объектными модулями (а это ныне файлы вида *.ko — для отличия от обычных объектных файлов) копируется в надлежащие каталоги. Для этого предназначены специальные цели команды `make`. Во FreeBSD это `make install` или `make kernelinstall`. В Linux для установки модулей предназначена цель `make modules_install`, а само ядро может быть инсталлировано различными способами (в том числе и просто ручным копированием).

В *разделе о системных загрузчиках главы 13* мы видели, что в принципе ядро может быть размещено чуть ли не в любой части файловой системы. Нужно только позаботиться о том, чтобы программа, выступающая в качестве загрузчика, знала о его расположении. Однако нестандартное размещение ядра может создать некоторые проблемы и в любом случае потребует лишних телодвижений, так что лучше придерживаться традиционной схемы.

Вопросы оптимизации

Сказавши "а", логично было бы добавить "б". А именно: после пересборки ядра выполнить оптимизацию приложений под имеющееся "железо", по крайней мере, критически важных для производительности системы в данных условиях. Тем более что современные версии компилятора gcc предоставляют к тому много возможностей.

Выбор версии компилятора gcc вообще очень важен для целей оптимизации. До недавнего времени практически на равных существовали две его ветки:

2.95.X и 3.X. Причем преимущества второй были отнюдь не очевидны. Дело в том, что ветка 3.X на протяжении длительного времени развивалась в сторону улучшения генерации кода Си++, тогда как код, генерируемый из программ на чистом Си, по свидетельству знатоков, получался даже хуже, чем при использовании gcc-2.95.X. И потому эта версия рекомендовалась для сборки наиболее ответственных частей Linux- и BSD-систем, таких как ядро, главная системная библиотека и сам компилятор, не содержащие кода Си++. А это и есть первые претенденты на оптимизацию, не считая "тяжелых" приложений типа KDE или GNOME, а также мультимедийных и узкоспециальных программ. В BSD-системах gcc-2.95.X вообще использовался по умолчанию. Однако компилятор этой ветки не предусматривал возможности оптимизации под современные процессоры с их специализированными наборами инструкций (типа SSE и 3DNow!), ибо появился задолго до них.

Положение изменилось с выходом gcc-3.4.X, который первым из своей линии генерировал Си-код не хуже предшественника, а подчас и лучше. И потому использование его видится предпочтительным для всех компонентов любых систем. Оптимизация приложений достигается заданием соответствующих флагов, параметров компилятора gcc, определяемых при вызове. Первый среди них — уровень оптимизации, который задается флагом `-O#` и может варьировать от нулевого (флаг `-O0`, при котором никакой оптимизации не происходит) до максимального `-O3`. Большинство пакетных дистрибутивов Linux, декларирующих свою оптимизированную природу, собирается с флагом `-O2`. В BSD-семействе по умолчанию принята оптимизация уровня `-O1` для всех ее компонентов: ядра, базовой системы и портов; более того, до недавнего времени выполнить операцию `make world` (полную пересборку системы) при более высоких уровнях оптимизации было практически невозможно.

Выбор уровня оптимизации — не столь однозначный вопрос, как может показаться. И максимально высокий уровень `-O3` далеко не всегда дает лучшие результаты, а подчас производительность собранных с его использованием программ даже снижается. Не говоря уж о том, что некоторые программы вообще при этом уровне собираться отказываются. И в любом случае при уровне `-O3` размер исполняемого кода оказывается больше, нежели при `-O2`, что ведет к снижению скорости запуска приложений.

Вообще говоря, мои (и не только мои) наблюдения показывают, что самый большой скачок в быстродействии происходит при переходе от уровня `-O0` (то есть отказа от оптимизации) к `-O1`. Флаг `-O2` обычно обеспечивает лишь небольшой прирост скорости запуска и выполнения программ, а результаты применения флага `-O3` неоднозначны. В частности, эксперименты с пересборкой DragonFlyBSD показали стабильное (хотя и очень маленькое численно) отставание от уровня `-O2`.

Следующие флаги, весьма влияющие на производительность, задают конкретный процессор для целевой машины:

`-mcpu=значение`

или

`-march=значение`

Различие между ними в том, что программа, собранная с флагом `-mcpu`, будучи оптимизированной под заданный в качестве значения процессор, сохраняет способность запуска на более младших моделях, тогда как флаг `-march` доведет программу до того, что она сможет запуститься только на указанном процессоре или более старшем.

Допустимые значения флагов `-mcpu` и `-march` зависят от версии компилятора gcc. Однако в современных версиях, входящих в состав большинства дистрибутивов (gcc от 3.3.X и выше), здесь могут быть заданы все популярные ныне процессоры семейства x86:

- `pentium`;
- `pentium-mmx`;
- `pentiumpro`;
- `pentium2`;
- `pentium3`;
- `pentium4`;
- `athlon`;
- `athlon-tbird`;
- `athlon-xp`;
- `athlon-mp`;
- `athlon64`;
- `winchip`.

Детали можно посмотреть в документации на наличную версию gcc.

Особенно богатые возможности для оптимизации под современные процессоры предоставляет gcc-3.4.X. В нем имеется поддержка и AMD64, и Prescott с его набором новых инструкций и улучшенным HyperThreading, а главное, судя по имеющимся сообщениям, существенно выросло качество (в том числе и поддержка HT). И если раньше (в версиях 3.3.X), сборка, например, под Pentium 4 подчас давала худшие результаты, нежели чем под абстрактный i686, то теперь для процессоров от Intel имеет смысл подбирать наиболее близкое значение флага `-march` (для процессоров AMD точное указание процессора всегда давало хорошие результаты).

Следующие специфичные флаги оптимизации позволяют задействовать специальные наборы инструкций современных процессоров, от MMX до 3DNow, и SSE всех номеров. Для этого сначала определяем, куда в этих случаях должен обращаться процессор, к стандартному сопроцессору или соответствующим блокам специализированных инструкций. Делается это с помощью флага

```
-mfpmath=значение
```

в качестве какового могут выступать `387` (стандартный сопроцессор) или `sse` (блок SSE-расширений); согласно документации, можно задать оба параметра, правда, смысл этого остается мне не вполне ясным.

Насколько я понимаю, при компиляции под Athlon более оправданным будет флаг

```
-mfpmath=387
```

учитывая максимально мощный сопроцессор этого "камушка". А вот в случае с Pentium III и особенно Pentium 4 резонным выбором будет скорее

```
-mfpmath=sse
```

К этому следует приложить еще парочку флагов: `-mmmx` плюс `-msse` или `-msse2`.

Впрочем, флаги для специальных наборов инструкций имеют смысл только для программ, которые в принципе способны их использовать. По имеющимся у меня сведениям, они дают хорошие результаты для некоторых видеоприложений и вообще всякого рода multimedia. Ожидать же от них выигрыша в быстродействии того же компилятора вряд ли оправданно.

Процессорно-ориентированными флагами возможности оптимизации не исчерпываются. Есть еще флаги, специфическим образом обрабатывающие код для достижения максимальной производительности вне зависимости от типа процессора (вплоть до нарушений стандартов POSIX/ANSI, типа

```
-ffast-math
```

благодаря чему теоретически можно собрать максимально быстрый код). Детальное их описание далеко выходит за рамки нашей темы. Тем более, что его можно найти в фундаментальном руководстве по gcc, написанном Ричардом Столлменом с соавторами. Русский перевод доступен, например, на **http://linuxshare.ru/docs/devel/languages/gcc/gcc1.html**.

Как уже говорилось, при использовании флагов оптимизации, особенно достаточно жестких типа

```
-O3-march=значение
```

не говоря уже о

```
-ffast-math
```

необходимо учитывать, что отнюдь не все программы гарантированно соберутся с ними. А те, что соберутся, вовсе не обязаны безукоризненно функционировать. А уж что при этом будет достигнут большой выигрыш в быстродействии, приходится только надеяться.

Я достаточно много экспериментировал с различными флагами оптимизации, в том числе и весьма жесткими, при разных версиях компилятора и на разных системах (Gentoo, CRUX, Archlinux, FreeBSD пятой ветви, DragonFlyBSD). И, в конце концов, пришел к весьма умеренному решению (на примере Pentium 4):

```
CFLAGS="-O2 -march=pentium4"
CXXFLAGS="$CFLAGS"
```

Вероятно, рекордных результатов при нем не добиться, но оно дает гарантированно стабильный результат. По крайней мере, и ядро, и среда DragonFlyBSD (при gcc-3.4.X) собираются без проблем, как и большинство используемых мной портов.

Тем не менее, любители экстремальной сборки могут попробовать нечто вроде этого:

```
CFLAGS="-O3 -march=pentium4 \
       -fomit-frame-pointer \
       -funroll-loops -pipe \
       -mfpmath=sse -mmmx -msse2"
CXXFLAGS="$CFLAGS"
```

А при желании еще и подобрать специальные флаги оптимизации для программ Си++, чем я никогда не озадачивался вообще.

К слову сказать, флаги оптимизации задаются различными способами:

- в командной строке при конфигурировании программы
  ```
  CFLAGS="значения" ./configure
  ```
- в виде переменных окружения в данном сеансе шелла в sh-совместимых оболочках
  ```
  export CFLAGS="значения"
  ```
- в csh и tcsh
  ```
  setenv CFLAGS "значения"
  ```
- глобально, раз и навсегда, в профильном файле (например, в /etc/profile или /etc/cshrc);
- в конфигурационном файле системы портов, если таковая используется (типа /etc/make.conf в BSD).

Осталось определить, что дает столь жесткая оптимизация? Компилятор gcc, пересобранный указанным мной ранее образом, на сборке ядра демонстрирует быстродействие на 10—15% выше, чем собранный по умолчанию. Учитывая, что на современных машинах ядро собирается минут за 10—15, казалось бы, немного. Однако при сборке монстров типа оконной системы Икс, интегрированной среды KDE или, скажем, OpenOffice, игра может и стоить свеч. Да и прикладные программы, собранные с флагами оптимизации, работают существенно быстрее: Mozilla, для примера, чуть не в полтора раза, хотя для других Исковых программ столь высоких результатов получить не удается.

Средства управления пакетами

Вопросу ручной сборки программ выше было уделено очень много места. Однако на самом деле пользователю не так уж и часто приходится заниматься этим делом. Потому что любая уважающая себя система, как было сказано в *главе 2*, обладает тем или иным способом автоматизации установки и удаления программ, избавляющей не только от отслеживания зависимостей, но обычно даже и от компиляции, то есть системой управления пакетами.

Средства управления пакетами чрезвычайно разнообразны. В первом приближении их можно разделить на два класса: системы портов и собственно системы пакетного менеджмента.

Системы портов пришли из мира BSD, где впервые появились во FreeBSD. Собственно, порты — это родовое имя системы пакетного менеджмента этой ОС. Все остальные аналогичные системы возникли и развивались под сильным ее влиянием. Наибольшее распространение и известность получили pkgsrc и портежи (portages).

Первая система возникла первоначально в NetBSD, постепенно превратившись в кроссплатформенное средство управления пакетами: существуют ее версии для всех BSD-систем (включая DragonFly), ряда дистрибутивов Linux и даже для таких проприетарных UNIX, как Solaris и IRIX.

Система портежей была разработана для дистрибутива Gentoo Linux. Однако и она быстро обрела черты кросс-платформенности: более или менее активно разрабатываются ее варианты для FreeBSD и NetBSD, поговаривают о переносе ее на Hurd и другие дистрибутивы Linux.

Тем не менее, практически каждый дистрибутив Linux, относимый к классу Source Based, обладает собственной, более или менее оригинальной, портообразной системой (почему я и полагаю, их следует называть скорее порти-

руемыми дистрибутивами). Тут следует упомянуть и ABS (Archlinux Building System) из одноименного дистрибутива, и порты CRUX, и Sorcery из Sorcerer Linux и его потомков (Source Mage и Linar). И примеры портообразных систем множатся с каждым днем: только за истекший год появились дистрибутивы Rubyx и MyGeOS, каждый со своей портообразной системой.

Несмотря на такое разнообразие конкретных реализаций, общие черты любой портообразной системы вычленить очень легко: это набор правил для отыскания в Сети исходников устанавливаемых программ, их получения на локальную машину, развертывания тарбаллов, конфигурирования, сборки (компиляции и линковки), инсталляции в файловую систему и, возможно, постинсталляционного конфигурирования. То есть по этим правилам выполняется весь тот цикл действий, которые мы проделываем при ручной сборке любой программы, но в автоматическом режиме. Однако вдобавок к этому система портов включает такой важный для управления пакетами компонент, как средства выявления и отслеживания зависимостей, как жестких, так и мягких, опциональных, коррекции последних в ту или иную сторону, ну и, конечно же, автоматического удовлетворения тех и других.

Собственно системы пакетного менеджмента — это инструменты для установки и удаления ранее скопмилированных бинарных пакетов. Что такое бинарный пакет, легко представить себе, если вернуться к процессу сборки программ, описанному ранее. Как мы помним, собственно стадия сборки (исполнение процедуры `make`) завершается тем, что полностью собранные, готовые к употреблению, компоненты пакета располагаются в промежуточном каталоге.

По умолчанию каталог для промежуточных продуктов сборки тот же самый, что и для исходных текстов. Однако, если на стадии конфигурирования задать какой-либо иной адрес сборки, например,

```
$ ./configure --prefix=/path_to/pkgname
```

то бинарные компоненты пакета будут собраны в отдельном каталоге с подкаталогами pkgname/bin, pkgname/lib, pkgname/share и т. д. И если этот каталог заархивировать утилитой `tar` и сжать посредством `gzip` или `bzip2` — то получится автономный бинарный пакет в простейшей своей форме — в виде простого тарбалла. Для использования остается только развернуть его и инкорпорировать в древо файловой системы. Эту задачу и выполняют система пакетного менеджмента, обеспечивая заодно и контроль зависимостей при установке.

По формату пакеты в первом приближении можно разделить на две группы: те, что содержат внутри себя метаинформацию (в частности, информацию о зависимостях пакетов), и таковой лишенные.

К первой группе относятся широко распространенные форматы пакетов rpm (Red Hat Packages Manager, характерный для одноименного дистрибутива и его многочисленных потомков) и deb (свойственный дистрибутиву Debian и его клонам). И тот, и другой, помимо собственно архива бинарных файлов и путей к ним, содержат данные о зависимостях, хотя и представленные в разной форме.

Пакеты без метаданных — это обычные тарбаллы, то есть компрессированные tar-архивы типа *tar.gz/*tar.bz2 (часто фигурирующие в форме tgz и tbz). Важно, что сами по себе tgz,tbz и им подобные — это форматы вовсе не пакета, а именно архива (то есть определяются используемой утилитой компрессии gzip или bzip2 соответственно). А важно это потому, что те же самые tgz/tbz архивы могут прекрасно содержать в себе и метаинформацию, оказываясь более сходными с пакетами rpm или deb. Характерным примером чего являются packages BSD-систем, включающих в себя полное описание зависимостей.

Еще более существенно то, что отсутствие в составе пакета информации о его зависимостях отнюдь не препятствует контролю: он может осуществляться за счет внешних баз данных репозиториев пакетов (то есть официальных их хранилищ) и локальных баз данных пакетов установленных. А функции удовлетворения зависимостей в этом случае целиком ложатся на программы, осуществляющие пакетный менеджмент. И надо отметить, что управление "чистыми" тарбаллами подчас оказывается более гибким, чем пакетами с информацией об их зависимостях.

Средства пакетного менеджмента жестко привязаны к формату пакетов: для установки rpm-пакетов служит одноименная утилита (`rpm`), пакеты deb устанавливаются посредством утилиты `dpkg`, для пакетных тарбаллов предусмотрены собственные средства. Конечно, существуют средства взаимной трансформации пакетов разных форматов (типа `rpm2cpio`, `rpm2tgz` или почти универсальной утилиты `alien`), однако возможности их применения ограничены, очевидно, что из rpm-пакета (и тем более "чистого" тарбалла) получить полноценный deb-пакет невозможно.

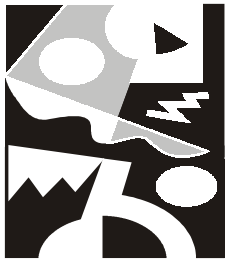
Однако существуют еще и средства пакетного метаменеджмента, если так можно выразиться (собственно, только они и заслуживают названия систем управления пакетами). Наиболее известное и распространенное из них — apt. Появившийся сначала в Debian и рассчитанный, соответственно, на deb-пакеты, он очень быстро стал универсальным кросспакетным механизмом установки, удаления и обновления программ, успешно работая с пакетами rpm (дистрибутивы Connectiva, Altlinux), тарбаллами Slackaware (механизм slapt-get). И в принципе не видно препятствий к применению его

для тарбаллов любого формата, от "чистых" до сколь угодно насыщенных метаинформацией.

Под явным влиянием apt возникли и иные системы пакетного менеджмента: yum, urpmi и т. д. Однако они ориентированы только на rpm-пакеты (вероятно, их можно использовать и для иных форматов, но кому это нужно при наличии apt?) и потому не получили столь широкого распространения, оставаясь принадлежностью "родительских" дистрибутивов и более-менее точных их клонов (yum, насколько мне известно, используется только в Red Hat/Fedora и ASPlinux).

Следует помнить, что резкую границу между системами портов и средствами пакетного менеджмента провести нелегко. Все портообразные системы позволяют выполнить сборку автономных бинарных пакетов и, соответственно, включают средства их установки, регистрации и удаления. А в системах пакетного менеджмента можно обнаружить инструментарий для автоматической компиляции программ из исходников, хотя они и играют сугубо вспомогательную роль.

Глава 15



О шеллах

- О шеллах вообще.
- Какие бывают шеллы.
- Принципы конфигурирования.
- Проблема выбора.
- Sh-совместимые оболочки.
- Кое-что о csh и tcsh.

Перефразируя классиков советской фантастики, можно сказать, что жизнь дает POSIX'ивисту три радости: дружественный шелл, любимый текстовый редактор и много, очень много приложений для работы. Без любой из первых двух радостей прожить можно. Но это значит, что радостей будет меньше. А ведь их всего три. Так что эту главу я посвящаю первой из радостей — шеллам. Тем более, что это еще и первое приложение, с которым сталкивается пользователь после авторизации в системе.

О шеллах вообще

Шелл (shell), именуемый по-русски командной оболочкой, командным интерпретатором, командным процессором или иными, столь же изящными словосочетаниями, — это первая программа, с которой сталкивается пользователь POSIX-совместимой ОС. И с ним же последним он расстается, выходя из системы. А все его действия во время работы суть прямые или опосредованные команды, выполняемые в среде шелла. И даже если основная часть работы пользователя проходит в графическом режиме, в окружении интегрированных сред или оконных менеджеров — окно терминала с приглашением командной строки быстро станет неотъемлемым атрибутом любого десктопа. Ибо, как я пытался показать в *главе 12* и ряде интермедий, именно команды оболочки — самый простой и эффективный путь к выполнению всех операций по управлению файлами, многих задач обработки тек-

ста, да и просто запуска любых программ, что в консоли, что в графическом режиме.

Кроме того, всегда следует помнить, что вообще функционирование UNIX-подобной системы в значительной мере происходит в шелл-среде. Ибо шелл-сценариями являются все скрипты инициализации системы, многие общесистемные и пользовательские конфигурационные файлы, такие системы управления пакетами, как порты FreeBSD и, частично, их потомки в разных дистрибутивах Linux. Ну и то, что любой пользователь свободных ОС рано или поздно начинает писать собственные сценарии оболочки, неизбежно, как распад мировой системы социализма (кто еще не начал, уж поверьте мне на слово).

Сказанного, думаю, достаточно, чтобы отнестись к выбору шелла со всей ответственностью. При выборе этом должны учитываться два аспекта применения шелла, как среды для пользовательской работы, так и системы для исполнения общесистемных и пользовательских сценариев.

Первый аспект охватывается понятием интерактивного шелла. Это любой экземпляр командной оболочки, запущенный пользователем непосредственно. Если этот экземпляр запускается при входе пользователя в систему, его называют login shell (то есть главная пользовательская оболочка). Очевидно, что login shell также интерактивен, однако в сеансе работы каждого пользователя он будет единственным. Просто же интерактивных шеллов можно запустить сколько угодно, например, в каждом окне эмулятора терминала в Иксах будет функционировать собственная копия интерактивного шелла.

Имя исполняемого файла, запускающего login shell (вместе с полным путем, например, /bin/sh), — атрибут учетной записи каждого реального пользователя системы. Теоретически в этом качестве могут выступать не только собственно шеллы, но и интерпретатор какого-либо языка программирования (например, Tcl), программа типа Midnight Commander или даже текстовый редактор. Однако эти случаи специальные и далее рассматриваться не будут.

Неинтерактивный шелл — это экземпляр командной оболочки, вызываемый при выполнении пользователем любой команды или любого сценария. Он может быть вызван неявным или явным образом. Первый случай имеет место быть при выполнении команды из строки оболочки, ведь в этом случае, как мы видели в *главе 7*, посредством системного вызова `fork` создается точная копия породившего процесса (то есть той же интерактивной оболочки), а уже она вызовом `exec` запускает на исполнение введенную пользователем команду. Явный же вызов шелла происходит при выполнении сценариев оболочки, любой из них запускает собственный экземпляр командного интерпретатора.

При составлении пользовательского или системного сценария шелл, вызываемый для его исполнения, можно указать явным образом — и настоятель-

но рекомендуется этой возможностью пользоваться. Делается это в первой строке скрипта, называемой sha-bang, которая по правилам должна иметь вид вроде

```
#!/bin/bash
```

После символов решетки (#) и восклицания (! — видимо, для пущей экспрессии) следует указывать полный абсолютный путь к исполняемому файлу, оболочку запускающему. Делается это и во избежание недоразумений: так, просто единичный символ решетки в первой строке может быть интерпретирован не как комментарий, а как указание на запуск командной оболочки csh. И если сценарий был написан не для нее (а, например, для того же /bin/sh), он, в силу различия синтаксиса, просто не будет выполнен.

Подчеркнем еще раз: хотя команда, запускающая интерактивный или неинтерактивный шелл, может носить то же имя, что и команда на запуск login shell (а в Linux обычно так и есть), это разные экземпляры программы, которые в общем случае могут быть настроены независимо. А значит, и вести себя разным образом. И потому не следует удивляться, когда bash в эмуляторе терминала не реагирует на управляющие клавиши, привычные для того же bash в виртуальной консоли. А Midnight Commander отказывается в своей командной строке опознавать пути к исполняемым файлам.

Очевидно, что претензии пользователя к интерактивному и неинтерактивному шеллам могут быть разными. В первом случае, как явствует из названия, важнее всего удобство интерактивной работы: развитые средства автодополнения и работы с историей команд, возможности гибкой и информативной настройки приглашения командной строки. Для неинтерактивного же шелла на первый план выходят быстродействие и совместимость.

С быстродействием все понятно, когда единственной задачей командной оболочки является вызов на исполнение другой команды (или, в случае скрипта, серии связанных команд), тратить ресурсы на обеспечение дополнительных возможностей было бы излишеством.

Какие бывают шеллы

Большая часть командных оболочек делится по синтаксису интерпретируемого ими языка на две группы: sh-совместимые и csh-совместимые. На самом деле различия между ними синтаксисом команд не исчерпываются, а лежат глубже — в подходе к обработке командных конструкций, к чему мы еще вернемся.

Оболочки, относимые к sh-совместимым, происходят от первой командной оболочки первых UNIX-систем, которую так и называют — shell, или Bourne Shell (то есть шелл Борна). В ней были заложены многие возможности для

интерпретации команд и их конструкций, то есть составления системных и пользовательских сценариев. Однако по своим интерактивным возможностям она оставляла желать лучшего, и потому на базе ее была создана оболочка Корна (Korne Shell).

Шелл Корна сохранил совместимость с борновским шеллом на уровне синтаксиса, однако привнес в него как дополнительные возможности интерпретации команд, так и приемы, направленные на удобство интерактивной работы. И потому именно он лег в основу стандарта POSIX, которому должны удовлетворять командные оболочки совместимых с ним систем.

Следует заметить, что именно такой стандартизированный шелл должен вызываться при исполнении общесистемных сценариев инициализации любой POSIX-системы. Сам же по себе POSIX shell — некая мифическая абстракция, ближе к которой подходят /bin/sh — пользовательская оболочка из FreeBSD и ash, принятая в качестве стандартной в NetBSD (а также широко используемая во всяких мини-дистрибутивах Linux).

Шеллы и Борна, и Корна не были свободно распространяемыми программами в терминах GPL-совместимых лицензий. Поэтому ни тот, ни другой не могли использоваться в таких ОС, как *BSD или Linux, хотя свободная реализация оболочки Корна (pdksh) и была создана. Однако, как и ее прототип, шелл Корна, она, несколько развившись относительно первозданного Bourne shell, обладала интерактивными достижениями, уже далекими от идеала. Столь же слабы они были и в ash. И потому наиболее широкое распространение из всего sh-совместимого семейства получила оболочка bash (что расшифровывается как "еще одна оболочка Борна", "заново рожденный шелл" и тому подобным образом), разработанная в рамках проекта GNU.

Популярность bash в немалой степени была обусловлена его интерактивными возможностями, он аккумулировал все достижения интерактивной мысли sh- и csh-совместимых оболочек, прибавив к ним немало собственных. И умудрившись при этом сохранить базовую совместимость с POSIX shell. Конечно, многие его собственные особенности выходили за рамки этого стандарта. Однако для соответствия оному был предусмотрен режим совместимости: bash был способен эмулировать стандартный POSIX shell.

Однако главным для популярности bash было то, что эта оболочка оказалась тесно интегрирована с операционной системой Linux: именно bash волею судеб стал одной из первых программ, которые Линус Торвальдс запустил поверх созданного ядра. И потому идеи bash-скриптинга пронизали Linux до самых его оснований. Ну, а для совместимости использовался тот самый режим эмуляции: в Linux файл, запускающий POSIX shell (по стандарту он должен именоваться /bin/sh), являет собой жесткую или символическую ссылку на реальный /bin/bash. Последний же, будучи вызванным таким образом, полностью воспроизводит функционально стандартный POSIX shell (разумеется, путем утраты своих продвинутых функций).

Клан csh-совместимых оболочек развивался параллельно сынам и пасынкам Борна. Собственно оболочка csh (или C-shell) была создана в университете Беркли на начальных этапах реализации проекта BSD UNIX. Первым ее разработчиком был Билл Джой — автор также и культового текстового редактора юниксоидов, vi, а в последующем один из основателей фирмы Sun.

Оболочка C-shell получила дополнительные интерактивные возможности. Главное же, языку, ею интерпретируемому, были приданы черты синтаксического сходства с языком Си (откуда, собственно, и название — C-Shell, хотя не следует думать, что на настоящий Си ее язык похож). В результате оболочка csh оказалась весьма эффективной как для интерактивной работы, так и при создании сценариев. Только вот сценарии эти не были совместимы со скриптами POSIX shell, обретшего уже силу стандарта. При всей эффективности пользовательского шелл-программирования для создания общесистемных сценариев она оказалась практически непригодной.

В отличие от большинства прочих достижений берклианской мысли, оболочка csh, по не вполне ясным для меня причинам, не обрела статуса свободной программы. Поэтому она не могла использоваться даже в BSD-системах. На замену ей была изобретена свободная оболочка tcsh — не просто функциональное воспроизведение, но и дальнейшее развитие оболочки csh. По интерактивным возможностям она, как минимум, не уступает bash и потому прочно утвердилась в стане свободных BSD-клонов как пользовательский шелл по умолчанию.

В частности, оболочка tcsh принята в качестве login shell для суперпользователя во FreeBSD. Вызывается она в режиме совместимости с csh, однако /bin/csh — не более, чем жесткая ссылка на /bin/tcsh.

Оболочка tcsh используется в качестве универсального пользовательского шелла также в OpenBSD и DragonFlyBSD. Однако характерно, что все общесистемные сценарии в обеих ОС написаны, тем не менее, в соответствии с требованиями POSIX Shell.

Принципы конфигурирования

Поведение конкретного экземпляра шелла того или иного вида определяется, кроме принадлежности к одному из описанных семейств, также и файлами его конфигурации. Практически все широко используемые шеллы имеют минимум два конфигурационных файла: профильный файл (profile), считываемый при запуске login shell, и rc-файл, из которого берутся настройки любого шелла интерактивного. В некоторых оболочках предусмотрен еще и конфигурационный файл, обращение к которому происходит при запуске любого экземпляра шелла, в том числе и неинтерактивного. А под-

час предусматривается и конфигурационный файл для завершения данного шелл-сеанса (очевидно, что он имеет смысл только для login shell).

Содержание профильного файла — это переменные, которые должны наследоваться всеми процессами. В их числе такие, как тип терминала (переменная `TERM`), каталоги для поиска исполняемых файлов и man-страниц (переменные `PATH` и `MANPATH` соответственно), имя редактора и pager по умолчанию (`EDITOR`, `PAGER`). В rc-файл резонно помещать переменные оболочки и псевдонимы команд, имеющие смысл только для интерактивно запускаемых экземпляров шелла. Впрочем, это не обязательно, подчас профильный файл содержит единственную строку, предписывающую прочитать содержимое rc-файла для данного шелла (в котором и определяются все параметры пользовательского окружения).

Имена конфигурационных файлов различны в разных оболочках. В sh-совместимых оболочках конфиг для login shell обычно имеет в своем имени слово "profile" (откуда и пошло — профильный файл), имя интерактивного конфига образуется по модели shell_namerc. Место размещения общесистемных шелловских конфигов — обычно каталог /etc. Эти файлы будут считываться в любом случае для login shell любого пользователя и для любого экземпляра запущенного каждым пользователем интерактивного шелла. И их изменение — компетенция исключительно администратора системы.

Однако каждый пользователь имеет возможность создать также свои собственные файлы конфигурации для своего login shell (и любых шеллов, запускаемых им интерактивно или из собственных сценариев). Они располагаются в корне домашнего каталога пользователя (~/), обычно одноименны с общесистемными, но предваряются символом точки, почему и называются dot-файлами (это, впрочем, относится к пользовательским конфигурационным файлам практически любых программ). Пользовательские конфиги считываются после общесистемных, и потому содержащиеся в них параметры перекрывают, дополняют или отменяют значения параметров общесистемных.

А относительный порядок считывания профильного и rc-файла также различен в разных оболочках, и сложился он исторически. Первый шелл Борна имел один-единственный набор конфигов — общесистемный /etc/profile и пользовательские ~/.profile). И когда в его клонах появились отдельные конфиги для интерактивных шеллов, они считывались после профильного в таком порядке:

```
/etc/profile -> /etc/shrc -> ~/.profile -> ~/.shrc
```

В C-shell изначально имелось два конфига: общий cshrc и login — конфиг для пользовательского шелла. Причем первый считывался при старте любого экземпляра шелла, в том числе и запускаемого при авторизации. Лишь

после этого определялось, является ли данный экземпляр шелла пользовательским, и если да — происходило считывание файла login:

```
/etc/csh.cshrc -> /etc/csh.login -> ~/.cshrc -> ~/.login
```

Имена С-конфигов могут меняться в разных ОС и дистрибутивах (приведенный пример относится к FreeBSD и DragonFlyBSD). Однако порядок обращения к ним унаследован современным свободным клоном C-shell — tcsh (именно он под именем /bin/csh фигурирует на самом деле во всех BSD-системах).

Порядок обращения к конфигурационным файлам не есть нечто данное от века: в сущности, он определяется при компиляции данной оболочки (как — в конкретном случае можно посмотреть из вывода сценария конфигурирования

```
./configure --help
```

перед компиляцией). И может быть изменен на противоположный более того, имена конфигов могут быть изменены произвольным образом, или для разных оболочек могут быть установлены одни и те же конфиги. Однако в большинстве случаев с прекомпилированными шеллами он по умолчанию именно таков, как я описал.

Да, чуть не забыл сказать: в приведенных примерах молчаливо предполагалось, что исполняемый шелл и его общесистемные конфиги находятся непосредственно в подкаталогах корня файловой системы (/bin и /etc соответственно). Это отнюдь не обязательно: например, файлы bash или zsh в портообразных FreeBSD будут иметь своим местопребыванием по умолчанию подкаталоги в /usr/local, в дистрибутивах Linux все оболочки, кроме общесистемной bash, окажутся, скорее всего, в подкаталогах /usr, и т. д. Однако очень важно, чтобы login shell пользователя root был собран (с помощью соответствующих опций, о чем говорилось в *главе 14*) так, чтобы его исполняемый файл и конфиг инсталлировались бы именно в /bin и /etc: иначе они окажутся недоступными в однопользовательском режиме, когда такие ветви файловой системы, как /usr и /usr/local, могут быть просто не смонтированы.

Проблема выбора

Из приведенного краткого обзора можно видеть, что в плане шеллов выбор пользователя достаточно обширен. А ведь я остановился только на самых распространенных. Однако рискну предположить, что большинство начинающих пользователей Linux об этом не особо задумываются. Ведь во всех его дистрибутивах в качестве общесистемного шелла и пользовательского шелла по умолчанию принят bash, обладающий как развитыми средствами интерпретации, так и продвинутыми интерактивными возможностями, да

еще при сохранении совместимости со стандартом. Так зачем, казалось бы, искать добра от добра?

Первый вариант ответа очевиден: добро это дополнительное ищется в тех случаях, когда необходима минимизация ресурсов. Когда bash оказывается излишне громоздким и медленным. Тем не менее, маленький и быстрый шелл Альмквиста (ash) может оказаться подходящим не только в спасательных целях, но и для всякого рода скриптинга. Хотя работать в строке ash регулярно мне бы не хотелось...

Вторая причина поиска нового шелла — неудовлетворенность возможностями имеющегося. Эта проблема остро встает перед пользователями FreeBSD, уж очень убог его login shell для обычного пользователя (/bin/sh) в плане интерактивной работы. Что особенно наглядно проступает в сравнении с могучим tcsh, каковой по определению получает в свое распоряжение root-оператор.

Должен заметить, что именно при первом приобщении к FreeBSD я и перепробовал целый ряд доступных в ней шеллов. Благо, через систему портов или коллекцию пакетов они столь же легко удалялись, как и устанавливались. Не обошел я своим вниманием и tcsh — и в целом остался им доволен. Как интерактивная оболочка, tcsh, на мой взгляд, не уступает bash. А простой, логичный и лаконичный синтаксис его языка немало способствует в деле шелл-скриптинга. Правда, в этом и кроется, пожалуй, единственный недостаток tcsh: даже виртуозное им владение не освобождает от необходимости хоть как-то изучать какой-либо POSIX-совместимый шелл, ведь общесистемные сценарии все равно пишутся на нем...

И, наконец, третья причина для изысканий — поиски идеала. А не они ли движут, осознанно или нет, изрядной долей пользователей свободных POSIX-систем? Спору нет, bash — оболочка хорошая, но до такого идеала явно не дотягивающая, слишком много в нем направлено на достижения баланса компактности и функциональности. И тут стоит присмотреться к Z-Shell (или просто zsh), оболочке с лучшими, пожалуй, интерактивными возможностями.

Итак, круг знакомств очерчен, остается к этому знакомству приступить. Приводимые далее описания наиболее распространенных шеллов поневоле будут очень разной степени глубины и подробности, ведь я руководствуюсь исключительно собственным, неизбежно ограниченным, опытом и своими сугубо субъективными симпатиями.

Sh-совместимые оболочки

Оболочку ash (и /bin/sh из FreeBSD) можно рассматривать в качестве эталонного POSIX-шелла. Интерактивные ее возможности проще всего оха-

рактеризовать в сравнении с более "продвинутыми" шеллами, и исключительно от противного. А именно: она не поддерживает автодополнения, не имеет удобных средств доступа к истории команд, и даже средства навигации по командной строке и редактирования сводятся к клавише <Backspace> или эквиваленту — <Ctrl>+<H>. Встроенных команд также немного (около двух десятков).

Что же остается в сухом остатке? Остается поддержка командных конструкций (перенаправления и конвейеров), возможность фонового выполнения команд, определения псевдонимов и функций. Вряд ли этого достаточно для комфортной интерактивной работы. А вот для сочинения сценариев и их исполнения довольно вполне. Более того, скрипты, написанные для исполнения в чистом шелле, то есть имеющие sha-bang вида

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

будут исполняться в любой POSIX-совместимой системе, так как каждая из них содержит исполняемый файл sh и именно в каталоге /bin.

Средства настройки /bin/sh также не блещут разнообразием. Штатно для этой оболочки предусмотрен единственный файл /etc/profile (и парный ему пользовательский профильный файл ~/.profile), считываемый при авторизации. Однако, как мы только что выяснили, применять /bin/sh в качестве login shell не очень удобно, а все прочие его экземпляры оказываются без всяких настроек окружения вообще, что тоже нехорошо. И потому для /bin/sh можно установить и второй конфиг, однако этот факт, как и имя такого файла, необходимо определить явным образом в /etc/profile (или в ~/.profile), например, таким образом:

```
ENV=etc/shrc; export ENV
```

или, соответственно, так:

```
ENV=$HOME/.shrc; export ENV
```

Это увеличивает гибкость настроек /bin/sh, приближая его к более развитым аналогам.

Следующей sh-совместимой оболочкой является bash. Ей посвящено немерянное число материалов, к которым я, не являясь ни ее любителем, ни, тем более, знатоком, добавить ничего не могу. Однако bash — наиболее распространенная среди пользователей Linux командная оболочка, выступающая в этой ОС к тому же общесистемной. Популярна она, насколько мне известно, и среди пользователей иных POSIX-систем, по крайней мере, свободных. И потому не сказать о ней хоть пару слов здесь нельзя.

Оболочка bash поддерживает все интерактивные возможности, свойственные развитым шеллам, как то: автодополнение для команд и путей к файлам, историю оных (включая средства инкрементного поиска), мощные возможности навигации и редактирования командной строки. Важно, что сущест-

вует дополнительный пакет bash-completion: установка его обогащает базовую оболочку множеством опциональных средств настройки автодополнения (в том числе и для командных аргументов).

Схема инициации bash предусматривает наличие пары файлов /etc/profile и /etc/bashrc (для пользовательского шелла и просто интерактивного его экземпляра, а также соответствующих им пользовательских конфигов ~/.bash_profile и ~/.bashrc. Однако порядок их считывания имеет некоторую специфику. При авторизации первым в любом случае считывается общесистемный профильный файл /etc/profile, вслед за ним пользовательский профильный файл ~/.bash_profile, после чего происходит обращение к ~/.bashrc. Однако порядок обращения к конфигам может быть изменен при сборке пакета. А также особым положением файла /etc/profile, в него часто помещают переменные окружения (например, локально-зависимые), которые должны быть общими для всех пользователей данной системы.

В общем, о bash можно было бы говорить еще долго. Однако достаточно заметить, что почти в любой толстой книге про Linux, когда речь заходит о командных оболочках вообще, как правило, имеется в виду именно bash. Немало сведений о ней есть и в Интернете, в том числе в русскоязычном сегменте. И потому в заключение ограничусь ссылками на источники информации:

- BASH конспект, где кратко просуммированы все важные для пользователя данные по этой оболочке (**http://www.junik.lv/~georg/unixlearn/bash-conspect.htm** или **http://www.ods.com.ua/koi/unix/bash-conspect.html**);
- особенности работы оболочки bash, содержащей массу сведений по использованию ее в интерактивном режиме, настройке и т. д. (**http://linux.yaroslavl.ru/docs/conf/gnu-util/bash/bash.html** или **http://www.opennet.ru/docs/RUS/bash/bash-1.html**);
- Advanced Bash-Scripting Guide (**http://gazette.linux.ru.net/rus/articles/index-abs-guide.html**) — практически исчерпывающее руководство по программированию bash-скриптов, которое будет полезно при изучении любых Shell-совместимых оболочек.

А из печатных изданий нельзя забывать о такой книге, как Д. Тейнсли "Linux и UNIX: программирование в shell", Киев: Издательская группа BHV, 2001. В принципе она посвящена абстрактному шеллу, но в ней оговорены все случаи, относимые именно к bash.

И, наконец, Z-Shell, сокращенно zsh. Эта оболочка также принадлежит к клану sh-совместимых. Причем существует мнение, что в ней нашли свое воплощение все прогрессивные тенденции таких развитых оболочек, как bash и tcsh. И, ознакомившись с его возможностями, с этим трудно не согласиться: в zsh есть все, что было хорошего в тех обеих оболочках, но, если так можно выразиться, в превосходной степени. А ознакомиться с ней можно в ближайшей же интермедии.

Кое что о csh и tcsh

Оболочки семейства C-Shell не избалованы вниманием русскоязычных авторов, почему я и решил уделить им особое внимание. Тем более, что tcsh объединяет преимущества синтаксиса csh и удобные средства настройки и интерактивной работы bash: многие из них были ассимилированы в zsh.

Командная оболочка csh пришла к нам из BSD-мира и традиционно пользуется популярностью среди пользователей этих систем. Правда, изначальный ее вариант (собственно csh) не принадлежит к числу свободно распространяемых программ. И потому в открытых BSD-клонах используется ее разновидность от Open Sources — оболочка tcsh. Которую, впрочем, никто не запрещает применять и в Linux.

Как уже говорилось, внешние различия между основными семействами командных оболочек лежат в первую очередь в области синтаксиса собственного их языка. В клонах Bourne Shell язык этот ни на что, насколько я понимаю, особенно не похож. В оболочке C-Shell и ее производных синтаксис встроенного языка, как и следует из названия, схож с таковым языка программирования Си.

Си-подобный синтаксис встроенного языка csh (и tcsh) обеспечивает существенно больший лаконизм сценариев, чем в скриптах shell-совместимых интерпретаторов. Правда, именно при создании сценариев использование оболочки csh может быть ограничено ее несовместимостью со стандартом POSIX.

Различия в синтаксисе между семействами sh и csh отражают различие их обращения с условными выражениями. В классических шеллах это просто последовательности команд (подобные конвейерам), в которых выполнение каждой последующей определяется успешным или неуспешным завершением предыдущей. В csh же они представляют собой вычисляемые арифметические или логические выражения.

Далее, важное с точки зрения пользователя различие — обращение с путями к исполняемым файлам. Клоны шелла Борна при вводе команды перечитывают состав каталогов, включенных в качестве значений переменной `PATH`. В клонах же csh эти значения, считываясь один раз при старте, далее хранятся в чем-то типа собственного буфера, именуемого хэш-таблицей. В результате чего достигается выигрыш в быстродействии исполнения внешних команд. Оборотная сторона — при добавлении к одному из каталогов переменной `PATH` нового файла (типичный случай — при установке новой программы) оболочка csh его просто не увидит; то есть для вызова такой новой программы в текущем сеансе придется указывать полный путь к ее исполнимому файлу. Или перестроить хэш-таблицу, для чего предназначена специальная встроенная команда `rehash`.

Наконец, в csh по иному определяются переменные. Если во всех POSIX-шеллах для этого достаточно задать имя и значение, то здесь этой цели служит специальная встроенная команда `set`, например:

```
set EDITOR joe
```

определит редактор по умолчанию. Впрочем, таким образом будет задана только переменная оболочки, средств экспорта их в среду в csh не имеется. Для задания переменных среды существует отдельная встроенная команда `setenv`:

```
setenv EDITOR joe
```

Каждая из этих команда, данная без аргументов, выведет список определенных переменных оболочки и окружения соответственно. А для вывода значений абсолютно всех переменных есть специальная встроенная команда `printenv`.

Командная оболочка tcsh — это вариант C-Shell с мощными возможностями дополнения имен файлов и редактирования командной строки; нельзя не отметить, что в оригинальной csh они развиты существенно слабее, чем, скажем, в bash. Тем не менее, tcsh полностью совместима со своим прототипом, с точки зрения синтаксиса.

Как и все другие оболочки, tcsh объединяет в себе интерактивный командный процессор и интерпретатор собственного языка сценариев, превращающий ее в простую в обращении, но весьма мощную среду программирования.

Одна из основных функций любой командной среды — исполнение команд, внешних (то есть независимых программ) и встроенных. Встроенные и внешние команды могут иногда дублировать функции друг друга, но при прочих равных условиях применение первых предпочтительней, они выполняются быстрее. И набор встроенных команд — это то, что, помимо всего прочего, отличает командные среды друг от друга и определяет их функциональность.

Среда tcsh содержит достаточно большое количество встроенных команд. Полный их список можно получить с помощью команды `builtins` (к слову сказать, также встроенной), ответом на которую будет список вроде этого:

```
:
@          alias      alloc      bg         bindkey    break
breaksw    builtins   case       cd         chdir      complete   con-
tinue
default    dirs       echo       echotc     else       end        endif
endsw      eval       exec       exit       fg         filetest   foreach
glob       goto       hashstat   history    hup        if         jobs
kill       limit      log        login      logout     ls-F       nice
```

nohup	notify	onintr	popd	printenv	pushd	rehash
repeat	sched	set	setenv	secodec	secodey	
shift						
source	stop	suspend	switch	telltc	time	umask
unalias	uncomplete	unhash	unlimit	unset	unsetenv	wait
where	which	while				

Все эти команды могут использоваться как в интерактивном режиме, так и в составе сценариев (скриптов). Описание команд можно найти в экранной документации. На некоторых наиболее используемых из них я буду останавливаться по ходу дела.

Основные возможности, предоставляемые tcsh в интерактивном режиме, помимо собственно исполнения команд, включают:

- редактирование командной строки;
- дополнение слов (word completion), как для путей, так и для команд;
- хранение и воспроизведение истории команд;
- управление текущими задачами (job control).

Редактор командной строки предоставляет средства навигации внутри нее, с возможностью изменения отдельных знаков и компонентов команд, их опций и аргументов.

Навигация по командной строке и ее редактирование осуществляются двумя различными способами. Первый — использование стандартных клавиш управления курсором для навигации и клавиш <Delete> и <Backspase> для редактирования. Достоинство его в простоте, вернее, в привычности: в ряде случаев эти клавиши ведут себя так же, как и в программах для DOS/Windows. Однако далеко не всегда: на многих типах терминалов хоть какая-то из этих клавиш (а то и все сразу) обнаруживает аномальные особенности поведения.

Этого недостатка лишен второй способ навигации и редактирования — с использованием специальных клавишных комбинаций.

Управляющие комбинации (bindkeys) в большинстве случаев имеют вид <Ctrl>+символ (то есть символьная клавиша нажимается при нажатой клавише <Ctrl>) или <Esc>+символ (когда символьная клавиша нажимается непосредственно сразу после клавиши <Esc>). Все они не чувствительны к регистру и, насколько мне удалось выяснить, также и к раскладке клавиатуры (то есть работают вне зависимости от переключения, например, с латиницы на кириллицу и обратно).

Полный список управляющих комбинаций для tcsh может быть получен командой

```
$ binkdkey
```

Одни из них дублируют стандартные клавиши перемещения курсора, такие как:

- <Ctrl>+<A> — перемещение курсора в начало строки;
- <Ctrl>+<E> — перемещение курсора в конец строки;
- <Ctrl>+<F> — перемещение курсора на один знак вперед;
- <Ctrl>+<B> — перемещение курсора на один знак назад.

Другие же управляющие комбинации дают возможность перемещаться на одно слово вперед или назад (Escape-F и Escape-B соответственно), в предыдущую позицию курсора (<Ctrl>+X-X, когда непосредственно после комбинации клавиш повторно нажимается клавиша <X>), удалять целиком слово (Escape-D) или часть строки после курсора, перемещать знаки (transpose-chars, <Ctrl>+<T>), изменять регистр знаков и многое другое. А поскольку под все эти операции задействованы только клавиши основой части клавиатуры, скорость их выполнения — непревзойденная (при наличии некоторого навыка, доведенного, желательно, до рефлекторного уровня).

Следующая неоценимая возможность tcsh — дополнение слов. Которое работает как для команд, так и для имен файлов и путей к ним. То есть при наборе первых знаков команды (или файла) соответствующее действие (например, нажатие клавиши табуляции) автоматически дополняет недостающие знаки. Если, конечно, набранных знаков хватает для однозначной идентификации. Если же не хватает, есть возможность просмотреть списки доступных вариантов и выбрать из них подходящий (эта функция именуется `autolist`).

Как обычно, дополнение слов выполняется двояким способом: или клавишей <Tab>, или управляющими комбинациями. Из них отметим <Ctrl>+ +<I> — собственно дополнение слова, и <Ctrl>+<D> — вызов списка вариантов для дополнения; в последнем случае курсор обязательно должен стоять после последнего введенного символа — иначе эта комбинация сработает на удаление знака под курсором.

Следует отметить, что вывод альтернатив для автодополнения в tcsh по умолчанию не предусмотрен, требуя специальных настроек в профильном файле (каких — скажу чуть позже).

История команд подразумевает, что некоторое количество ранее введенных команд сохраняется в специальном буфере, и может быть вызвано для просмотра, исполнения или редактирования.

Для этого можно использовать клавиши управления курсором <Up> и <Down>, с помощью которых как бы пролистываются по одной все ранее введенные команды. Аналогичного результата можно добиться и управляю-

щими комбинациями — <Ctrl>+<P> и <Ctrl>+<N> или <Esc>+<P> и <Esc>+ +<N>, каждая пара из которых является аналогом пары <Up> и <Down> соответственно.

Кроме этого, историю эту можно просмотреть с помощью встроенной команды `history`, которая выдаст нумерованный список всех выводившихся ранее (в количестве, определенном в файле конфигурации среды) команд, например:

```
$ history
1  17:19   logout
2  17:57   history
3  17:57   pwd
4  17:57   ls
5  17:57   ls -laFG
6  17:57   history
```

Любая из них вызывается в командную строку с помощью конструкции

```
!#
```

где `#` — порядковый номер команды в списке. Как и во всех прочих развитых оболочках, команду из истории можно отредактировать и (или) запустить на исполнение.

Общесистемные файлы конфигурации tcsh — файлы типа /etc/csh.cshrc и /etc/csh.login (точные имена в разных системах BSD-семейства, в базовый комплект которых входит эта оболочка, моут быть разными, пример приведен для DragonFlyBSD). При необходимости их аналоги, ~/.cshrc и ~/.login, создаются в домашнем каталоге пользователя. В отличие sh-совместимых оболочек, файлы csrc-группы считываются (сначала — общесистемный, затем — "домашний") при запуске любого экземпляра tcsh. Обращение же к файлу /etc/csh.login, а потом и к его "домашнему" аналогу ~/.login, происходит только при запуске оболочки пользователя (login shell), хотя и после файлов cshrc-группы. Для csh порядок считывания dot-файлов определяется при компиляции и может быть изменен пересборкой оболочки.

Кроме того, в домашнем каталоге пользователя может обнаружиться еще два конфигурационных файла, считываемых последовательно после главных: истории команд ~/.history и ~/.cshdirs, в котором описываются стартовый каталог оболочки, если требуется сделать ее отличной от умолчания ~/. При выходе из пользовательской оболочки выполняются действия, предписанные в файле /etc/csh.logout или ~/.logout.

Впрочем, в большинстве случаев пользователь вполне может обойтись единственным конфигом ~/.cshrc (не считая ~/.history, который образуется и за-

полняется автоматически. Как пример, могу привести прокомментированное содержимое своего пользовательского dot-файла для tcsh.

```
# .cshrc — стартовый конфиг tcsh
# считывается при каждом запуске

# Псевдонимы для команд управления файлами

alias h       history 25
# Вывод последних 25 команд из файла истории

alias cp      cp -iR
# Рекурсивное копирование с запросом подтверждения перезаписи
alias cpf     cp -Rf
# Рекурсивное копирование с принудительной перезаписью

alias rm      rm -i
# Удаление файлов с запросом подтверждения

alias rmf     rm -Rf
# Принудительное рекурсивное удаление файлов

alias mv      mv -i
alias mvf     mv -f
# Перемещение/переименование файлов
# с запросом подтверждения и принудительно,
# соответственно

# Псевдонимы команды ls

alias ls      ls -FG
# Колоризованный вывод с типизацией файлов

alias la      ls -A
# Вывод всех файлов, за исключением . и ..

alias ll      ls -l
# Вывод списка файлов в "длинном" формате

alias li      ls -ial
# Вывод всех файлов в длинном формате с указанием идентификаторов
```

```
# Прочие псевдонимы
alias df      df -h
alias du      du -h
# Вывод с подбором единиц измерения

alias less    less -M
Вывод команды less в "more-подобном" виде

# Установка прав доступа для новообразованных файлов
umask 022

# Основные переменные оболочки и окружения

set path = (/usr/bin /usr/local/bin /usr/local/sbin /usr/X11R6/bin)
# Определение путей к исполняемым файлам

set autolist
# Вывод списка альтернатив для автодополнения
# по нажатию табулятора

set ignoreeof
# Запрет завершения сеанса по комбинации <Ctrl>+D

set correct = cmd
# Автокоррекция команд

set histdup = erase
# Очистка файла истории команд от дубликатов

set prompt = '%~->'
# Установка вида приглашения

setenv EDITOjoe
setenv PAGEless
# Редактор и Pager по умолчанию

# Переменные для интерактивных экземпляров tcsh
if ($?prompt) then
       # Установить следующие переменные,
       # если определена переменная prompt
```

```
        set history = 1000
        set savehist = 1000
        # Установить число строк
        # в истории команд текущего сеанса
        # и в файле истории, соответственно
        if ( $?tcsh ) then
                # Поведение клавиш (только для tcsh)
                bindkey "^W" backward-delete-word
                bindkey -k up history-search-backward
                bindkey -k down history-search-forward
        endif
endif
```

Приведенным примером возможности настройки tcsh отнюдь не исчерпываются, за подробностями можно обратиться к команде

`man tcsh`.

Интермедия: командная оболочка zsh

- Обзор возможностей.
- Приступаем к установке.
- Начинаем настройку.
- Личный рецепт приготовления.

Главным героем этой интермедии выступит командная оболочка Z-Shell, сокращенно — zsh. Она принадлежит к клану sh-совместимых и первоначально разрабатывалась Паулем Фальстадом (Paul Falstad), ныне же развивается в рамках самостоятельного проекта (**http://www.zsh.org/**) сообществом энтузиастов (Zsh Development Group) при координации Петера Стефенсона (Peter Stephenson). В отличие от bash, прямого (как, впрочем, и косвенного) отношения к GNU zsh не имеет, распространяется под собственной лицензией BSD-стиля, и, следовательно, является полностью свободной программой.

Обзор возможностей

В zsh нашли свое воплощение все прогрессивные тенденции таких развитых оболочек, как bash и tcsh. И, ознакомившись с его возможностями, с этим трудно не согласиться — в zsh есть все, что было хорошего в тех обеих оболочках, но, если так можно выразиться, в превосходной степени.

Действительно, какими особенностями определяется в первую очередь удобство интерактивной работы в командной оболочке? В порядке, в котором с ними сталкивается пользователь, это будут:

- автодополнение командной строки;
- возможности навигации по ней и ее редактирования;
- просмотр буфера истории команд;
- возможность минимизации ввода за счет использования псевдонимов.

Автодополнением клавишей <Tab> команд, частично введенных в ответ на приглашение оболочки, трудно удивить пользователей bash или tcsh. Столь же естественно, что при возможности безальтернативного дополнения именно оно и происходит, а при наличии некоторых альтернатив выводятся возможные варианты. Однако zsh идет дальше и после вывода таковых в от-

вет на последующие нажатия клавиши табулятора начинает автоматический их перебор.

Автодополнение путей к файлам, выступающим в качестве аргументов команд, тоже не бог весть какое новшество. И единственное отличие zsh от прочих оболочек выражается в том, что и для путей действует автоматический перебор вариантов клавишей табулятора. А вот то, что автодополнение работает так же и для аргументов команд, например, команды `man`, окажется приятной неожиданностью. Так, чтобы вызвать полное экранное руководство по zsh, достаточно набрать

```
$ man zsha
```

и нажать табулятор, чтобы развернуть его до полного имени:

```
$ man zshall
```

Более того, автодополнению (и автоматическому перебору его возможностей) подвержены даже опции многих команд. Это особенно показательно для таких синтаксически сложных команд, как `find`. Так, последовательность

```
$ find / -na
```

будет автоматически дополнена до

```
$ find / -name
```

А после указания этого для указания опции действия можно ограничиться вводом символа дефиса

```
$ find / -name filename -
```

и выбрать необходимое действие из предложенного списка. Например, `print` — вывод на экран, или `exec` — исполнение сторонней команды.

И еще один частный случай автодополнения уникален для zsh: развертывание путей при сокращенном их наборе. Так, чтоб просмотреть содержимое каталога /usr/portage/distfiles, достаточно набрать в командной строке

```
$ ls /u/p/di
```

и нажать клавишу табуляции: сокращенный ввод пути (разумеется, при безальтернативности оного) будет автоматически развернут до полного.

Автодополнение в zsh гармонично сочетается с автокоррекцией (spelling командной строки). Конечно, и это само по себе не уникально. Однако проверка правильности ввода и автокоррекция в zsh распространяются не только на встроенные (как в bash) и даже внешние (как в tcsh) команды, но даже на пути и аргументы. Причем если автокоррекция становится назойливой (например, для команд типа `cp` или `mv` она порывается исправить аргументы на имена существующих файлов), ее можно отключить — и именно только для определенных команд.

Средства навигации по командной строке и ее автоматического редактирования — необходимое условие комфорта в интерактивной работе внутри оболочки. Здесь говорить, казалось бы, не о чем, управляющие клавишные последовательности для таких действий давно уже вошли в обиход всех командных оболочек, претендующих на развитость. Однако и в этой области zsh есть чем похвастаться: в нем задействованы все комбинации клавиш для перемещения и удаления (как посимвольного, так и командными "словами" и фрагментами строки), которые существуют в bash и tcsh, причем — построенные по тем же принципам.

Управляющие последовательности в zsh построены по принципу сочетания клавиши <Ctrl>+литера или <Meta>+литера, причем вторая комбинация обычно выступает в качестве "усиленного" варианта первой. Так, если <Ctrl>+<D> удаляет символ в позиции курсора, то сочетание <Meta>+<D> проделывает это для всех символов от позиции курсора до конца командного слова.

Предусмотрены в zsh и клавишные комбинации для таких действий, как преобразование регистра литерных символов, "перетасовки" символов и командных "слов" в строке, заключения строки в кавычки (а при необходимости — и экранирования их символами обратного слэша). Есть, конечно же, и комбинация для многоуровневой отмены ввода.

Управляющие последовательности в zsh реализованы с стиле emacs. Однако это лишь один из возможных режимов, тот, который принят по умолчанию. При желании ничего не стоит переключить навигацию и управление в режим vi, если таковой кажется более привычным для пользователя.

Оболочка zsh обладает всеми стандартными средствами доступа к буферу истории команд: перехода к началу и концу буфера истории, просмотра вперед и назад (как клавишами управления курсором, так и соответствующими управляющими последовательностями), обычного и наращиваемого поиска в обоих направлениях, исполнения выуженной из буфера команды с автоматическим переходом к следующей. Плюс весьма изощренные способы вывода в строку отдельных фрагментов команд из буфера истории, например, отдельного командного "слова", начиная с последнего, с дальнейшим перебором "слов" буферизованных команд назад. Или вывод полного списка команд из буфера с их последовательным перебором в том или ином направлении.

И, наконец, такое мощное средство минимизации пользовательского ввода, как псевдонимы команд (aliases). Разумеется, в zsh (как и в bash или tcsh) псевдоним может быть присвоен любой команде со сколь угодно длинным набором опций. Так, куда как проще раз и навсегда определить команду `ls`

как псевдоним самой же себя, но с опциями `-FG`, нежели каждый раз вспоминать, как отличить в ее выводе каталоги от обычных файлов.

Однако zsh идет дальше: в нем это дополняется возможностью определения псевдонимов для командных конвейеров в форме опции `-g` (от global aliases, именно так именуется эта возможность). Так, всем известно, что для обеспечения постраничного вывода любой команды (например, той же `ls`) вывод этот нужно передать по конвейеру (pipe) программе `less` или `more`. Однако не лениво ли каждый раз вводить что-нибудь вроде

```
$ ls | less
```

да еще и не забывать это делать? Если лениво, в zsh на помощь придут глобальные псевдонимы. Опять же, раз и навсегда определяем, что опция `-g` со значением `L` есть псевдоним для конвейера `'| less'`:

```
$ alias -g L='| less'
```

Теперь имеем возможность, указывая ее после команды, требующей постраничного ввода, именно его и получать:

```
$ ls -g L
```

Казалось бы, не намного проще? Нет, ведь ничто не препятствует нам создать еще один, обычный, псевдоним для команды `ls` с этой опцией, например:

```
$ alias lp='ls -g L'
```

и при необходимости постраничного вывода списка каталога именно к нему и прибегнуть.

Важный момент облегчения существования пользователя в любом шелле — вид приглашения командной строки, должная настройка которого может часто избавить от лишнего набора команд (уж от команды `pwd` я по возможности стараюсь избавиться именно таким образом). Так вот, zsh поддерживает несколько независимо настраиваемых обычных приглашений: обычное, или первичное, вторичное — для многострочных команд, "выделенное", приглашение при выводе вариантов автокоррекции и даже специальное "приглашение" в правой части командной строки. Которое, конечно, собственно приглашением не является, но позволяет вывести полезную информацию, например, текущее время или дату, номер виртуальной консоли, и т. д. Для пущей экспрессии настраивается также передача символов в любом приглашении: цветом ли, выделением, инверсией, подчеркиванием.

Очень полезная возможность — различение вида приглашения для обычного пользователя, получившего права root в результате команды `su` без опций (то есть без перечитывания профильных файлов администратора), от собственно пользовательского приглашения.

Надеюсь, что мне удалось убедить читателя в превосходных интерактивных возможностях оболочки zsh. Теперь стоит поговорить о функциональности, которая проявляется не только в интерактивной работе (но и, скажем, при сочинении скриптов). Функциональность же любой оболочки можно в первом приближении оценить по количеству встроенных в нее команд. То есть команд, выполняемых внутри самого шелла, без порождения новых процессов, как это происходит при выполнении внешних команд. Очевидно, что такие встроенные команды будут выполняться быстрее и отъедать меньше ресурсов. Что, конечно, не скажется при интерактивной работе на современных машинах, но в сложных сценариях — вполне может.

В zsh поддерживается весь набор встроенных команд, стандартизированный для POSIX-shell, большинство команд из развитых оболочек bash и tcsh, ну и, разумеется, специфичные для этого шелла команды. Общее число их превышает 80 — примерно столько же, сколько встроено в tcsh и bash, вместе взятые.

Очень интересна (и удобна) в zsh работа с командными конструкциями перенаправления. Здесь и множественное перенаправление вывода, когда результат выполнения команды направляется сразу в несколько файлов, и множественное перенаправление ввода, когда команда, напротив, получает аргументы последовательно из более чем одного файла, перенаправление без команды, когда конструкция типа

```
$ < filename
```

просто выведет на экран содержимое указанного файла, без привлечения команд типа `cat` или `less`.

При перенаправлении возможна группировка команд по шаблону. Так, файлы с именами вида file1 и file2 можно просмотреть одной командой

```
$ < file{1,2}
```

Перенаправление ввода/вывода может иногда заменять конвейеризацию команд. Так, конструкция вида

```
$ sort < file{1,2}
```

отсортирует содержимое обоих файлов точно так же, как это сделал бы конвейер команд

```
$ cat file1 file2 | sort
```

Наконец, еще одна специфическая особенность zsh — предисполняемая модификация команд (precommand modifier), осуществляющаяся перед их интерпретацией. Именно таким образом можно отменить чрезмерно навязчивую автокоррекцию аргументов для одной отдельно взятой команды, например, копирования:

```
$ nocorrect cp file1 file2
```

Легко видеть, что все изобилие возможностей zsh далеко выходит за рамки стандарта POSIX для командных оболочек. Однако, в подтверждение своего соответствия оному, zsh, наступая на горло собственной песне, способен к эмуляции POSIX Shell, для этого достаточно создать файл /bin/sh как символическую ссылку на исполнимый файл zsh, например:

```
$ ln -s /usr/bin/zsh /bin/sh
```

Впрочем, делать это следует только в случае полной уверенности, что все общесистемные скрипты полностью совместимы с zsh, а такая уверенность может быть только в том случае, если они написаны собственноручно. Я успешно применял zsh в качестве общесистемного шелла. Однако в других дистрибутивах (и тем более во FreeBSD) от этого лучше воздержаться.

Кроме того, имеется и некий режим совместимости с командными оболочками csh-клана (в нем я, впрочем, не разбирался).

Здесь перечислена лишь небольшая часть возможностей оболочки zsh, в частности, я не останавливался на его встроенных функциях, хотя именно они и составляют ту базу, что обеспечивает все описанное (и не описанное) богатство возможностей. Не говорил я и о подгружаемых модулях (по типу plug-ins), а ведь среди последних есть даже собственный FTP-клиент. Ибо для этого потребовалось бы пересказать всю экранную документацию к нему — более дюжины man-страниц общим объемом (в виде сжатого файла gz) свыше 250 Кбайт, плюс официальное руководство с сайта проекта, включающее файл в PDF-формате объемом 260 страниц (к слову, автором этой документации является тот же Петер Стефенсон).

Тем не менее, надеюсь, что я убедил вас в том, что zsh — штука стоящая.

Приступаем к установке

Оболочка zsh стандартно входит в большинство (я бы сказал — во все из мне известных) полнофункциональных дистрибутивов Linux, в качестве портов и пакетов доступна во FreeBSD, NetBSD, OpenBSD и DragonFlyBSD, портирована даже в AtheOS (вернее, в ее современный потомок Syllable). Так что проблем с ее получением быть не должно. Устанавливаем zsh штатным для данного дистрибутива методом и переходим к следующему разделу.

Если же почему-либо zsh в составе дистрибутива не обнаружился, отправляемся на главный сервер проекта (**ftp://ftp.zsh.org/pub/**) или какое-либо его зеркало (список на **http://www.zsh.org**), можно в каталог distfiles FTP-серверов FreeBSD или Gentoo. И качаем в свое удовольствие исходники последней стабильной версии.

Исходники zsh распаковываются и собираются обычным образом: `./configure`, `make`, `make install`, никаких неожиданностей здесь не предвидится. Лично я предварительно, чисто для интереса, поинтересовался бы опциями конфигурирования:

```
$ ./configure --help
```

Из которых не побрезговал бы опцией `--bindir=/bin`, это будет полезно, если zsh будет использоваться как login shell вообще и пользовательская оболочка администратора — в особенности.

В дистрибутивах Linux, предусматривающих автоматическое обновление общесистемных профильных файлов (например, в Gentoo), может возникнуть необходимость в том, чтобы zsh брал свои переменные окружения из какого-либо общего конфига, например, /etc/profile. Для этого при начальном конфигурировании исходников следует указать

```
$ ./configure --enable-zprofile=/etc/profile
```

И еще: ручная сборка zsh может потребоваться в некоторых пакетных дистрибутивах.

Теперь же, установив zsh тем или иным образом, делаем его своим пользовательским шеллом по умолчанию — login shell (одной из команд типа `usermod`, `pw`, `chsh`).

Начинаем настройку

Без этого нам, скорее всего, не обойтись. Дело в том, что в установленном zsh мы имеем шанс не увидеть почти ничего из описанных ранее прелестей — ни развертывания сокращений путей, ни автодополнений опций и аргументов, ни выразительных приглашений командной строки. Перед нами будет безликая строка с именем машины (типа `localhost%`), которая едва-то будет справляться с обычным автодополнением команд и путей (и то не обязательно). Могут возникнуть проблемы даже с вызовом собственной экранной документации `man zsh`. Почему?

Дело в том, что zsh имеет очень богатый набор собственных конфигурационных файлов. Но при установке его эти файлы не помещаются автоматически ни в каталог /etc, ни в домашний каталог пользователя — то есть ни в одно из тех мест, где их можно было бы ожидать. Конечно, совсем без первичных настроек zsh не останется: он прекрасно может воспринимать их из таких общесистемных профильных файлов, как /etc/profile, /etc/login и т. д. Однако, во-первых, для этого он должен быть собран должным образом. А во-вторых, и вести себя при этом он будет почти точно так же,

как и соответствующие оболочки (bash или tcsh, а то и /bin/sh). От одного из этих профильных файлов zsh унаследует и переменные окружения, включая `MANPATH`, почему подчас не сможет найти и свою собственную документацию.

Так что для придания zsh полного блеска следует прибегнуть к его собственным конфигурационным файлам. Правда, сначала придется отыскать их примеры из штатной поставки (должен заметить, что в разных системах они могут обнаружиться в весьма неожиданных местах). Так, во FreeBSD их штатное место каталог /usr/local/share/examples/zsh, в дистрибутиве Gentoo Linux примеры оказываются в /usr/share/doc/zsh-XXX-rX/StartupFiles, в иных вполне могут оказаться где-нибудь в районе /usr/share/zsh и т. д.

Для минимизации времени на поиски выдам секрет: примеры конфигурационных файлов zsh, входящие в штатный комплект, называются zlogin, zshenv и zshrc (возможно, с расширением gz), их нетрудно будет отыскать командой `find`, за пределы каталога /usr файлы эти попасть не должны (даже в том случае, если, как в Gentoo, сам исполняемый файл zsh оказывается в каталоге /bin).

После обнаружения конфигов для сердца вольного есть два пути. Первый путь простой: копируем их в наш домашний каталог в качестве dot-файлов (~/.zlogin, ~/.zshenv и ~/.zshrc соответственно), после чего наслаждаемся жизнью. В ряде случаев этого достаточно, чтобы получить доступ к базовым (но очень даже расширенным, сравнительно с собратьями) возможностям этого шелла.

Разумеется, суперпользователь может скопировать эти файлы и в каталог /etc (без точек в имени), в этом случае они будут определять конфигурацию zsh для всех пользователей, его запускающих.

Второй путь — попытаться разобраться, за что отвечают скопированные файлы и как они устроены. Это понадобится в двух случаях: для идеальной настройки своего нового шелла, и если zsh работает не совсем так хорошо, как вы ожидали.

Первый шаг на этом пути — задаться вопросом, а зачем ему так много конфигов, если другие шеллы спокойно обходятся двумя (а то и одним, как /bin/sh по умолчанию). На это я отвечу, что конфигов в zsh вовсе не много, а очень много: в дополнение к трем примерным в разделе `FILES` его man-страницы можно найти упоминание еще о `zprofile` и `zlogout` (и, соответственно, ~/.zprofile и ~/.zlogout). А в ходе пользования им вы, скорее всего, увидите в своем каталоге еще и такие файлы, как ~/.zcompdump и ~/.zhistory.

Так для чего нам такое богатство? Чтобы разобраться в этом, вспомним о трех видах функционирования любого шелла: неинтерактивном, интерактивном и подвиде последнего — главном пользовательском (login shell). Так вот, файл /etc/zshenv (или ~/.zshenv) считывается при каждом запуске любого экземпляра zsh, независимо от того, происходит он интерактивно или опосредованно. Настройки файла /etc/zshrc (и ~/.zshrc) имеют силу для любого интерактивного запуска zsh. И, наконец, файлы /etc/zlogin и /etc/zprofile оба вместе (как и соответствующая им пара, ~/.zlogin и ~/.zprofile) относятся только к тому экземпляру интерактивно запущенного zsh, который выступает в качестве login shell.

Зачем так сложно? А для того, чтобы можно было гибко (и индивидуально) настроить неинтерактивные, интерактивные и пользовательские экземпляры шеллов. Действительно, очевидно, что на настройку неинтерактивного шелла влияет только содержимое файла /etc/zshenv (и ~/.zshenv), на настройку любого интерактивно запущенного экземпляра — он же, но уже вкупе с /etc/zshrc (и ~/.zshrc), тогда как поведение login shell определяется кумулятивным эффектом всех трех (или даже четырех) их пар.

Хорошо, но зачем же нам два конфига для login shell? — спросите вы меня. Ответ прост: из соображений совместимости с bash и tcsh. Для пояснения вернемся к истории вопроса. В первозданном шелле Борна, как уже говорилось, существовал только один конфиг — /etc/profile (~/.profile) для любых экземпляров шелла. В bash к нему прибавился еще и файл /etc/bashrc (~/.bashrc) для интерактивного использования.

В csh же набор конфигов был совсем иной. Там изначально существовали два конфига: /etc/cshrc (~/.cshrc) на все случаи жизни и /etc/login (~/.login) — в качестве конфигуратора login shell, считываемые именно в таком порядке.

В zsh же, дабы удовлетворить привычки пользователей любых предшествовавших шеллов, были включены оба "логируемых" конфига, причем порядок их считывания был унаследован от каждого из родителей. В результате получилась довольно сложная последовательность при запуске login shell:

```
zshenv -> zprofile -> zshrc -> zlogin
```

Причем каждый конфиг сначала, естественно, считывается из каталога /etc, а затем из домашнего каталога пользователя берется его аналог. Разумеется, если все четыре файла присутствуют (и там, и там). Что, сразу скажем, отнюдь не обязательно. Очевидно, что совместное использование `zprofile` и `zlogin` ни малейшего смысла не имеет. Просто бывшим пользователям bash привычней первая схема запуска login shell, бывшим приверженцам tcsh — вторая. Забегая вперед, замечу, что вообще пользователь может обойтись только одним конфигом в своем домашнем каталоге (например, ~/.zshrc для любого интерактивного экземпляра шелла, ведь login shell также будет инте-

рактивным), а все общие настройки получать из общесистемного конфига (например, /etc/profile). Более того, в дистрибутиве Gentoo Linux именно так поступить лучше всего — к причинам вернусь позднее.

Осталось объяснить смысл остальных dot-файлов из пользовательского каталога. Каковой, впрочем, ясен из названий:

- ~/.zlogout — это сценарий, отрабатываемый при выходе из login shell;
- ~/.zhistory хранит историю команд (это и есть ее буфер);
- ~/.zcompdump — делает то же самое, но для встроенных функций zsh.

Два последних файла возникают (при выполнении некоторых условий) сами собой, и речи о них больше почти не будет.

Собственно конфигурирование

Процесс этот начнем с того, что отделим зерна от плевел, то есть решим: а какие же именно файлы нужно настроить. Для начала я оставил бы в покое все файлы из каталога /etc, вернее, просто не стал бы копировать туда примеры. Почему? Да потому, что во многих дистрибутивах Linux имеются либо предварительно настроенные общесистемные конфиги, либо предусмотрены средства для их автоматического создания и обновления (ниже я продемонстрирую это на примере Gentoo). А во FreeBSD в каталоге /etc принято хранить только профильные файлы общесистемных шеллов (сиречь /bin/csh и /bin/sh), тогда как настройка пользовательского шелла — сугубо личное дело пользователя.

Так что ограничиваемся только нашим домашним каталогом. Однако и здесь многое зависит от ОС и дистрибутива. Так, в user-ориентированных дистрибутивах Linux первый кандидат на удаление — файл ~/.zshenv. Конечно, его можно создать в предельно облегченном виде (например, настройка приемов интерактивной работы здесь абсолютно лишняя). Характерно, что в штатном примере zshenv присутствует только переменная `PATH`. Но ведь ее можно получить из общесистемного /etc/profile, не так ли?

Особенно лишним выглядит ~/.zshenv для пользователей Gentoo. Здесь существует прекрасный механизм установки общесистемных переменных — env-update, автоматически обновляющий главный профильный конфиг /etc/profile. Так что при отсутствии ~/.zshenv все необходимое (в актуальном виде) будет браться оттуда. В присутствии же его zsh откажется от считывания /etc/profile, в результате чего такие переменные, как `EDITOR`, `PAGER` и т. д., придется определять дополнительно.

А вот в дистрибутивах Linux, которые (как, например, CRUX и Archlinux) не имеют средств автоматизации установки общесистемных переменных, файл ~/.zshenv оказывается практически необходимым. И далее я попытаюсь обосновать это (а заодно и продемонстрировать, что в нем может содержаться).

Далее, решаем, а нужен ли нам отдельный конфиг для login shell, и если нужен — то какой из них, ~/.zlogin или ~/.zprofile. Теоретически можно обойтись и без того, и без другого. Уж без последнего — так наверняка, не зря же его нет ни среди штатных примеров, ни среди многочисленных конфигов, которыми поделились с народом активные пользователи zsh (**http://www.posix.ru/download.shtml**). А файл ~/.zlogin у меня, например, есть, хотя содержание его может варьировать от системы к системе.

Не помешает и файл ~/.zlogout, его возможное содержание также будет рассмотрено позднее.

Остается главный (по крайней мере, самый большой) конфиг — ~/.zshrc. Что с ним делать? Это легко уяснить, ознакомившись со штатным примером.

А в примере этом мы найдем и настройку приглашений командной строки, и определение псевдонимов, обычных и глобальных, для конвейеров команд, и определение переменной `cdpath`, и величину буфера истории команд, и многое, многое другое.

Описывать все опции из примера было бы долго и скучно. А потому здесь я остановлюсь только на тех, которые создают удобство при интерактивной работе и определяют внешний вид нашего шелла.

Начнем с последнего, то есть приглашения командной строки. В этом качестве могут использоваться:

- полное или сокращенное имя хост-машины (последнее принято по умолчанию для первичного);
- путь к текущему каталогу в различных формах;
- номер текущей команды в буфере истории или строки в данном сеансе работы;
- имя пользователя или командной оболочки;
- номер текущего терминала;
- дата и время в разных форматах;
- индикация работы от лица суперпользователя);
- любые символы типа стрелок, крышечек, скобочек;
- текстовые сообщения (например, поздравление с началом трудового процесса);
- многое другое (подробности — командой `man zshhmisc`).

Плюс к этому приглашения могут быть оформлены визуально различно: выделением жирным шрифтом (boldface mode) или повышенной яркостью (underline mode), инвертированием текста/фона (standout mode), а также цветами. Все это позволяет добиться максимальной информативности приглашения и его внешней выразительности.

Из опций настройки интерактивности — вспомним о том, что обеспечивает расширенные возможности автодополнения. А обеспечиваются они строками, следующими после комментария:

```
# Setup new style completion system. To see examples of the old style
(compctl
# based) programmable completion, check Misc/compctl-examples in the zsh
# distribution.
```

и имеющими вид

```
autoload -U compinit
compinit
```

Вообще, штатный пример файла zshrc очень неплохо прокомментирован.

Далее опции управления историей команд. Здесь для начала следует определить файл, в котором эта история будет храниться (в примере таковой отсутствует, и никакая история по умолчанию не сохраняется по выходе из сеанса zsh):

```
HISTFILE=~/.zhistory
```

Теперь объем нашей исторической памяти, задаваемый двумя опциями:

```
HISTSIZE=1000
```

определяющей память текущего сеанса, и

```
SAVEHIST=1000
```

устанавливающей размер буфера, сохраняемого в `HISTFILE`. Очевидно, что первую бессмысленно делать больше второй (рекомендуются равные значения).

Для обеспечения наращивания файла истории после каждого сеанса работы (в пределах установленной квоты) определяем:

```
setopt APPEND_HISTORY
```

Объем исторической памяти можно установить любым, однако не стоит расходовать его на повторение дублирующихся команд, ошибочными нажатиями <Enter> в пустой строке и т. д.:

```
setopt HIST_IGNORE_ALL_DUPS
setopt HIST_IGNORE_SPACE
setopt HIST_REDUCE_BLANKS
```

Можно настроить и еще много чего, но на этом пока закончим (некоторые опции будут подробнее рассмотрены в следующем разделе). Потому что наступило время подумать, а что представляют из себя установленные нами опции (типа `compinit` и т. д.)? А представляют они собой по преимуществу встроенные функции из комплекта zsh, которые в изобилии можно обнаружить в каталоге /usr/share/zsh/4.0.6/functions. И именно благодаря этим функциям и удается все настроить.

А еще становится понятным, почему иногда zsh в пакетных дистрибутивах, устанавливаясь из бинарников, ведет себя не вполне подобающим образом (а подчас — и просто неподобающим). Да именно потому, что комплект функций в поставке и их использование в файлах примеров бывыют не вполне сходны (по крайней мере, мне такие случаи встречались). При сборке же из исходников с сайта проекта, насколько мне довелось наблюдать, идентичность гарантирована. Тот же результат, естественно, достигается и в Source Based дистрибутивах (или, скажем, в системе портов FreeBSD).

Вообще говоря, я долго не мог понять, почему пользователи не любят zsh. И в конце концов пришел к выводу, именно потому, что сборщики дистрибутивов подчас не умеют его готовить. Это не только мое мнение: многие пользователи пакетных дистрибутивов свидетельствуют, что установленный из "коробки" zsh в их системах работает весьма криво.

Личный рецепт приготовления

Как-то на одном форуме промелькнул вопрос: а как сделать, чтобы в командной строке zsh клавиши типа <Delete>, <End>, <Home> вели себя нормально (по умолчанию они этого делать не собираются). Однако некоторое чувство дискомфорта преследовало: как же так, какой-то там bash умеет нормально обращаться с клавишами, а любимый zsh — не умеет. А как-то случай представился, и я наконец-то разобрался с клавишами в zsh. О чем с удовольствием рапортую в этом разделе.

Из многочисленных zsh-конфигов я (в настоящее время) использую: ~/.zshenv, считываемый при каждом запуске экземпляра оболочки (интерактивном и неинтерактивном), ~/.zshrc, считываемый при каждом интерактивном ее запуске, и ~/.zlogin, считываемый при каждом запуске zsh в качестве login shell. При этом установка переменных окружения происходит в стиле C-Shell: сначала из ~/.zshenv, затем из ~/.zshrc и, наконец, из ~/.zlogin.

Так что первым в моей схеме идет ~/.zshenv. Он оказывает воздействие только при shell-скриптинге. Поэтому у меня он очень мал.

В Linux-варианте (для Archlinux) он выглядит так:

```
#
# My ~/.zshenv for Linux
#export PATH="/bin:/usr/bin:\
/usr/local/bin:/usr/X11R6/bin:\
/opt/bin:/opt/kde/bin"
```

Во FreeBSD все примерно то же, но, естественно, /opt в `PATH` включать не требуется:

```
#
# My ~/. zshenv for FreeBSD
#
PATH="/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin"
QTDIR=/usr/local/qt
PATH="$PATH:$QTDIR"
MANPATH="$QTDIR/doc/man:$MANPATH"
LD_LIBRARY_PATH="$QTDIR/lib:$LD_LIBRARY_PATH"
export QTDIR PATH MANPATH LD_LIBRARY_PATH
```

Понятно, что оба файла можно записать поизящней (с точки зрения формы), но так уж они исторически сложились (у меня).

Основное внимание я уделяю файлу ~/.zshrc, так как им определяется поведение zsh не только при авторизации в консоли, но и при запуске терминальных окон в Иксах.

```
#
# My ~/.zshrc
#
# Path для поиска командой cd: то есть вместо cd $HOME/docs/editors/
# можно набирать просто cd editors
cdpath=(~/media ~/docs)

## Установка нормального поведения клавиш Delete, Home, End и т. д.:
case $TERM in
       linux)
       bindkey "^[[2~" yank
```

```
        bindkey "^[[3~" delete-char
        bindkey "^[[5~" up-line-or-history
        bindkey "^[[6~" down-line-or-history
        bindkey "^[[1~" beginning-of-line
        bindkey "^[[4~" end-of-line
        bindkey "^[e" expand-cmd-path ## C-e for expanding path of typed
        command
        bindkey "^[[A" up-line-or-search ## up arrow for
        back-history-search
        bindkey "^[[B" down-line-or-search ## down arrow for
        fwd-history-search
        bindkey " " magic-space ## do history expansion on space
        ;;
        *xterm*|rxvt|(dt|k|E)term)
        bindkey "^[[2~" yank
        bindkey "^[[3~" delete-char
        bindkey "^[[5~" up-line-or-history
        bindkey "^[[6~" down-line-or-history
        bindkey "^[[7~" beginning-of-line
        bindkey "^[[8~" end-of-line
        bindkey "^[e" expand-cmd-path ## C-e for expanding path of typed
        command
        bindkey "^[[A" up-line-or-search ## up arrow for
        back-history-search
        bindkey "^[[B" down-line-or-search ## down arrow for
        fwd-history-search
        bindkey " " magic-space ## do history expansion on space
        ;;
esac
# Примечание: если, скажем, в KDE для konsole
# выбрать тип Linux console, необходимости
# во второй секции нет. Консоль Linux в KDE можно
# спокойно установить и во FreeBSD.
# Правда, совсем уж нормального
# поведения клавиш в syscons я до сих пор
# не добился.
```

```
# Use hard limits, except for a smaller stack and no core dumps
unlimit
limit stack 8192
limit core 0
limit -s

# Установка атрибутов доступа для
# вновь создаваемых файлов
umask 022

# Исправление поведения less — для ликвидации
# лишних Esc и прочего безобразия при
# выводе man-страниц.
# Насколько мне известно, нужно только в некоторых
# дистрибутивах Linux
export LESS="-R"

# Установка alias'ов

## alias'ы для команд, не требующих коррекции, но требующих подтверждения

alias mv='nocorrect mv -i'
# переименование-перемещение с подтверждением

alias cp='nocorrect cp -iR'
# рекурсивное копирование с подтверждением

alias rm='nocorrect rm -i'
# удаление с подтверждением

alias rmf='nocorrect rm -f'
# принудительное удаление

alias rmrf='nocorrect rm -fR'
# принудительное рекурсивное удаление
```

```
alias mkdir='nocorrect mkdir'
# создание каталогов без коррекции
## Примечание: если не определить здесь nocorrect,
## zsh будет настойчиво предлагать подстановку
## существующих имен при создании каталогов,
## копировании и т. д.

## Разные полезные (ИМХО) alias'ы
alias h=history
alias grep=egrep

### вывод свободного дискового пространства
### в мегабайтах
alias df='df -m'

### Представление вывода less в more-подобном виде
### (с именем файла и процентом вывода)
alias less='less -M'

### Русский словарь для ispell по умолчанию
alias ispell='ispell -d russian'

## aliases для команды ls<

### показ классификации файлов в цвете и
### символически (Linux)
alias ls='ls -F --color=auto'
### Во FreeBSD достаточно
alias ls='ls -F'

### вывод в длинном формате
alias ll='ls -l'

### вывод всех файлов, включая dot-файлы, кроме . и ..
alias la='ls -A'
```

```
### вывод всех файлов в длинном формате, включая inodes
alias li='ls -ial'

### вывод только каталогов
alias lsd='ls -ld *(-/DN)'

### вывод только dot-файлов
alias lsa='ls -ld .*'

# Установка глобальных псевдонимов
# для командных конвейеров
alias -g M='|more'
alias -g L='|less'
alias -g H='|head'
alias -g T='|tail'
alias -g N='2>/dev/null'

# Ниже даны опции, относящиеся к функциям zsh,
# которыми собственно и определяется мощь этой оболочки

# Shell functions
setenv() { typeset -x "${1}${1:+=}${(@)argv[2,$#]}" }

# csh compatibility
freload() { while (( $# )); do; unfunction $1; autoload -U $1; shift;
done }

# Where to look for autoloaded function definitions

fpath=($fpath ~/.zfunc)

# Autoload all shell functions from all directories in $fpath (following
# symlinks) that have the execu<Tab>le bit on (the execu<Tab>le bit
# is not
# necessary, but gives you an easy way to stop the autoloading of a
```

```
# particular shell function). $fpath should not be empty for this
# to work.
for func in $^fpath/*(N-.x:t); autoload $func

# automatically remove duplicates from these arrays
typeset -U path cdpath fpath manpath

# Указание путей к man-страницам.
## Linux:
manpath="/usr/man:/usr/share/man:\
/usr/local/man:/usr/X11R6/man:/opt/qt/doc"

## FreeBSD:
manpath="/usr/share/man:/usr/local/man:/usr/X11R6/man"

export MANPATH

# Список хостов, к которым будет применяться
# автодополнение при наборе в командной строке
# например, как аргументов браузера или
# FTP-клиента (see later zstyle)
hosts=('hostname' ftp.freebsd.org ftp.archlinux.org)

# Установка вида приглашения

## Обычное приглашение вида ~%=>
## (каталог от домашнего — пользователь/root — стрелка)
PROMPT='%~%#=> '

## Приглашения для второй линии многострочных команд
## вида #_строки>
PROMPT2='%i%U> '

## Приглашение с правой стороны экрана вида
## 19:15 vc/5 (время — номер консоли)
RPROMPT=' %T %y%b'
```

```
# Всякие переменные

## файл истории команд
## если не указан, история не будет сохраняться
## при выходе из сеанса
HISTFILE=~/.zhistory

## Число команд, сохраняемых в HISTFILE
SAVEHIST=5000

## Число команд, сохраняемых в сеансе
HISTSIZE=5000
## Примечание:
## рекомендуются равные значения для
## SAVEHIST и HISTSIZE

DIRSTACKSIZE=20

# Опции истории команд

## Дополнение файла истории
setopt  APPEND_HISTORY

## Игнорировать все повторения команд
setopt  HIST_IGNORE_ALL_DUPS

## Игнорировать лишние пробелы
setopt  HIST_IGNORE_SPACE

## Удалять из файла истории пустые строки
setopt  HIST_REDUCE_BLANKS

# Установка-снятие опций шелла
setopt   notify globdots correct pushdtohome cdablevars autolist
setopt   correctall autocd recexact longlistjobs
```

```
setopt   autoresume histignoredups pushdsilent noclobber
setopt   autopushd pushdminus extendedglob rcquotes mailwarning
unsetopt bgnice autoparamslash

## Отключение звукового сигнала
## при ошибках
setopt  No_Beep

## Не считать Control+C за выход из оболочки
setopt  IGNORE_EOF

# Autoload zsh modules when they are referenced
zmodload -a zsh/stat stat
zmodload -a zsh/zpty zpty
zmodload -a zsh/zprof zprof
zmodload -ap zsh/mapfile mapfile

# Опции общего поведения
# bindkey -v  # режим навигации в стиле vi
bindkey -e    # режим навигации в стиле emacs

bindkey ' ' magic-space    # also do history expansion on space
bindkey '^I' complete-word # complete on <Tab>, leave expansion to
_expand

# Для разворота сокращенного ввода типа cd d/e в docs/editors
autoload -U compinit
compinit

# Completion Styles

# list of completers to use
zstyle ':completion:*:::: ' completer _expand _complete _ignored
_approximate

# allow one error for every three characters typed in approximate
completer
```

```
zstyle -e ':completion:*:approximate:*' max-errors \
'reply=( $(( ($#PREFIX+$#SUFFIX)/3 )) numeric )'

# insert all expansions for expand completer
zstyle ':completion:*:expand:*' tag-order all-expansions
# formacodeing and messages
zstyle ':completion:*' verbose yes
zstyle ':completion:*:descriptions' format '%B%d%b'
zstyle ':completion:*:messages' format '%d'
zstyle ':completion:*:warnings' format 'No matches for: %d'
zstyle ':completion:*:corrections' format '%B%d (errors: %e)%b'
zstyle ':completion:*' group-name ''
# match uppercase from lowercase
zstyle ':completion:*' matcher-list 'm:{a-z}={A-Z}'
# offer indexes before parameters in subscripts
zstyle ':completion:*:*:-subscript-:*' tag-order indexes parameters

# command for process lists, the local web server details and host
# completion
#zstyle ':completion:*:processes' command 'ps -o pid,s,nice,stime,args'
#zstyle ':completion:*:urls' local 'www' '/var/www/htdocs' 'public_html'
zstyle '*' hosts $hosts

# Filename suffixes to ignore during completion (except after rm command)
zstyle ':completion:*:*:(^rm):*:*files' ignored-pacodeerns '*?.o' '*?.c~'
\
'*?.old' '*?.pro'
# the same for old style completion
#fignore=(.o .c~ .old .pro)

# ignore completion functions (until the _ignored completer)
zstyle ':completion:*:functions' ignored-pacodeerns '_*'

# Флаги оптимизации для gcc
CFLAGS="-O2 -march=pentium4 -pipe"
```

```
CXXFLAGS="$CFLAGS"
BOOTSTRAPCFLAGS="$CFLAGS"
export CFLAGS CXXFLAGS BOOTSTRAPCFLAGS
```

И, наконец, файл ~/.zlogin. Что осталось неохваченным в ~/.zshrc и требуется только при авторизации в системе? Правильно, пользовательские переменные для определения терминала, редактора, пейджера и т. д.

```
#
# My ~/.zlogin for FreeBSD
#
from 2>/dev/null
EDITOR=joe
PAGER=less
TERM=${TERM:-cons25r}
export EDITOR PAGER TERM
```

Кроме того, в Linux здесь же резонно установить локаль (во FreeBSD локаль лучше определять через класс пользователя). И потому для Linux дополнение:

```
#
# My ~/.zlogin appendix for Linux
#
# Установка всех локально-зависимых переменных,
# кроме LC_ALL
export LANG="ru_RU.koi8r"

# Установка десятичной точки
# вместо запятой
# (требуется для некоторых счетных программ)
export LC_NUMERIC="POSIX"
```

Да, самое последнее, файл ~/.zlogout, отрабатываемый по завершении сеанса пользовательского шелла. У меня он включает две строчки:

```
sync
clear
```

Назначение их более чем понятно (синхронизация дисковых кэшей и очистка экрана).

Это мои пользовательские конфиги. Почти те же самые я использую и для root, с минимальными коррективами. Так, переменная `path` в /root/.zshenv дополняется значениями

```
/sbin:/usr/sbin:/usr/local/sbin
```

В /root/.zshrc опция `cdpath` имеет вид

```
cdpath=(/etc /usr)
```

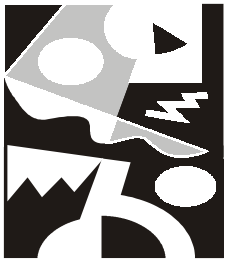
Кроме того, во FreeBSD здесь я исключаю переменные `CFLAGS`, `CXXFLAGS` и `BOOTSTRAPCFLAGS`, так как от лица root все собирается через систему портов, где их аналоги определяются в файле /etc/make.conf.

А в /root/.zlogin локаль (в Linux) установлена как

```
export LANG="POSIX"
```

Некоторые программы упорно не желают собираться при какой-либо иной.

Глава 16



Икс — он и в Африке X

- Кто вы, мистер Икс?
- Иксы: принципы организации.
- Иксы: сборка из исходников.
- Варианты конфигурирования.
- Варианты запуска.
- Немного о раскладках.
- Шрифты: полиграфическое введение.
- Разборки со шрифтами.

Если текстовый режим работы в каждой POSIX-совместимой операционной системе обеспечивается ее собственным консольным драйвером, то за режим по-настоящему графический в любой из них отвечает (почти) одна и та же программа — оконная система X, или в просторечии Иксы.

Кто вы, мистер Икс?

Иксы часто воспринимаются как неотъемлемая часть Linux. Однако на самом деле к Linux они не имеют никакого отношения, за исключением того, что могут быть запущены поверх него. Как, впрочем, поверх Free- или OpenBSD, а также любого UNIX и UNIX-подобной системы. И не только: помнится, в свое время Иксы были портированы на OS/2. А еще приходилось слышать, что существует и Windows-реализация каких-то X-серверов. Потому что, вообще говоря, Иксы изначально были разработаны для того, чтобы запускаться в любой ОС, способной поддерживать растровую графику.

К слову, о названии. В оригинале, то есть по-английски, система эта называется X Window System, наиболее точным переводом чего будет "оконная система X". Ну, что это система, интуитивно понятно. Прилагательное "оконная" призвано отличать ее от полноэкранных графических сред (были ведь и такие; в начале 90-х каждая уважающая себя DOS-программа считала своим долгом обзавестись собственным графическим интерфейсом, функционирующим, как правило, именно в полноэкранном режиме).

И, наконец, X — это имя собственное. Непосредственно по времени ей предшествовала другая графическая система, носившая имя W (от слова Window, ибо также была оконной). Так что те, кто еще не забыл латинский алфавит, легко могут понять происхождение литеры X в названии системы. И потому попадающиеся в литературе псевдопереводы типа система X Window (а подчас даже система X Windows) гораздо хуже передают суть дела, чем краткое жаргонное словечко — Иксы.

История Иксов уходит в седую древность — 1984 г., и у истоков ее трое разработчиков: Роберт Шейфлер (Robert Sheifler), Джим Геттис (Jim Gettys) и Рон Ньюмен (Ron Newman). В дальнейшем разработкой Иксов занимался Массачусетский технологический институт (MIT) и фирма DEC. В 1987 г. разработчики Иксов, при участии ряда компьютерных фирм, создали организацию под названием X Consortium, которая приступила к распространению оконной системы X открыто и свободно, под собственной лицензией, сходной с GPL и особенно BSD.

За время жизни оконная система X сменила множество версий, из которых последней на долгие годы (и по сей день) стала одиннадцатая. Видоизменения внутри которой получали статус релизов, нынешний из которых — шестой, почему вся система сокращенно и называется X11R6. Наконец, внутри релиза вариации также нумеруются, и потому на сей момент последний вариант Иксов носит такой весьма запутанный титул: X11R6.9.

Сами по себе Иксы — это не какая-либо конкретная программа, и даже не графический интерфейс как таковой, а лишь набор спецификаций, которым этот самый графический интерфейс (точнее, даже метаинтерфейс, как скоро станет ясно, никаких средств взаимодействия с пользователем сами по себе Иксы не содержат) должен соответствовать. Конкретная же их реализация целиком на откупе (и на совести) разработчиков конкретных же систем. И потому на базе Иксов было построено множество самостоятельных графических метаинтерфейсов, большинство из которых проприетарные, закрытые и распространяются за деньги (подчас немалые). Среди таких реализаций мне известны Metro-X и Accelerated-X: они использовались в проприетарных же UNIX-системах. Хотя и подборки Linux, выпускавшиеся известной компанией Walnut Creek, одно время комплектовались некоторыми версиями коммерческих серверов.

Однако пользователь Linux, FreeBSD имеет дело, как правило, только со свободными реализациями же X-спецификаций. До недавнего времени здесь следовало употребить единственное число, потому что такая реализация существовала в единственном экземпляре. И имя ей было (да и по сей день есть) XFree86, что, как несложно догадаться, означало: свободные Иксы для платформы x86, разрабатывающиеся в рамках одноименного проекта (**http://www.xfree86.org**). Она также имеет собственную лицензию, которую адепты FSF до недавнего времени полагали "правильной". Однако выход релиза XFree86 4.4 в начале 2004 г., сопровождавшийся сменой лицензионной политики, изменил ситуацию: теперь она полагается злостным порождением (ну, не знаю чего — мирового империализма, вероятно), и в звании GPL-совместимой ей отказано.

В связи с этим в народе возник новый проект, именуемый Xorg (**http://www.x.org/**). Он основывается на последней бета-версии XFree86 4.39, которая имела еще "правильную" лицензию и в целом соответствует этой версии по содержанию и развивает ее традиции.

Похоже, однако, что отделение проекта Xorg всем пошло на пользу. В том числе и проекту XFree86, который активизировал свою деятельность, выпустив весной 2005 г. версию 4.5, содержащую множество усовершенствований. Впрочем, как официальный компонент дистрибутива она вошла только в NetBSD.

Таким образом, на данный момент мы имеем несколько неопределенную ситуацию — сосуществование трех свободных реализаций Иксов:

- "неправильной" XFree86 4.5, которую, тем не менее, многие разработчики как Linux-дистрибутивов, так и BSD-систем включают, в той или иной форме, в свои поставки;
- "правильной" Xorg (последние версии на момент данного сочинения X11R6.9 и X11R7, различающиеся лишь политикой распространения), которая медленно, но верно внедряется в качестве "Иксов по умолчанию";
- старой доброй (хотя на счет последнего существуют и иные мнения) XFree86 4.3, сохраняемой как компонент консервативных систем (FreeBSD — пример которых).

Мне не хотелось бы заниматься юридическим крючкотворством. Хотя новую лицензию XFree86 4.4, в меру своего понимания вопроса, прочел и ни малейшей крамолы в ней не обнаружил.

Так что далее речь пойдет о неких абстрактных Свободных Иксах (или Иксах просто), тем паче, что, по моим наблюдениям, принципиальных различий между XFree86 и Xorg не просматривается. Если не считать, конечно,

разных названий исполняемых и конфигурационных файлов, так что в необходимых местах я буду делать соответствующие оговорки.

Иксы предназначены для установки на любой из свободных UNIX-клонов (и не только свободных, те же самые Иксы используются в Solaris для платформы Intel, возможно, и в других проприетарных UNIX). Важно понимать, что когда речь идет об Иксах в каком-либо дистрибутиве Linux или любой ОС из клана BSD, мы имеем дело с одним и тем же исходным кодом, а мелкие различия между ними обусловлены либо номером версии, либо именем реализации, либо особенностями сборки под конкретную платформу, либо, наконец, предпочтениями конкретного дистрибутива. И потому установка и настройка Иксов в любой из этих систем осуществляется одними и теми же методами.

Пользователь любого дистрибутива Linux, Free-, Net- или любой иной BSD всегда может обратиться к универсальным средствам. О них-то, в первую очередь, и пойдет разговор в последующих разделах этой главы.

Иксы: принципы организации

Что же представляют собой Иксы? Одно из определений (заимствованное из статьи Андрея Зубинского на **http://www.i2r.ru/static/278/out_9441.shtml**) гласит, что это не зависящая от платформы растровая графическая оконная система, обеспечивающая прозрачное использование ее ресурсов с помощью сетевых соединений. Определение верное, но вызывающее больше вопросов, чем дающее ответов. Их мы и попробуем наметить.

Независимость от платформы, пожалуй, в объяснениях не нуждается: как я уже говорил, Иксы функционируют на любом "железе" и поверх любой ОС. Единственное требование — способность видеосистемы данной платформы хоть как-то воспроизводить графику. Причем графику растровую: растровый характер Иксов определяется тем, что они работают только с устройствами вывода, обеспечивающими пиксельное отображение информации, а не символьное, как текстовая консоль, и не векторное (впрочем, векторных видеосистем я никогда не видел, хотя говорят — есть и такие). Оконная же она потому, что имеет более высокий, нежели пиксел, уровень абстракции — "окно" (window) в основе всех интерфейсных элементов: и собственно окон, и кнопок, и выпадающих меню, и строк ввода, и даже обрамления вокруг окон.

Сетевой характер системы X Window обусловлен ее моделью — клиент-серверной, в которой взаимодействие между сервером и клиентами осуществляется (даже на локальной машине) за счет специального сетевого X-протокола. По этому протоколу происходит также любое взаимодействие

между клиентскими программами Иксов и их связь с главной Иксовой библиотекой — Xlib.

Сервер и клиент — первые из базовых понятий системы X. При этом понимание их кажется обратным общепринятому: в качестве сервера выступает аппаратно-зависимая программа ввода/вывода, непосредственно взаимодействующая с "железом" — видеоустройствами, клавиатурой и мышью. Она так и называется — X-сервер, и именно в ней кроются главные различия между различными реализациями Иксов. X-сервер предоставляет свои ресурсы клиентам — прикладным программам, обеспечивающим остальные функции. То есть сервером в общем случае является локальный терминал, тогда как клиентская часть может исполняться и на удаленной машине.

Впрочем, для персонального компьютера это не принципиально. Более важным здесь оказывается другое базовое понятие — дисплей. Под которым в данном случае понимается отнюдь не экран монитора, вернее, не только он один, но его комбинация с устройствами ввода и позиционирования, клавиатурой и мышью. В случае персонального компьютера такой физический X-дисплей представляет собой просто нашу наличную машину. Хотя, теоретически рассуждая, к одной машине можно прикрутить несколько дисплеев, благо, двухмониторные карты нынче не редкость, и USB-разъемов для подключения дополнительных клавиатур и мышей обычно также в достатке.

Маленькое отступление

Очевидно, что для открытия дисплея (то есть запуска на нем X-сервера) последний должен взаимодействовать с аппаратурой, его составляющей, быть совместимым с нею. И если мыши и особенно клавиатуры давно стандартизированы, и сложностей здесь обычно не бывает, то многообразие видеокарт до недавнего времени часто оказывалось камнем преткновения при использовании Иксов. Именно в области поддержки видеоадаптеров и различаются между собой, в первую очередь, отдельные их реализации. И сильной стороной коммерческих X-серверов был как раз широкий спектр поддерживаемого видеооборудования. Ныне все существующие видеокарты практически одинаково хорошо поддерживаются всеми X-серверами, в том числе и свободными. Тем не менее, нужно понимать: когда говорят о поддержке, например, в Linux (или даже в конкретном его дистрибутиве) той или иной видеокарты, имеется в виду именно совместимость с Иксами, а не с операционной системой как таковой.

Один и тот же физический X-дисплей в состоянии воспроизводить некоторое количество дисплеев виртуальных — аналогов виртуальных консолей текстового режима. Каждый из них целиком занимает собственную виртуальную консоль, поэтому для запуска Иксов необходимо, чтобы хотя бы одна из таковых была доступна (для ядра ОС) и неактивизирована (процессом типа `getty`). Общее количество X-дисплеев для 32-разрядных компьютеров теоретически составляет 232, нумеруются они с 0.

Каждый виртуальный дисплей может содержать множество окон — отображаемых областей растровой памяти. Окна образуют строгие иерархии на основании простого правила: в каждом экране по умолчанию существует одно уникальное окно-родитель (корневое, или root-окно), каждое окно должно иметь родителя и само может быть родителем для других окон. Окна могут создаваться, перерисовываться, удаляться, в них может выводиться текстовая и графическая информация.

Между окнами возможен обмен данными через буферы памяти; в отличие от консольного режима, в Иксах таких буферов задействовано несколько. Один из них обычный экранный буфер, подобный консольному: информация в него помещается при выделении мышью или стрелками клавиатуры и вставляется нажатием средней кнопки мыши. Остальные буферы памяти требуют принудительного копирования в них выделенного фрагмента — например, привычной по Windows комбинацией клавиш <Ctrl>+<C> или <Ctrl>+<X>, после чего вставляется также каким-либо специальным способом (например, через <Ctrl>+<V>). Однако это уже зависит от реализации в конкретных программах. К сожалению, способов обмена данными между X-дисплеем и текстовыми консолями не существует (по крайней мере, я никаких указаний на этот счет не нашел).

Кроме этого, X-сервер обеспечивает рендеринг шрифтов — претворение их формального описания из специальных шрифтовых файлов разного формата в те самые буквы, которые мы видим на экране. Впрочем, эта функция может быть возложена на отдельный шрифтовой сервер (X Fonts Server).

Функции X-сервера осуществляются за счет специальной библиотеки Xlib. Она обеспечивает открытие дисплея и взаимодействие с ним любой клиентской программы, создание и уничтожение окон, а также их отображение вместе с некоторыми простыми атрибутами.

Важно, что сам по себе X-сервер (вместе с Xlib) никоим образом не способен управлять элементами пользовательского интерфейса — даже собственными окнами, которые он воспроизводит. Забегая вперед, замечу: чтобы получить представление о возможностях чистых Иксов (точнее, отсутствии таковых), можно для пробы запустить только X-сервер. Делается это командой (из текстовой консоли)

```
$ X
```

Она представляет собой символическую ссылку на исполняемый файл X-сервера /usr/X11R6/bin/XFree86 или /usr/X11R6/bin/Xorg (для соответствующих реализаций). После чего можно наблюдать за перемещением крестообразного курсора по серо-решетчатому фону — и ничего более. Правда, есть еще и возможность прервать X-сеанс комбинацией клавиш <Alt>+ +<Ctrl>+<BackSpace>; к слову сказать, это универсальный способ выхода из

графического режима, к которому приходится прибегать, если штатные средства выхода почему-либо не работают.

Для использования мощи X-сервера в мирных целях он должен быть запущен совместно с какой-нибудь клиентской программой. В простейшем случае такой программой может быть так называемый эмулятор терминала, который, таким образом, оказывается непременным компонентом оконной системы X (в комплект ее входит его разновидность под названием xterm). Впрочем, часто *X-терминал* обозначает и аппаратное решение — слабые сетевые рабочие станции, служащие только для запуска X-сервера, а для работы клиентских программ использующие ресурсы мощных компьютеров, выступающих в качестве серверов приложений.

Работа в окне X-терминала полностью имитирует действия в консоли: здесь запускается отдельный экземпляр пользовательского шелла, позволяющего вводить команды, в том числе и фоновые, и запускать любые программы текстового режима (например, редактор, файловый менеджер, браузер), которые будут исполняться в том же окне.

Позволяет терминал запускать и программы графического режима, специально предназначенные для работы в оконной системе X. Такие программы, в отличие от консольных, будут исполняться в собственном, отдельном от терминального, окне. А поскольку Иксы наследуют многозадачный режим, то таких программ можно запустить много, пользуясь, например, средствами запуска в фоновом режиме командной оболочки. И каждая запущенная программа будет исполняться в отдельном окне. Однако средств управления окнами ни сами Иксы, ни тем более терминал не предоставляют, так что даже переключаться между такими задачами окажется затруднительно.

Таким образом, чтобы в полной мере воспользоваться возможностями, предоставляемыми Иксами совместно с базовой ОС, потребуется еще одна программа, которая осуществляет управление окнами, их открытием, закрытием, представлением на экране, масштабированием, перемещением, фокусировкой и т. д. Почему она и называется менеджер окон и являет собой графический интерфейс пользователя (GUI, Graphic User Interface) в полном смысле слова: вид всех управляющих элементов (рамок окон и их заголовков, полос прокрутки, меню, кнопок и пиктограмм) определяется именно оконным менеджером.

В комплект Иксов штатно входит оконный менеджер под названием twm. Это весьма архаичная программа с ограниченными возможностями. А вообще менеджеров окон существует бессчетное множество, и специальный разговор о них впереди. Здесь они упомянуты лишь потому, что без них, в сущности, невозможно не только практическое использование Иксов, но даже и иллюстрация их возможностей или проверка правильности настройки.

Кроме этого, в комплект Иксов входит множество клиентских программ — утилит и приложений. Одни из них (xvidtune, xfontsel) широко используются в настроечных целях, другие (например, текстовый редактор xedit) давно вытеснены из обихода более совершенными аналогами и сохраняются в качестве реликтов.

Впрочем, и сами Иксы на фоне таких графических систем, как Windows и особенно Mac OS, производят впечатление реликта эпохи. Эпохи мирного сосуществования множества аппаратных платформ и операционных систем, для взаимодействия с которыми Иксы и разрабатывались. Ибо понятно, что за такой универсализм приходится расплачиваться, в первую очередь, потерей производительности и требовательностью к ресурсам, главным образом к памяти. Ибо хотя сам по себе X-сервер — программа достаточно неприхотливая и готова довольствоваться 16 Мбайт суммарной (физической и виртуальной) памяти, уже самый простой оконный менеджер с несколькими пользовательскими приложениями эти требования как минимум удваивает или утраивает. И при этом быстродействие их, в сравнении с Windows-аналогами, отнюдь не поражает воображения.

Мощные интегрированные среды, такие как KDE и сложные комплексные приложения типа OpenOffice, Mozilla, GIMP, начинают более-менее шевелиться при объемах физической памяти от 256 Мбайт. С увеличением объема памяти сверх этого и быстродействие "тяжелых" приложений возрастает, достигая оптимума при 512 Мбайт (правда, по современным масштабам это не кажется чем-то из ряда вон выходящим).

Не способствует повышению быстродействия и сама клиент-серверная модель построения Иксов, в которых каждое взаимодействие между процессами имитируется сетевым соединением. А если вспомнить еще и о незащищенности X-протокола, то модель эта создает определенную угрозу безопасности. Не случайно одна из настоятельных рекомендаций в этом отношении — никогда не запускать на сетевой машине Иксы от лица суперпользователя.

Есть у Иксов и другие недостатки. Один из важнейших с точки зрения пользователя — низкая скорость и качество рендеринга шрифтов. Это в сочетании с из рук вон плохим качеством самих шрифтов, особенно кириллических, дает подчас эффект просто ужасающий. Тем не менее, Иксы использовались, используются и будут использоваться во всех POSIX-системах еще долгое время (возможно, вечно). По двум причинам. Во-первых, другой графической системы у нас, POSIX'ивистов, нет. А во-вторых, многочисленные недостатки Иксов компенсируются полной изоляцией графической среды от базовой ОС и ее системного окружения, с одной стороны, и клиентских программ от X-сервера — с другой. В результате этого ошибки исполнения пользовательских приложений крайне редко приводят к краху са-

мих Иксов и практически никогда — всей операционной системы. И потому Иксы в сочетании с применяются и будут применяться везде, где критичной оказывается бессбойная работа системы. А это все машины производственного назначения, не так ли?

Иксы: сборка из исходников

Знакомство с Иксами начинается с их установки. Собственно, никто не понуждает пользователя выполнять эту процедуру вручную, в большинстве пакетных дистрибутивов они устанавливаются автоматически, да еще обычно по умолчанию, при первичной инсталляции. Однако при этом они подчас тянут за собой столько разного и непонятного, что возникает резонное желание разобраться, что из этого софта действительно необходимо, а что относится к архитектурным излишествам. А для этого необходимо хоть раз установить Иксы самому.

Сделать это можно из прекомпилированных пакетов, таковые входят в состав любого полнофункционального дистрибутива, руководствуясь методами принятого в данной системе пакетного менеджмента (rpm там, apt-get, порты FreeBSD или портежи Gentoo и т. д.). Если версия из дистрибутива окажется недостаточно свежей, более свежую бинарную сборку XFree86 можно получить с сервера проекта: **ftp://ftp.xfree86.org/pub**. В частности, нынче это основной способ получения XFree86 для тех систем, которые официально от этой реализации Иксов отказались.

Лучше собрать Иксы из исходников, да еще в соответствии с собственными представлениями об оптимизации. Тем более что это единственный пока способ ознакомиться с новейшими версиями "правильных" Иксов от Xorg, если они не успели попасть в дистрибутив, в бинарном виде эта реализация на сайте проекта недоступна, распространяясь только в виде исходных текстов.

Так вот, оказывается, что для установки Иксов необходимо и достаточно иметь архивы их исходных текстов. Ну и, конечно, Base Linux — Иксы прекрасно ставятся на чистый LFS Герарда Бикманса. Или, например, на минимальный установочный комплект FreeBSD или DragonFlyBSD.

В современном своем виде исходники Иксов разбиты на семь тарбаллов. В реализации от XFree86 они имеют вид XFree86-4.5.0-src-[1-7].tgz, в варианте от Xorg — X11R6.8.2-src[1-7].tar.gz, суммарным объемом более 50 и 70 Мбайт соответственно. Только они и необходимы, прочее, что можно найти в каталоге src на серверах проекта (утилиты, документация) относится к категории опционального. Так что скачиваем тарбаллы, переходим в ката-

лог, предназначенный для исходников, и развертываем архивы любым удобным способом. Например, вот так:

```
$ find /path_to_src -name XFree86-4.5.0-src-?.tgz \
       -exec tar xzpvf {} \;
```

После чего все исходники оказываются в одном каталоге xc. Переходим в него и внимательно читаем файл INSTALL-X.org (BUILD в реализации от Xorg), по крайней мере, его начало. Из которого выясняется, что установка Иксов на самом деле очень проста и сводится к двум основным командам

```
$ make World
$ make install
```

и одной дополнительной (опциональной):

```
$ make install.man
```

Вывод этих команд к тому же можно перенаправить в файл, например:

```
$ make World >& world.log
```

Освободили тем самым командную строку нашей консоли для общественно-полезной деятельности. Это отнюдь не лишнее, сборка Иксов даже на мощной машине может занять не один час.

Возникает резонное желание предварительно задать некоторые условия оптимизации. Заметим сразу, что обычные флаги gcc, заданные в переменной `CFLAGS` (в командной строке или в общем профильном файле), не окажут на процесс компиляции никакого влияния. Чтобы сборка проходила с оптимизацией, флаги эти нужно определить в переменной `BOOTSTRAPCFLAGS`:

```
make World BOOTSTRAPCFLAGS="значения"
```

Я, например, обычно определяю ее так:

```
BOOTSTRAPCFLAGS="$CFLAGS"
```

в sh-совместимых оболочках и

```
setenv BOOTSTRAPCFLAGS $CFLAGS
```

в tcsh. А значения переменной `CFLAGS` заданы в профильном файле так:

```
export CFLAGS="-O2 -march=pentium4 -fPIC"
```

или

```
setenv CFLAGS "-O2 -march=pentium4 -fPIC"
```

для zsh и tcsh соответственно. Разумеется, процессор должен соответствовать объявленному во флаге `-march`. Обратим также внимание на флаг `-fPIC`. Он необходим только в том случае, если в дальнейшем исполняемые файлы и библиотеки предполагается подвергнуть предварительному связыванию программой prelink (в Linux). Если таковое не предвидится (или невозможно, как в BSD-системах), его можно опустить.

Вот, собственно, и все. Не пройдет и часа (двух, трех, в зависимости от мощности машины), как вы станете счастливым обладателем работоспособной и полнофункциональной системы Икс. И, повторяю, ничего, кроме перечисленного (то есть семи архивов и толики свободного времени), для этого не потребуется.

Варианты конфигурирования

Немедленный запуск установленных Иксов стопроцентно окончится неудачей: прежде всего их необходимо сконфигурировать, то есть описать в специальном файле параметры аппаратуры: устройств ввода (мыши, клавиатуры) и вывода (видеоадаптера и монитора), а также пути к файлам шрифтов. Файл этот именуется XF86Config (в реализации от XFree86) или xorg.conf (в варианте от Xorg) и местопребыванием своим имеет каталог /etc/X11. Причем в ходе установки Иксов любым способом (из исходников или бинарников) он сам собой не возникает, перед первым запуском Иксов его необходимо каким-либо способом создать и заполнить должным содержанием.

Способов таких несколько.

1. Написать конфиг вручную при знании формата и смысла опций не сложно. Но долго, скучно и, учитывая приличный его объем, чревато элементарными ошибками.
2. Сгенерировать конфиг одной из штатных, специально для этого предназначенных утилит. Их в комплекте также две: xf86config, работающая в диалоговом режиме, и меню-ориентированная xf86cfg, которую можно запустить в графическом (по умолчанию) или текстовом (с опцией `-textmode`) режиме.
3. Можно положиться на возможности автоконфигурирования, которые заложены в обеих современных реализациях X-сервера и с каждой версией становятся все совершеннее.
4. Наконец, есть и четвертый способ — прибегнуть к одной из фирменных утилит конфигурирования, входящих в комплект ряда дистрибутивов Linux (Mandrake, Red Hat). Но поскольку мы занимаемся настройкой сугубо кроссплатформенной системы, да еще в целях самообразования, воспользоваться им было бы просто неприлично.

Так что, отметя с порога первый и последний способы, мы остались перед простым выбором. При этом, как ни странно, третий способ (автоконфигурирование) я не стал бы рекомендовать совсем уж начинающему пользователю. Ибо он потребует в дальнейшем ручной доводки, очень несложной, но подразумевающей наличие определенных знаний, причем в одной из са-

мых важных для пользователя сфер — в области локализации. А вот приобрести эти знания проще всего, выполнив хотя бы один раз конфигурирование с помощью штатной Иксовой утилиты xf86config. Так что с нее мы и начнем, после чего, вооруженные некоторым эмпирическим багажом, обратимся к автоконфигурированию.

Должен предупредить: какой бы способ настройки Иксов мы ни избрали, без некоторой ручной доводки обойтись все равно не получится.

Итак, запускаем диалоговый конфигуратор Иксов из текстовой консоли:

```
$ xf86config
```

Ответом будет сообщение, что

```
This program will create a basic XF86Config file,
based on menu selections you make...
...
```

и т. д., и последует два предложения:

```
Press enter to continue, or ctrl-c to abort
```

Соглашаемся с первым, ведь это и есть наша цель, не так ли? В награду за смелость нам предложат выбрать мышиный протокол. И хотя вариантов выбора довольно много,

```
 1.  Auto
 2.  SysMouse
 3.  MouseSystems
 4.  PS/2
 5.  Microsoft
 6.  Busmouse
 7.  AceCad
 8.  GlidePoint
 9.  IntelliMouse
10.  Logitech
11.  MMHitTab
12.  MMSeries
13.  MouseMan
14.  ThinkingMouse
```

ломать над ними голову особо не стоит, большая их часть относится к устаревшим или редким сериальным и шинным моделям. А для подавляющего большинства современных (с интерфейсом PS/2 или USB) подойдет автоматический подбор (в FreeBSD и DragonFlyBSD он оказывается практически безальтернативным).

Следующий вопрос звучит так:

```
Please answer the following question with either 'y' or 'n'.
Do you want to enable Emulate3Buttons?
```

Если ваш грызун принадлежит к вымирающему семейств двухкнопочных, следует ответить положительно, средняя кнопка мыши в Иксах работает ничуть не менее активно, чем в текстовой консоли. А эмуляция ее осуществляется одновременным нажатием двух наличных кнопок. Однако если, как это обычно бывает, мышь имеет колесо прокрутки, оно прекрасно справится с функциями средней клавиши, и, соответственно, эмулировать ее нет необходимости.

Ответ на вопрос о файле мышиного устройства

```
Mouse device:
```

во FreeBSD (и DragonFlyBSD) также безальтернативен (и должен быть введен руками):

```
/dev/sysmouse
```

Это виртуальное устройство, подменяющее собой любое из реальных мышиных устройств: ни одно из них под Иксами сосуществовать с драйвером консольной мыши (moused) не может. А в Linux в большинстве случаев проходит ответ:

```
/dev/mouse
```

как символическая ссылка на файл реального устройства, вне зависимости от его интерфейса. Лучше, однако, впечатать руками имя последнего, это будет

```
/dev/psaux
```

для мыши с разъемом PS/2, и

```
/dev/input/mice
```

для USB-грызунов.

Теперь предстоят ответы на вопросы о клавиатуре. Сначала определимся с ее типом:

```
Please select one of the following keyboard types
that is the better description of your keyboard.
If nothing really matches, choose 1 (Generic 101-key PC)
```

Выбор здесь обширен (около 90 позиций). Но, вопреки предложенному, в большинстве "настольных" случаев (как, впрочем, и в ноутбуках) лучше всего подойдут варианты

```
Generic 104-key PC
```

или

```
Generic 105-key (Intl) PC
```

Для фирменных клавиш Cherry, Microsoft или Logitech можно попробовать подобрать соответствующие варианты, но это труда не составит. А клавиатуры для ноутбука нужно выбирать только в случае полного соответствия реалиям, при неполном лучше избрать один из стандартных настольных вариантов.

Далее следует выбрать раскладку в соответствии, как задумчиво сказано, со страной:

```
Enter a number to choose the country
```

Не поленимся пролистать список вплоть до появления России, это важно для русификации (хотя от ручной доводки все равно не избавит):

```
60  Russian

61  Russian (cyrillic phonetic)
```

Выбираем 60-й вариант (cyrillic phonetic — штука весьма странная, в стародавние времена применялась на некоторых типах терминалов, ныне сугубо реликтовая раскладка).

```
Please enter a variant name for 'ru' layout.

Or just press enter for default variant
```

Теперь предлагается выбрать (точнее, ввести) вариант раскладки. Единственно приемлемый в современных условиях — winkeys, он соответствует фабричной маркировке ныне продаваемых клавиш. Если же нажать <Enter> для сохранения варианта по умолчанию, получим раскладку для DOS-маркированных клавиатур, кои сейчас найдешь разве что в музее.

Следующий вопрос:

```
Please answer the following question with either 'y' or 'n'.

Do you want to select additional XKB options (group switcher,

group indicator, etc.)?
```

На него очень важно ответить положительно, это позволит довести русификацию клавиатуры почти до ума, а именно, выбрать переключатель раскладок латиницы/кириллицы из весьма длинного списка:

```
Group Shift/Lock behavior


  1  R-Alt switches group while pressed

  2  Left Win-key switches group while pressed

  3  Right Win-key switches group while pressed

  4  Both Win-keys switch group while pressed

  5  Right Alt key changes group

  6  Left Alt key changes group
```

```
 7  Caps Lock key changes group
 8  Both Shift keys together change group
 9  Both Alt keys together change group
10  Both Ctrl keys together change group
11  Control+Shift changes group
12  Alt+Control changes group
13  Alt+Shift changes group
14  Menu key changes group
15  Left Win-key changes group
16  Right Win-key changes group
17  Left Shift key changes group
18  Right Shift key changes group
19  Left Ctrl key changes group
20  Right Ctrl key changes group
```

Понимаю, что в столь интимном деле рекомендации неуместны, но мой твердый (и субъективно обоснованный) выбор — 7-й, переключение по нажатию <CapsLock> (как и в консоли собственно фиксируемое переключение на верхний регистр при этом переходит на комбинацию <Shift>+ +<CapsLock>).

Следующие два вопроса

```
Third level choosers
```

и

```
Control Key Position
```

я всегда игнорирую (для меня это не актуально). А в ответ на предложение индицировать альтернативную (то есть кириллическую) раскладку

```
Use keyboard LED to show alternative group
```

```
 1  Num_Lock LED shows alternative group
 2  Caps_Lock LED shows alternative group
 3  Scroll_Lock LED shows alternative group
```

отвечаю второй позицией (`Caps_Lock LED`), чтобы не путаться, так как `Scroll_Lock LED` во FreeBSD и DragonFly задействуется в консоли для режима пролистывания буфера экранной истории.

Следующие два вопроса

```
CapsLock key behavior
```

и

```
Alt/Win key behavior
```

также игнорируем (если у вас, конечно, есть на него осмысленный ответ — отвечайте). После чего плавно переходим к настройке видеосистемы нажатием на <Enter>, согласившись с таким предложением

```
Now we want to set the specifications of the monitor.
The two critical parameters are the vertical refresh rate,
which is the rate at which the the whole screen is refreshed,
and most importantly the horizontal sync rate,
which is the rate at which scanlines are displayed.

The valid range for horizontal sync and vertical sync
should be documented in the manual of your monitor.
If in doubt, check the monitor database
/usr/X11R6/lib/X11/doc/Monitors
to see if your monitor is there.

Press enter to continue, or ctrl-c to abort.
```

Правда, предварительно не худо вооружиться документацией на имеющийся в наличии монитор. Потому что ответ на первый же вопрос, о частоте строчной развертки,

```
You must indicate the horizontal sync range of your monitor.
```

следует искать именно там. Или, если документация давно потеряна, согласиться с одним из предлагаемых вариантов:

```
1  31.5; Standard VGA, 640x480 @ 60 Hz
2  31.5 — 35.1; Super VGA, 800x600 @ 56 Hz
3  31.5, 35.5; 8514 Compatible, 1024x768 @ 87 Hz interlaced (no 800x600)
4  31.5, 35.15, 35.5; Super VGA, 1024x768 @ 87 Hz interlaced, 800x600 @
56 Hz
5  31.5 — 37.9; Extended Super VGA, 800x600 @ 60 Hz, 640x480 @ 72 Hz
6  31.5 — 48.5; Non-Interlaced SVGA, 1024x768 @ 60 Hz, 800x600 @ 72 Hz
7  31.5 — 57.0; High Frequency SVGA, 1024x768 @ 70 Hz
8  31.5 — 64.3; Monitor that can do 1280x1024 @ 60 Hz
```

```
9  31.5 — 79.0; Monitor that can do 1280x1024 @ 74 Hz
10  31.5 — 82.0; Monitor that can do 1280x1024 @ 76 Hz
11  Enter your own horizontal sync range
```

Действуем по старорусскому принципу: лучше занизить частотные характеристики, чем завысить их. Правда, страшные истории о сгоревших от этого мониторах в прошлом, и худшее, что грозит при переоценке возможностей своей техники — выпадение в черный экран при старте Иксов, но все равно, приятного мало. И в любом случае нужно быть готовым к тому, что даже при строгом следовании спецификации для CRT-монитора идеальной настройки без ручного вмешательства получить не удастся. А для LCD-панели и этот, и следующий параметр роли не играют. Потому что следующий вопрос — о частоте кадровой развертки —

```
You must indicate the vertical sync range of your monitor.
```

предполагает выбор из

```
 1  50-70
 2  50-90
 3  50-100
 4  40-150
 5  Enter your own vertical sync range
```

и также получает ответ из документации.

Далее присваиваем нашему монитору идентификатор, это просто любая последовательность символов, например, наименование его модели. И положительно отвечаем на вопрос, а хотим ли мы ознакомиться с базой данных видеокарт для выбора подходящей:

```
Do you want to look at the card database?
```

База данных эта весьма обширна, более пяти с половиной сотен позиций. Правда, большая часть включенных в нее имен давно ушла в область преданий, а те немногие, что сохранили актуальность, могут выступать под несколькими именами. И для них приемлемы следующие варианты.

Для наиболее, пожалуй, распространенных карт на чипах Nvidia — один из двух:

```
18  ** NVIDIA (generic)             [nv]
349  NVIDIA GeForce          GeForce
```

Для современных карт ATI:

```
5  ** ATI (generic)          [ati]
```

Для встроенного чипсетного видео от Intel:

```
15  ** Intel i810 (generic)  [i810]
291  Intel 810
```

Для эстетов, сохранивших у себя Matrox от G400 и выше:

```
17  ** Matrox Graphics (generic)    [mga]
319  Matrox Millennium G400         mgag400
```

Возможно, вам доводилось слышать о фирменных драйверах от производителей видеокарт ATI, Nvidia, Matrox. Так вот, на данной стадии они недоступны: их следует скачивать с фирменных же сайтов и устанавливать в соответствии с прилагаемой к ним документацией.

Следом идет вопрос об объеме видеопамяти. Здесь можно отвечать буквально что угодно, все равно в итоговом конфиге эта строка окажется закомментированной. Хотя, аккуратности ради, можно поставить и реальное значение.

Теперь задаем идентификатор видеокарты. Также любой, как и идентификатор монитора, он имеет значение только в том случае, если и то, и другое у вас более чем в одном экземпляре (это не шутка. Иксы прекрасно работают в двухмониторных вариантах со всеми картами, такую роскошь поддерживающими, или просто при двух видеокартах). И переходим к настройкам видеорежима. То есть выбираем наборы и последовательности разрешений для каждой из возможных глубин цветности:

```
For each depth, a list of modes (resolutions) is defined.
The default resolution that the server will start-up with
will be the first listed mode that can be supported
by the monitor and card.
Currently it is set to:

"1280x1024" "1024x768" "800x600" "640x480" for 8-bit
"1280x1024" "1024x768" "800x600" "640x480" for 16-bit
"1280x1024" "1024x768" "800x600" "640x480" for 24-bit

Modes that cannot be supported due to monitor or
clock constraints will be automatically skipped
by the server.

 1  Change the modes for 8-bit (256 colors)
 2  Change the modes for 16-bit (32K/64K colors)
 3  Change the modes for 24-bit (24-bit color)
 4  The modes are OK, continue.
```

Набор разрешений не исчерпывается приведенным на экране, он много шире, что можно видеть, если выбрать любой цветовой режим:

```
Enter your choice: 1


Select modes from the following list:


 1  "640x400"
 2  "640x480"
 3  "800x600"
 4  "1024x768"
 5  "1280x1024"
 6  "320x200"
 7  "320x240"
 8  "400x300"
 9  "1152x864"
 a  "1600x1200"
 b  "1800x1400"
 c  "512x384"
 d  "1400x1050"
```

Так что остается только сообразоваться со своими потребностями (в разрешении) и возможностями (монитора, так как видеопамяти в современных картах столько, что они потянут все, что угодно). Следует помнить только о двух вещах: что разрешение, выбранное первым, будет умолчанием для данной глубины цвета, и мало-мальски комфортная работа в интегрированных средах типа KDE или Xfce начинается с разрешения 1024×768. А еще: для LCD-дисплеев, как известно, наилучшее качество изображения получается при разрешении, равном физическому количеству пикселов матрицы, так что ниже лучше не устанавливать, а больше не получится. И последнее: разрешение в Иксах можно переключать на лету, причем в обе стороны, посредством клавишных комбинаций: <Alt>+<Ctrl>+Серый плюс (т. е. символ плюса на малой цифровой клавиатуре, именуемой также KeyPad) в сторону увеличения и <Alt>+<Ctrl>+Серый минус — в сторону уменьшения.

Следующий вопрос весьма интересен:

```
Please answer the following question with either 'y' or 'n'.
Do you want a virtual screen that is larger than
the physical screen?
```

Дело в том, что Иксы поддерживают так называемый виртуальный экран (не путать с виртуальным дисплеем, о котором говорилось ранее): такой, в котором количество точек по горизонтали и вертикали больше, чем физическое разрешение экрана в пикселах. Это выглядит так, будто рабочий стол тянется за пределы дисплея, смещаясь вслед за движением мыши за его границы. Мне это кажется неудобным, и я на этот вопрос почти всегда отвечаю отрицательно, но некоторым такое нравится. И к тому же, может быть полезно при мониторах с маленькой диагональю и на LCD-панелях, при желании получить большее рабочее поле, чем допускает физическое разрешение их матрицы.

Теперь настало время выбрать глубину цвета по умолчанию.

```
Please specify which color depth you want to use by default:
  1  1 bit (monochrome)
  2  4 bits (16 colors)
  3  8 bits (256 colors)
  4  16 bits (65536 colors)
  5  24 bits (16 million colors)
```

Поскольку, как я уже говорил, современные видеокарты допускают любой вариант, думать особенно нечего, 24 бита на пиксел не будет для них обременительным.

Настройка Иксов подошла к концу. Остается только сохранить изменения, ответив положительно на последний вопрос:

```
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?
```

Результатом положительного (`yes`) ответа будет запись всех выбранных параметров в файл /etc/X11/XF86Config (или /etc/X11/xorg.conf) в принятом для него формате.

Вот теперь можно запускать Иксы, для пробы все той же командой:

```
$ X
```

И если пред глазами предстает экран в серую клеточку с крестообразным, реагирующим на движения мыши курсором, процедура завершилась успешно.

Я надеюсь, что из ответов на вопросы, задаваемые программой xf86config, вы поняли, что она делает (если не совсем, окончательная ясность придет при рассмотрении созданного конфигурационного файла). А также обнаружили один из двух его недостатков: при любой ошибке в ответах на вопросы единственная возможность исправить ее — оборвать выполнение программы (комбинацией <Ctrl>+<C>) и начать все сначала.

Второй же недостаток — чрезвычайно громоздкий, перегруженный комментариями (большая часть из которых не нужна) конфигурационный файл, генерируемый этой утилитой (и, добавлю, программой xf86cfg — тоже). Разобраться в котором нелегко. В результате же самоконфигурирования создается конфиг маленький и удобопонятный, вот на нем мы и продолжим свои тренировки.

Самоконфигурирование Иксов начинается просто с запуска X-сервера, но обязательно от лица суперпользователя и с соответствующей опцией (для определенности считаем, что у нас реализация от Xorg версии X11R6.8.2, но с XFree86 все то же самое, с поправкой на имена файлов):

```
$ Xorg -configure
```

Впрочем, этим оно и заканчивается: X-сервер выпадет в осадок с сообщением о невозможности сделать то-то и то-то, например, отыскать мышиное устройство. Пугаться этого не следует: перед безвременной кончиной он успевает создать прототип своего конфига /root/xorg.conf.new (размером менее 3 Кбайт). Остается только скопировать его куда следует:

```
$ cp /root/xorg.conf.new /etc/X11/xorg.conf
```

Теперь можно попробовать запустить Иксы, как это делалось ранее. Если X-сервер встал, все хорошо. Если нет, разбираемся в причинах, читая, какие ошибки он обнаружил. А поскольку ручная доводка сгенерированного конфига все равно потребуется, заодно разберемся и с его устройством.

Файл /etc/X11/xorg.conf разбит на несколько секций, каждая заключена в строки:

```
Section "Имя секции"
...
EndSection
```

Рассмотрим их последовательно, тем более, что в файле /etc/X11/XF86Config они будут точно такими же (различия лишь в некоторых строках, которые или очевидны, или будут оговорены явно). Первой идет секция `ServerLayout`. Содержание ее — указание на генератор конфига (очевидно, зависящий от реализации Иксов):

```
        Identifier     "X.org Configured"
```

и идентификаторы дисплея, мыши и клавиатуры:

```
Screen      0  "Screen0" 0 0
InputDevice    "Mouse0" "CorePointer"
InputDevice    "Keyboard0" "CoreKeyboard"
```

Поскольку все эти устройства у нас в единственном экземпляре, никаких изменений вносить не потребуется. Вариант двухмониторных конфигураций здесь не рассматривается, а два указательных устройства, обычные на ноутбуках (touchpad и, например, внешняя USB-мышь) в данном случае воспринимаются как одно (хотя оба работают нормально).

Следующая секция `Files`. В ней перечислены пути к важным для работы Иксов файлам, в частности, шрифтовым. И это первый объект для правки: в списке каталог с кириллическими шрифтами отсутствует, хотя в умолчальном комплекте Иксов таковые (в каталоге /usr/X11R6/lib/X11/fonts/cyrillic/) имеются. Так что вносим соответствующую строку, причем впереди всех остальных путей к шрифтовым файлам:

```
FontPath     "/usr/X11R6/lib/X11/fonts/cyrillic/"
```

Конечно, шрифты эти не верх совершенства, но на первое время сойдет (а вообще разговор об Иксовых шрифтах впереди).

В следующей секции `Module` нужно проследить главным образом за наличием строки

```
Load  "ttf"
```

Это модуль, обеспечивающий вывод шрифтов TrueType, лучше них пока для Иксов ничего не придумали, по крайней мере, в отношении кириллической составляющей. При желании можно избавиться от лишних модулей, например, для поддержки неиспользуемых шрифтов, таких как speedo (все равно кириллицу они не поддерживают). И хорошо бы добавить строку, весьма способствующую качеству рендеринга шрифтов, но это не всегда проходит:

```
Load  "freetype"
```

Далее идут две секции под одинаковым именем `InputDevice`. Первая по порядку отвечает за клавиатуру, вторая — за мышь, и обе потребуют некоторой правки. Клавиатурная секция в изначальном полуавтоматическом конфиге содержит всего две строки: идентификатор устройства (очевидно, тот же, что и в секции `ServerLayout`) и имя его драйвера:

```
Identifier  "Keyboard0"
Driver      "kbd"
```

Причем (внимание!) вторая строка в XFree86 (и в Xorg версии X11R6.7) выглядит так:

```
Driver      "keyboard"
```

В любом случае, как можно видеть, ни малейшей возможности ввода кириллицы. И чтобы ее получить, вписываем опции Xkb — модуля расширенного управления клавиатурой. Он включен по умолчанию, однако при конфигурировании через xf86config неплохо проследить, не слетел ли случайно символ комментария со строки

```
#        Option "XkbDisable"
```

Опции Xkb определяют правила описания клавиатуры (XkbRules), модель ее (XkbModel), собственно подключаемые раскладки и их варианты (XkbLayout и XkbVariant), а также способ переключения и индикации альтернативной группы (XkbOptions). Все возможные варианты их значений можно посмотреть в файлах каталога /usr/X11R6/lib/X11/xkb/ (каких именно — легко определить, например, командой `grep`). В нашем же случае они таковы:

```
Option "XkbRules"   "xfree86"
Option "XkbModel"   "pc104"
Option "XkbLayout"  "us,ru(winkeys)"
Option "XkbOptions" "grp:caps_toggle,grp_led:caps"
```

С первой строкой понятно (есть еще и другие правила, для иных аппаратных архитектур, но они для нас не интересны). К выбору модели также можно подходить спокойно, `pc104` или `pc105` подходит почти во всех случаях (в том числе и для ноутбуков, где количество клавиш вроде бы меньше). В следующей же секции нужно проследить, чтобы имена раскладок были перечислены именно так и в таком порядке, а вариант `(winkeys)` указан именно в скобках после `ru` — ведь только в этой раскладке он имеет место быть. Обычно подходит также

```
Option "XkbLayout"  "us,ru"
Option "XkbVariant"  ",winkeys"
```

Следует обратить внимание на запятую перед `winkeys`: очевидно, что в раскладке `us` такого варианта нет.

К слову сказать, при настройке Иксов через xf86config следует позаботиться о приведении соответствующей секции в такой же вид: этот конфигуратор полагает, что раз в нем была выбрана русская раскладка клавиатуры, то необходимости в раскладке `us` нет, и соответствующая строка имеет вид:

```
Option "XkbLayout"  "ru"
```

В результате оказывается возможным ввод только кириллицы, но не латиницы.

Секция `InputDevice` для мыши также требует некоторого внимания. Во-первых, в ней таится весьма частая причина ошибки при запуске X-сервера, выражающаяся в невозможности найти все то же мышиное устройство. Почему-то при самоконфигурировании в Linux ему часто приписывается имя

/dev/mouse, тогда как при использовании файловой системы устройств devfs такого имени может и не быть. Так что тут нужно согласиться с наличествующим строками

```
Identifier  "Mouse0"
Driver      "mouse"
Option      "Protocol" "auto"
```

и прописать истинное имя файла, каковым в Linux будет нечто вроде:

```
Option      "Device" "/dev/input/mice"
```

А во FreeBSD — безальтернативно:

```
Option      "Device" "/dev/sysmouse"
```

А для мышей с колесиком еще и добавить такую строку, обеспечивающую в современных версиях Иксов скроллинг без всяких дополнительных ухищрений (типа установки программы `imwheel`, требовавшейся ранее в обязательном порядке):

```
Option      "ZAxisMapping" "4 5"
```

И, наконец, то, что традиционно считается главным при конфигурировании Иксов (и ранее вызывало наибольшие проблемы) — настройка видеорежима. Тут к автоконфигурированию претензий почти нет: X-сервер при первом старте обычно правильно опознает видеокарту (с точностью до чипа), приписывает ему соответствующий драйвер, и, по крайней мере на LCD-дисплеях, устанавливает разрешение, равное физическому разрешению матрицы и 24-битную глубину цвета. А частотные характеристики для LCD-дисплея, как уже говорилось, значения не имеют.

Конечно, в секции `Device` можно кое-что подретушировать, например, включить DRI (Direct Rendering Interface), сняв комментарии со строки соответствующей опции, но это относится уже к поддержке трехмерной графики, нужной почти исключительно для игр.

В итоге путем некоторых обычных манипуляций сгенерированный конфиг доводится до удовлетворительного состояния, на котором можно и успокоиться. Сравним только, ради интереса, его размер с размером конфига, полученного ранее соответствующей утилитой. Это впечатляет, размер последнего около 15 Кбайт, образованных ненужными (и не всегда понятными) комментариями.

Так что в итоге можно констатировать: с точки зрения конечного результата, автоконфигурирование Иксов дает ничуть не худший результат, нежели xorg.config (или xf86config), но избавляет от скучной процедуры, позволяет сэкономить время, а в итоге генерирует компактный и, главное, удобопонятный конфиг. Что же до ручной доводки — она требуется в обоих случаях.

Варианты запуска

Как уже говорилось, Иксы в чистом виде можно загрузить одноименной командой; впрочем, `X` — это лишь символическая ссылка на имя реального файла X-сервера, XFree86 или Xorg (или на X-wrapper, о котором я скажу потом). Однако мы видели, что ничего доброго из этого не выходит; для использования X-сервера в мирных целях необходим его запуск совместно хотя бы с одной клиентской программой.

Благо, для этого в комплекте Иксов любой реализации существует штатное средство — специальный файл /usr/X11R6/bin/startx. Это обычный сценарий оболочки, описывающий последовательность запуска X-сервера и некоего X-клиента. А вот каким будет этот X-клиент, установлено в файле /usr/X11R6/lib/X11/xinit/xinitrc. По умолчанию этим клиентом является twm — единственный наличный оконный менеджер в комплекте Иксов. Вместе с ним запускается также несколько экземпляров программы xterm, что мы и видим на экране после набора команды `startx` в строке любой текстовой консоли. Строки, это описывающие, выглядят следующим образом:

```
# start some nice programs


twm &

xclock -geometry 50x50-1+1 &

xterm -geometry 80x50+494+51 &

xterm -geometry 80x20+494-0 &

exec xterm -geometry 80x66+0+0 -name login
```

Обращаем внимание на символы амперсанда в конце всех строк, кроме последней: это значит, что twm и остальные клиенты работают в фоновом режиме. Последний же экземпляр xterm выступает в качестве своего рода login shell и является программой активной. Выход из него (например, с помощью `exit` в его командной строке) знаменует собой окончание данного X-сеанса.

При запуске через скрипт `startx` фигурирующий в нем файл /usr/X11R6/bin/X по умолчанию является символической ссылкой непосредственно на исполняемый файл X-сервера (XFree86 или Xorg), а последний для запуска от лица обычного пользователя должен иметь бит суидности:

```
lrwxrwxrwx /usr/X11R6/bin/X@ -> XFree86

-rwsr-xr-x /usr/X11R6/bin/XFree86*
```

Такое решение считается неудовлетворительным с точки зрения безопасности, и поэтому часто для запуска Иксов используется специальная программа X-wrapper, вызывающая X-сервер опосредованно. В этом случае послед-

ний в бите суидности не нуждается, он устанавливается только на X-wrapper, и /usr/X11R6/bin/X должен симлинком на него. Именно такой способ используется при автоматическом запуске Иксов, как будет показано далее.

Нужно заметить, что во многих дистрибутивах Linux модифицируется файл /usr/X11R6/lib/X11/xinit/xinitrc по своему усмотрению. В частности, нередко в ответ на команду `startx` запускается просто X-сервер с единственным терминальным окном. Это своего рода безопасный (safe) режим для Иксов: нормально работать при этом затруднительно (ввиду отсутствия средства управления окнами), но вызвать текстовый редактор и поправить файлы конфигурации можно вполне.

Я уже говорил, что оконных менеджеров для системы X преизрядное количество, и большая их часть и функционально, и эстетически далеко превосходят штатный twm. Как обеспечить их запуск по умолчанию вместе со стартом Иксов? Напрашивающееся решение отредактировать /usr/X11R6/lib/X11/xinit/xinitrc не всегда удобно (так как требует прав суперпользователя) и вообще идеологически неправильно. Потому что штатным средством является создание соответствующего пользовательского dot-файла ~/.xinitrc. В простейшем случае в него достаточно внести строку вроде

```
exec fluxbox
```

Вместо `fluxbox` надо вписать имя файла, запускающего любимый оконный менеджер или интегрированный десктоп (который, разумеется, сначала должен быть установлен в системе). При желании запускать вместе с ним еще какие-либо приложения (терминальные окна, утилиты мониторинга системы и т. д.), имена их исполняемых файлов можно поместить в ~/.xinitrc отдельными строками, до или после строки, запускающей оконный менеджер. Важно только, чтобы все строки, кроме последней, заканчивались символом амперсанда (обязывающего к фоновому исполнению соответствующей задачи).

Однако предварительно нужно этот самый менеджер окон выбрать. В следующей главе я дам кое-какую информацию, которая, надеюсь, в этом деле поможет. Однако сделать окончательный выбор без собственного впечатления невозможно. А чтобы получить впечатление, нужно перепробовать их изрядное количество.

На вопросах установки оконных менеджеров здесь останавливаться неуместно, это делается либо штатными средствами управления пакетами данной ОС или дистрибутива, либо обычным порядком, компиляцией из исходников, не забывая только эти самые исходники сохранять, на предмет последующей деинсталляции. Так что поговорим только о средствах запуска оконных менеджеров, уже установленных.

Самый простой способ — это вписать все их имена построчно, закрыв символами комментария, в файл ~/.xinitrc. И запускать по очереди, снимая ремарки с интересующего в данный момент. Нужно только учесть, что некоторые из оконных менеджеров не то, чтобы требуют предварительного конфигурирования, но настоятельно проявляют такое желание, примером является WindowMaker. При этом происходит перезапись имеющегося файла ~/.xinitrc, в расчете на запуск именно сконфигурированной программы управления окнами (хотя старая копия стартового конфига и сохраняется под другим именем).

Другой способ — это все же отредактировать /usr/X11R6/lib/X11/xinit/xinitrc таким образом, чтобы в качестве X-клиента запускалось лишь одно терминальное окно:

```
xterm
```

А уж из его командной строки запускать подлежащий изучению менеджер окон. При этом можно, оказывается, запустить несколько X-сессий, каждую со своим менеджером окон, стартующим из командной строки эмулятора терминала.

Для запуска нескольких X-сессий одного и того же X-сервера следует вспомнить о понятии виртуального дисплея. X-сервер, запущенный первым, стартует на нулевом дисплее, который соответствует ближайшей доступной и свободной (то есть не активизированной процессом типа `getty`) виртуальной консоли. И попытка дать команду `startx` (или просто `X`) повторно (с другого виртуального терминала) выведет сообщение о том, что дисплей занят. Чтобы этого не случилось, номер дисплея (начиная с 1) для второй сессии Иксов (всех последующих) следует задать явно:

```
startx -- :1
```

Здесь следует не забыть про удвоение символов дефиса (`--`), знаменующее окончание списка аргументов (возможных X-клиентов и относящихся к ним опций). Теоретически указание на номер виртуального дисплея следует предварить номером дисплея физического, в форме

```
startx -- 0:1
```

Но практически для персоналки в обычной конфигурации его можно опустить. Опять-таки, теоретически, как говорилось ранее, можно открыть до 231 самостоятельных X-сессий, но практически, думаю, лимит оперативной памяти будет исчерпан много раньше: каждая сессия требует для себя 16 Мбайт суммарной (RAM + swap) памяти, из которых 4 Мбайт приходится на память физическую; и немало памяти отъест еще и оконный менеджер и, особенно, интегрированная среда типа KDE или Gnome.

В реальных условиях запуск нескольких X-сессий на пользовательском десктопе имеет смысл только в целях экспериментирования с оконными ме-

неджерами или каких-то специальных случаях (хотя каких именно — ума не приложу). Дело в том, что, в отличие от виртуальных консолей, обмен данными между X-сессиями невозможен (по крайней мере, я такого способа не нашел ни в ходе практических попыток, ни в документации). Так что запуск дополнительного сеанса Иксов ничего не прибавит нам в плане расширения рабочего пространства. Да его и так вдоволь, все современные оконные менеджеры, в дополнение к экранам с виртуальными разрешениями (это функция самого X-сервера) поддерживают большое количество так называемых виртуальных рабочих столов, на каждом из которых может быть открыто практически неограниченное количество окон.

Если основная работа пользователя проходит в графическом режиме, возникает естественное желание: обеспечить его старт при загрузке машины. В Linux это обычно делается очень просто: нужно только открыть файл /etc/inittab, посмотреть в начале его перечень runlevels:

```
# Runlevels:
# 0    Halt
# 1(S) Single-user
# 2    Not used
# 3    Multi-user
# 4    Not used
# 5    X11
# 6    Reboot
```

и определить, какой из них соответствует в данном дистрибутиве старту в графическом режиме (в примере, относящемся к Archlinux, — 5-й), назначить его умолчанием, изменив строку

```
id:3:initdefault:
```

на

```
id:5:initdefault:
```

Никаких дополнительных телодвижений, скорее всего, не потребуется: после перезагрузки машины перед глазами предстанет, вместо первой виртуальной консоли с предложением к авторизации, графический менеджер входа в систему, по умолчанию им будет xdm из штатного Иксового комплекта. Достаточно набрать учетное имя своего акаунта и пароль доступа, и пользователь оказывается внутри X-сессии. Под соответствующий ей X-дисплей будет задействована первая же свободная виртуальная консоль (в большинстве дистрибутивов седьмая). В отличие от X-сессии, запущенной вручную, управляющего терминала у нее не будет, все прочие виртуальные консоли остаются девственно чистыми.

Во FreeBSD и DragonFlyBSD дело обстоит ничуть не сложнее: открывается файл /etc/ttys, в нем отыскивается имеющаяся по умолчанию строка

```
ttyv8  "/usr/X11R6/bin/xdm -nodaemon"  xterm   off secure
```

и в ней значение `off` в четвертом поле заменяется на `on`, послед чего выполняется рестарт машины, с тем же результатом, что и в Linux.

В любом случае важно только проследить, чтобы `X` при этом был ссылкой на wrapper, а файл имел бы бит суидности:

```
lrwxr-xr-x /usr/X11R6/bin/X@ -> /usr/X11R6/bin/Xwrapper-4
-r-sr-xr-x /usr/X11R6/bin/Xwrapper-4*
```

Прервать запущенную X-сессию, то есть перезагрузить X-сервер (например, при изменении его настроек) можно клавишами <Alt>+<Ctrl>+<Backspace> или штатными средствами запущенного клиента, но это вызовет лишь возобновление приглашения xdm. Полностью избавиться от графического режима можно только возвратом общесистемных конфигов (/etc/inittab или /etc/ttys) в исходное состояние.

Авторизовавшись в панели xdm, пользователь оказывается в X-сессии, но не совсем в своем привычном и настроенном (через ~/.xinitrc) окружении. Ибо при автоматическом старте Иксов отработки сценария `xinit` (и, соответственно, считывания конфигов xinitrc и ~/.xinitrc, не производится, оно будет определяться умолчаниями общесистемного xinitrc. Чтобы обеспечить запуск своего любимого оконного менеджера и сопутствующих ему программ (то, что при команде `startx` загружается из "домашнего" ~/.xinitrc), необходимо иметь файл ~/.xsession. Для начала его можно сделать идентичным ~/.xinitrc, ограничившись строкой с именем оконного менеджера, а дальше — сами разберетесь.

Для графической авторизации можно использовать и другие менеджеры входа в систему, например, kdm из комплекта KDE, или gdm, входящий в состав GNOME. По сравнению с xdm, они предоставляют множество дополнительных возможностей. Правда, и требуют немало — соответствующих интегрированных сред в установленном виде.

Немного о раскладках

Упомянув об эмуляторах терминала и менеджерах окон, я несколько забежал вперед, однако без какой-либо из клиентских программ этого семейства невозможно даже проверить правильность настройки Иксов, например, клавиатуры и видеосистемы, а также шрифтов. Конечно, заведомо неподходящий драйвер клавиатуры или видеокарты, а также неправильный путь к шрифтам, просто не позволят стартовать X-серверу. Однако огрехи, скажем,

русификации раскладки или отсутствие кириллических шрифтов работать ему не помешают, вот только не совсем так, как нам нужно.

Для начала проверяем ввод кириллических символов, прямо в командной строке терминального окна, без запуска какого-либо иного приложения. И с удивлением обнаруживаем (не обязательно, но вполне возможно), что ничего при этом не происходит, хотя при настройке через xf86config была выбрана русская раскладка. Почему? — спрашиваем сами себя. И сами же себе отвечаем: потому что не позаботились о правильном определении системной локали. Сама по себе локаль как-будто бы к Иксам отношения не имеет, но вот ее несоответствие Иксовой раскладке блокирует клавиатурный ввод напрочь. Так что, если это не было сделано ранее, перво-наперво устанавливаем правильную локаль — в соответствии со страной (`ru`), языком (`RU`) и используемым набором символов (например, `KOI8-R`).

Теперь, вполне возможно, на экране при вводе мы видим сплошную абракадабру. Причина понятна — отсутствие русских шрифтов (или путей к ним в Иксовом конфиге). Однако попытка переключиться на латиницу дает тот же эффект. Вспоминаем, что это мы уже проходили несколькими разделами ранее, при настройке через xf86config выбор русской раскладки клавиатуры подразумевает почему-то, что в латинских буквах у пользователя потребности нет. И там же, в разделе о конфигурации Иксов, ищем рецепт — ручную правку соответствующей секции конфигурационного файла.

Теперь переключение осуществляется нормально, но абракадабра с экрана никуда не исчезает. Что свидетельствует о проблеме со шрифтами. Но это будет темой следующего раздела, а пока закончим с раскладками.

Определив все опции клавиатуры, как было описано ранее, получаем нормальную раскладку для win-маркированной клавиатуры (вариант winkeys), с заказанным переключателем и индикатором альтернативной группы. Однако стоит ли останавливаться на достигнутом? Меня, например, всегда просто бесили все стандартные русские раскладки: и та, что была на пишущих машинках (унаследованная dos-маркированными компьютерными клавишами), и win-модификация ее. Поневоле поверишь в то, что вообще раскладка qwerty была придумана для того, чтобы замедлить работу. А ее кириллические вариации дело еще усугубили (не иначе, на предмет дополнительного снижения амортизации казенной техники). По настоящему удобной представляется только русская раскладка FreeBSD (ru.koi8-r.kbd), в которой знаки препинания находятся в нижнем регистре верхнего ряда, а цифры — в верхнем. Да только уж больно непривычна, и клавиатур, таким образом маркированных, не сыскать.

Тем не менее, я по мере сил и в минимально необходимом объеме всегда пытаюсь выправить положение. Для этого в используемой мной кириллической раскладке перемещаю запятую на нижний регистр, ввожу в нее символ

прямого слэша (по умолчанию почему-то обычно имеется только обратный) и доллара (последнее не из любви к длинному и зеленому, а для обозначения приглашения командной строки). Как это сделать в консоли — можно сообразить. А вот как быть в Иксах?

На теории этого дела останавливаться не буду, она некогда была подробно описана Иваном Паскалем в его труде об Xkb (**http://pascal.tsu.ru/other/xkb/**), до сих пор не утратившем своего значения. Так что только пара практических рецептов.

Расширенные возможности управления клавиатурой в Иксах обеспечиваются модулем Xkb. Все относящиеся к нему файлы собраны в каталоге /usr/X11R6/lib/X11/xkb. В данный момент нас интересует лишь один из его подкаталогов — symbols, файлы которого описывают, как можно понять из его имени, непосредственно наборы символов, привязанные к клавиатурным раскладкам различных стран. Точнее, один из вложенных в него подкаталогов symbols/pc для одноименной архитектуры. Ну, а в нем уже важен файл ru, это непосредственно символы, привязанные к нашей, родной, кириллической раскладке.

Формат файла достаточно прозрачен, и становится понятным при его просмотре: каждому коду клавиши ставятся в соответствие два символа — в нижнем регистре и верхнем. Файл разделяется на несколько секций. Первая (`Basic`) описывает базовую русскую раскладку, остальные представляют собой варианты и содержат лишь изменения относительно базового набора символов.

В данный момент перед нами стоит задача внести косметические изменения в раскладку `ru(winkeys)`. Начинаю я, однако, с секции `Basic`:

```
partial default alphanumeric_keys

xkb_symbols "basic" {
```

Ибо первый напрашивающийся кандидат на модификацию — это клавиша <Left Backslash> (<Bar> на верхнем регистре), по прямому назначению мной никогда не используемая. Отыскиваем ее код (это будет `LSGT`) и значение нижнего регистра `backslash` заменяем на `comma` (сиречь запятую). За верхним же регистром резонно сохранить исходное значение `bar`.

Позиция, ранее задействованная под запятую в русской раскладке, у меня таким образом освободилась. Не пропадать же добру! Займу-ка я ее прямым слэшем. Для чего отыскиваем (то есть я отыскиваю) соответствующую секцию, теперь уже

```
partial alphanumeric_keys

xkb_symbols "winkeys" {
```

нахожу в ней нужный код (AB10) и умолчание comma верхнего регистра меняю на slash. Остается разобраться с любимым знаком доллара. Его я помещаю на место нормального (правого) backslash, тогда как последний передвигаю на регистр верхний (код их клавиши BKSL).

И в итоге все модифицированные строки выглядят таким образом (в порядке их исходного расположения в файле /usr/X11R6/lib/X11/xkb/symbols/pc/ru:

```
partial default alphanumeric_keys


xkb_symbols "basic" {


        name[Group1]= "Russian";


        ...
        key <LSGT> { [ comma, bar ] };
        ...
};


partial alphanumeric_keys


xkb_symbols "winkeys" {


        include "pc/ru(basic)"
        ...
        key <AB10> { [ period, slash ] };
        key <BKSL> { [ dollar, backslash ] };
};
```

Не знаю, как кому, а мне так очень удобно. Единственный минус — после этого работать на чужих машинах становится просто мучительно больно. Таким образом появляется стимул — делать это как можно реже...

Шрифты: полиграфическое введение

Шрифты — неотъемлемый атрибут среды обитания пользователя. Действительно, мало радости будет от иконок и панелек круто интегрированного графического десктопа, если подписи на них не будут читаемыми. Да еще

и нерусскими буквами. И потому в этом цикле мы попытаемся собрать все, что имеет отношение к шрифтам, используемым как в консоли, так и Иксах.

В этом разделе содержатся лишь краткие сведения о шрифтах вообще и относящейся к этой теме терминологии в частности. Более подробную информацию можно получить из следующего монументального труда: Б. М. Кисин. Графическое оформление книги. Л.: Гизлегпром, 1946, 408 с. Глава о шрифтах доступна и по адресу: **http://citforum.ru/open_source/fonts/theory**.

Разговор о шрифтах следует начать с терминологии, для чего вспомним традиции отечественной полиграфической школы. В соответствии с ней набор символов определенного облика (то, что в компьютерной литературе называют обычно шрифтом, а то и фонтом) носит имя "гарнитура". В англоязычной литературе ему соответствует выражение Font Family. Примерами классических гарнитур являются Times, Courier, Helvetica и др.

Визуальные вариации символов каждой гарнитуры описываются термином *шрифтоначертание*. В большинстве гарнитур вариации эти выделяются на основе насыщенности и геометрии. По первому признаку выделяются нормальное (normal или medium) шрифтоначертание и, скажем, полужирное (semibold) или жирное (bold); хотя в просторечии "жирным" обычно называют semibold. Геометрически различают прямое (regular) и курсивное (oblique или italic) шрифтоначертания. Кроме того, некоторые гарнитуры могут иметь еще и дополнительные вариации, такие как подчеркнутое или перечеркнутое шрифтоначертания.

Так вот, индивидуальное имя шрифта складывается из названия его базовой гарнитуры и шрифтоначертания, например: Times полужирный курсивный или "Петербург" нормальный подчеркнутый. Строго говоря, именно к сочетанию имени гарнитуры и шрифтоначертаний и относится термин шрифт (font).

Шрифтовые гарнитуры объединяются в семейства по двум независимым признакам. С одной стороны, различают семейства *моноширинных* и *пропорциональных* гарнитур. В первом все символы занимают одно и то же место по горизонтали, вне зависимости от их относительной ширины. Это шрифты пишущих машинок и их имитирующие: Courier, "Балтика". В семействе пропорциональных гарнитур под символы, относительно более узкие (например, русское "т"), отводится и меньшее (в абсолютных единицах) место по горизонтали, чем под символы широкие (скажем, "щ").

Второй критерий разделения гарнитур на семейства — наличие или отсутствие так называемых серифов (serif). Это слово обычно переводится как "засечки", хотя в советском учебнике полиграфии дается более точное определение — отсечки. Действительно, это как бы декоративные дополнитель-

ные выступы, которые "отсекают" строку по горизонтали, способствуя фокусировке глаза вдоль нее. Так вот, гарнитуры, символы которых таковые имеют, объединяются в семейство Serif (или "Антиква" — шрифты с отсечками), прочие же — в семейство Sans Serif (иногда называемое также "Гротеск").

В принципе, в старых книгах по полиграфии можно найти более дробные схемы классификации шрифтов. Так, шрифты с отсечками разделяются группы "Старая Антиква" и "Новая Антиква", обособляется группа египетских шрифтов, не говоря уже о шрифтах декоративных и имитирующих рукопись. Однако в контексте того, что в дальнейшем речь у нас будет о шрифтах компьютерных, а не полиграфических, сказанного ранее представляется достаточно.

Очевидно, что как антиквы, так и гротески могут быть и моноширинными, и пропорциональными.

Примерами моноширинных гарнитур с отсечками являются упомянутые уже Courier и "Балтика" (стандартные шрифты латиницы для импортных и кириллицы для советских пишущих машинок соответственно). Пример классических моноширинных гротесков мне на память не приходит, но в компьютерном мире они употребляются очень широко: это вариации на темы Fixed, Lucida Typewriter, Andale и им подобные.

Пропорциональные гарнитуры семейства антиква — это подавляющее большинство гарнитур, представленных в книгах и прочей полиграфической продукции. Исторически большинство из них восходит к классическим гарнитурам, создававшимся на протяжении XVI—XVIII вв. знаменитыми художниками и граверами. Среди них антиква Дидо, шрифты Гарамона, Кэслона, Баскервилля, Бодони (знакомые имена, не так ли?), к которым в XIX в. прибавляется Times. Ныне же вариациям на тему "Антиква" несть числа: гарнитуры "Книжная", "Петербург", "Академическая", гарнитура Лазурского... Перечислять можно было бы долго. А если вспомнить про их современные компьютерные вариации, так и вообще можно сбиться со счета.

Семейство пропорциональных гротесков зародилось в начале XIX в. в связи с формированием отдельных стилей оформления печатной продукции, для книг, газет и прочей периодики, да и рекламные проспекты массово начинают распространяться именно в это время. Ими и стали многочисленные гротески: классическая Helvetica и ее многочисленные вариации типа Swiss, вплоть до всем известного компьютерного Arial, шрифты Futura, Gothic (не путать с настоящими старинными готическими шрифтами и их современными имитациями, которые относятся к категории декоративных шрифтов).

Традиционно в полиграфии использовались преимущественно пропорциональные гарнитуры; моноширинные гарнитуры заняли в ней прочное место лишь в последнее время для передачи листингов программ, команд и тому

подобного хозяйства в околокомпьютерной литературе. Причем гарнитуры с отсечками и гротески занимали каждая свою нишу. Первое семейство предназначалось для передачи длинных и "сплошных" повествовательных текстов, второе же — для коротких текстов блочного характера и иногда заголовков. Широкое распространение, с одной стороны, компьютерных технологий в издательском деле, с другой — онлайновых изданий, предназначенных для восприятия с экрана, существенно сместило сферы применения гарнитур того и другого семейства: гротески стали широко применяться для сплошных текстовых массивов сначала в онлайновых, а затем и бумажных изданиях, а гарнитуры семейства Serif — напротив, для декоративного выделения блочных участков.

Интересно, что эксперименты для оценки влияния шрифтов на качество восприятия текста проводились еще задолго до появления компьютеров, в 30-х годах прошлого века. И результаты их оказались не вполне тривиальными. Конечно, общепринятое мнение о том, что восприятию сплошных текстовых массивов более способствуют гарнитуры с отсечками, подтвердилось. Но только для людей взрослых и образованных. Дети и люди малограмотные даже сплошные массивы лучше воспринимали в исполнении гротескными гарнитурами.

И еще выяснилось, что для текста, подвергнутого искажениям, читабельность лучше (или, скажем так, менее плохо) сохраняется именно в случае использования шрифтов без отсечек. Последнее имеет прямое отношение к использованию компьютерных шрифтов. Ведь на экране текст всегда в той или иной мере подвергается искажениям, за счет кривизны CRT-трубок, пикселизации, да и просто из-за недостаточно высокого качества рендеринга. И потому в качестве экранных шрифтов гротески почти всегда оказываются предпочтительными.

Разборки со шрифтами

Пора возвращаться к генеральной линии нынешнего раздела — компьютерным шрифтам вообще и их использованию в Иксах в частности. Существуют две формы представления шрифтов на экране: растровая и векторная. В первой форме изображение символа формируется из описания составляющих его точек в виде матрицы пикселов, во второй же — описывается уравнениями специальных кривых, именуемых кривыми Безье. Разумеется, и векторные шрифты выводятся на экран попиксельно, мы ведь помним, что компьютерные дисплеи суть устройства растровые по своей природе. Однако векторная форма представления символов позволяет получать более "гладкие" изображения на мониторах с высоким разрешением и добиваться

более качественного вывода на печать (ведь разрешение самого плохого принтера выше, чем самого лучшего монитора).

Очевидно, что растровые шрифты единственно пригодны для применения в чисто текстовом режиме. Теоретически в графической (через Frame Buffer) консоли можно было бы воспроизводить и векторные шрифты, но практически это нигде и никем не реализовано. А вот в Иксах могут быть использованы как растровые, так и векторные шрифты, причем последние — во многих форматах.

Для растровых шрифтов в Иксах обычно применяется формат PCF (Portable Compiled Font). Эти шрифты по своей природе почти не масштабируемы, как и консольные шрифты для текстового режима. Конечно, в графическом режиме символ, образованный матрицей точек, можно в принципе растянуть (или сжать) по вертикали и горизонтали как угодно, но вид его при этом будет варьировать от плохого до вполне отвратительного. И потому каждая PCF-гарнитура содержит файлы шрифтов с несколькими матрицами точек; собственно, число матриц конкретного шрифта и определяет возможности его масштабирования. Но не только: наборы растровых шрифтов предназначены для совершенно конкретных разрешений, как правило: 75 и 100 dpi (dot per inch, то есть точек на дюйм, здесь понятие разрешения используется в типографском смысле, а не в контексте видеосистемы). Первые используются при экранных (то есть пиксельных) разрешениях вплоть до 800×600, вторые — от 1024×768 и выше.

Важно, что в гарнитурных наборах для растровых шрифтов мы, как правило, не найдем отдельных файлов для шрифтоначертаний. Правда, как мы позже увидим, в большинстве случаев таким шрифтам можно придать полужирное или курсивное начертание, но делается это средствами X-сервера, а не на основе описания собственно шрифтового файла. Соответственно, и результат получается не блестящим...

В Иксах находят себе применение как моноширинные, так и пропорциональные гарнитуры растровой формы воспроизведения. Первые используются главным образом в терминальных окнах, вторые же — для элементов интерфейса, а также вывода текстов в большинстве прикладных программ.

Штатная поставка Иксов в обязательном порядке включает в себя два шрифтовых набора растровых шрифтов общего назначения, с разрешениями 75 dpi и 100 dpi: именно под такими именами фигурируют содержащие их подкаталоги в сводном шрифтовом каталоге /usr/X11R6/lib/X11/fonts/. Ни тот, ни другой набор (а в их составе гарнитуры Courier, Helvetica, Times и несколько специальных) символов кириллицы не содержат ни в какой кодировке ее.

Впрочем, дискриминация русскоязычных пользователей не столь уж выражена: испокон веку в комплекте Иксов есть и набор растровых шрифтов

производства фирмы Cronyx, включающий гарнитуры Courier, Fixed (моноширинные), Times и Helvetica (моноширинные), правда, только для кодировки KOI8-R. Качество их не то чтобы очень высокое, но на первое время сойдет.

Столь же обязательно наличие в комплекте Иксов так называемых misc-шрифтов, которые располагаются в каталоге /usr/X11R6/lib/X11/fonts/misc. Это чисто моноширинные шрифты единственной гарнитуры (Fixed), которая включает набор шрифтовых файлов с разными матрицами (4×6, 5×7, 6×8, и так вплоть до 10×20) и для разных наборов символов: в числе их, наряду со всеми кодировками ISO8859 (включая ISO8859-5, ранее это называлось кодировкой ГОСТ для кириллицы), присутствует и KOI8-R. Шрифты misc-fixed предназначены в основном для терминалов, но могут использоваться и в приложениях (например, в текстовых редакторах).

В отечественные дистрибутивы Linux, а также в порты FreeBSD, включены дополнительные растровые шрифты с поддержкой кириллицы. Насколько мне известно, все они представляют собой расширения и улучшения набора Cronyx, выполненные Дмитрием Болховитяновым, собравшим их в виде пакета cyr-rxf. Пакет этот включает, кроме стандартных гарнитур Times, Helvetica, Courier, дополнительные пропорциональные гарнитуры Lucida и Serene и моноширинные — Lucida Typewriter и Serene Typewriter, правда, только для кодировки KOI8-R. Несмотря на то, что все эти шрифты рассчитаны на разрешение 75 dpi, каждая гарнитура содержит большое количество шрифтовых файлов с разными матрицами (от 8 до 24 точек по вертикали), что позволяет использовать их при высоких разрешениях. И еще одна отличительная особенность пакета — наличие отдельных шрифтовых файлов для разных начертаний: нормального, курсивного и полужирного. С подробностями об этих шрифтах можно ознакомиться на сайте автора: **http://www.inp.nsk.su/~bolkhov/files/fonts/cyr-rfx/00index.ru.html**.

Что касается векторных шрифтов, Иксы, насколько мне известно, поддерживают все распространенные их форматы. Однако практическое значение в наших условиях имеют только Adobe Type 1 и TrueType Fonts (TTF). Прочие, типа Speedo, поддержкой кириллицы не обременены. Однако и для Type 1 или TTF кириллических вариантов штатно не предусмотрено. И потому обзаведение ими для использования в Иксах — дело рук самого пользователя.

Первоначально Иксы сами по себе (без специального фонт-сервера) поддерживали только шрифты ATM.

Положение начало меняться только с появлением в Иксах поддержки шрифтов TrueType. В этом формате Валентином Филипповым был создан первый самодостаточный и лицензионно безупречный шрифтовой набор,

поддерживающий все кодировки кириллицы. Он же был продублирован и в формате ATM, что делает его в некоторых случаях незаменимым.

Шрифты Валентина Филиппова распространяются в составе дистрибутивов Altlinux (и некоторых других), но доступны в и сами по себе (**ftp://ftp.gnome.ru/fonts/urw/**). Гарнитуры обоих наборов весьма многочисленны: здесь и гротески Avantgarde, Gothic, Helvetica, Nimbus Sans, и серифы Bookman, Centure Scoolbook, Nimbus Roman, Palatino, Times, и моноширинные гарнитуры Courier и Nimbus Mono. Однако полностью потребности всех пользователей (например, меня) они удовлетворить не могут.

В результате наибольший успех в мире свободных систем выпал, парадоксальным образом, на долю шрифтов от классового врага — Microsoft. Справедливости ради нужно заметить, что шрифты эти суть достижение творческого гения фонт-дизайнеров фирмы Monotype, специализирующейся на такого рода деятельности, а Microsoft выступала только как заказчик и, в дальнейшем, собственник их.

Малый джентльменский набор ttf-шрифтов от Microsoft, включающий гарнитуры гротеска Arial, Tahoma, Verdana, сериф Times New Roman, моноширинные Andale и Coutier New, долгое время (видимо, по недосмотру) был доступен свободно на сайте Microsoft. Правда, на условиях распространения только через Интернет (то есть без права на включение в дистрибутивы) и исключительно в первозданном виде, каковой являл собой, понятное дело, cab-архивы. Теперь с сайта Microsoft эти шрифты исчезли, но явного запрещения на их использование и сетевое распространение, насколько мне известно, так и не последовало. И потому их можно (в виде пакета corefonts) скачать с одного из сайтов и использовать в своей системе вполне легально. На сей предмет была даже специально придумана утилита cabextract и разработана методика ее применения.

Описывать эту методику не буду: я не настолько наивен, чтобы полагать, будто в наших условиях ею кто-то воспользуется (при наличии более простого способа получения этих шрифтов, на котором не будем акцентировать внимание).

Появился и другой источник качественных ttf-шрифтов — дистрибутив Suse Linux, начиная с версии 9.3 и выше. Он содержит шрифты разного происхождения. Во-первых, здесь можно видеть работы известного фонт-дизайнера Рэя Лараби. Правда, для мирных целей они не очень интересны, так как носят преимущественно декоративный характер (да и, между нами говоря, кириллицу не поддерживают).

Далее, в дистрибутиве Suse имеется большое количество каких-то изначально свободных шрифтов неизвестного происхождения, часть из них поддерживает кириллицу.

И, наконец, самое главное: шрифты производства фирмы Monotype, той самой, что разработала corefonts для Microsoft. И эти шрифты сработаны столь же аккуратно и качественно. Правда, представлены они лишь четырьмя гарнитурами: двумя серифами (Cumberland и Thorndale) и двумя шрифтами без отсечек (Albany и Andale Sans). Но зато поддерживают кириллицу во всех ее проявлениях, от Unicode до KOI8-R. А уж о качестве шрифтов и говорить не приходится...

Все шрифты от Novell в дистрибутиве Suse сопровождаются лицензией GPL, из чего можно сделать вывод о их свободном распространении. Хотя окончательной ясности в этом вопросе нет, но бесспорно, что никаких препятствий для использования этих шрифтов в личных целях нет.

Вернемся, однако, к столь полюбившимся пользователям шрифтам от MS (тем более, что все описанное ниже справедливо и для любых иных шрифтовых наборов). Как уже говорилось, файлы любого шрифтового набора принято помещать в отдельный каталог внутри каталога /usr/X11R6/lib/X11/fonts/. Поступим так и с извлеченными из cab-архива TTF-шрифтами, создав для них подкаталог corefonts (например). Однако чтобы Иксы были в состоянии их воспринять, шрифтовой подкаталог должен содержать еще файл fonts.dir и (для масштабируемых векторных шрифтов) fonts.scale. Это простые списки имен шрифтовых файлов и полного описания соответствующего им шрифтоначертания, например:

```
Arial.TTF -monotype-Arial-medium-r-normal--0-0-0-0-p-0-koi8-ru
```

Эту длинную последовательность можно представить в виде подобия базы данных, в которой имя шрифтового файла (в примере Arial.TTF) выступает в качестве идентификатора записи, а все остальное образовано отдельными полями с дефисом в качестве разделителя. Поскольку тот же формат описания шрифта используется утилитой xfontsel и в так называемых файлах X-ресурсов (речь вскоре дойдет и до того, и до другого), остановимся на смысле отдельных полей подробнее. Первое поле в терминологии xfontsel носит название `fndry` (от слова foundry), одно из значений которого — форма (шрифта). А его содержание — это имя фирмы-разработчика, известного разработчика шрифтов Monotype.

Маленькое отступление для тех, кто не может поступиться принципами

Так вот, все TTF-шрифты из поставки Windows разработаны именно фирмой Monotype. Более того, гарнитуры Arial, Courier New и Times New Roman — это вариации на тему классических гарнитур Helvetica, Courier и Times соответственно, которые вообще являются общенародным достоянием, а Tahoma и Verdana созданы Monotype по мотивам гротесков середины прошлого столетия.

О происхождении Andale, к сожалению, ничего сказать не могу, но полагаю, и тут ничто не ново под луной. Так что при использовании этих шрифтов поступаться принципами не приходится...

Второе поле (в xfontsel оно называется `fmly`, то есть family) — это имя гарнитуры, которую данный шрифтовой файл представляет. Третье (`wght`, от weight, то есть толщина) — шрифтоначертание в критериях насыщенности. В следующем же поле (`slant`) описывается шрифтоначертание с точки зрения наклона: нормальное (`r` от roman) или курсивное (`i` от italic и `o` от oblique соответственно). За этим следует поле `sWdth` (set Width), описывающее "компоновку" шрифта. Ну, а позиция между двумя дефисами — это принадлежность гарнитуры к одному из трех семейств (к традиционным serif и sans serif добавляется decorative); очевидно, что она следует из имени гарнитуры, и потому здесь это поле пустое.

Следующие четыре поля определяют: размер шрифта в пикселах (`pxlsz`), размер шрифта в "точках" (points, `ptsz`), разрешение шрифта по горизонтали (`resx`) и вертикали (`resy`) в точках на дюйм (dpi). В примере все они заполнены нулями, что свидетельствует о принадлежности рассматриваемого шрифта к категории масштабируемых (таковыми являются все векторные шрифты, как TTF, так и Type 1). Однако для растровых шрифтов здесь стояли бы соответствующие значения, характеризующие данный шрифтовой файл, например:

```
10-100-75-75.
```

Следующим полем (`spc` — от spacing) описывается принадлежность к пропорциональному (`p`) или моноширинному (`m`) семействам. А затем поле `avgWdth`, в котором определяется средняя ширина символа в шрифте. Наконец, в последних двух полях указываются набор символов (`rgstry` — от registry, и `encdng` — от encoding) шрифта (например, `koi8`) и его вариации (например, `ru` и `r` для KOI8-R).

Рассмотренное описание каждого шрифта может показаться несколько запутанным и непонятным. Тем не менее, это объективная реальность, данная нам в ощущениях разработчиков Иксов, и с ней приходится считаться. То есть — создать файл fonts.dir в новообразованном каталоге. Благо, для этого существуют утилиты: mkfontdir для шрифтов Type 1 и ttmkfdir — для шрифтов TrueType. Правда, в комплект Иксов входит лишь первая, вторая должна быть установлена самостоятельно. Однако использование обеих элементарно просто: достаточно дать одноименную команду с именем шрифтового каталога в качестве аргумента, в нашем примере это будет выглядеть так:

```
$ ttmkfdir /usr/X11R6/lib/X11/fonts/corefonts
```

И получить в результате тот самый файл fonts.dir со всеми его зубодробительными записями, а заодно — и с количеством оных в первой строке (это — обязательный элемент файла, без него X-сервер не сможет построить таблицу доступных шрифтов). Важно также, что для юникодных шрифтов автоматически будут заполнены поля `rgstry` и `encdng` в соответствии с реально доступными в них наборами символов.

Что же касается файла fonts.scale для масштабируемых шрифтов, то его содержание идентично таковому fonts.dir, и его можно создать простым копированием последнего (а также сделать жесткой или символической ссылкой на него).

Теперь остается совсем немногое: внести в секцию `Files` Иксового конфига строку вида

```
FontPath     "/usr/X11R6/lib/X11/fonts/corefonts/"
```

и перезапустить X-сервер, чтобы наслаждаться кириллическими TTF-шрифтами. Проследив предварительно, имеет ли место быть в секции `Module` конфига строка

```
Load  "ttf"
```

их поддержки: с некоторых пор это штатная функция X-сервера, тогда как ранее для использования TrueType требовались специальные серверы шрифтов (Fonts Server). Использование их не возбраняется и ныне, однако на настольной машине, по моему мнению, лишено смысла. Единственное, что дает Font Server, — это возможность загрузки шрифтов с удаленного компьютера, что для пользовательского десктопа не актуально. А вот тормозить работу Иксов сервер шрифтов может вполне...

Теперь нужно только найти способ разбираться с наличным шрифтовым богатством: выбирать нужные шрифты и устанавливать их. Что касается последнего, для собственно Иксовых приложений это делается либо опциями их командной строки при запуске, либо в специальных файлах X-ресурсов, о чем речь пойдет в ближайшей интермедии. А вот выбор нужного для установки шрифта возможен на поминаемую выше утилиту xfontsel. Она позволяет через пункты меню, соответствующие описанным ранее полям в файле fonts.dir, выбрать для последних нужные значения (с визуальным контролем результата), а затем, нажатием на кнопку **select**, скопировать их в буфер. После чего параметры нужного шрифта могут быть либо вставлены в командную строку в виде значения соответствующей опции (например, опции `-fn`), либо вставлены в описание ресурсов приложения в файле типа ~/Xdefaults.

Выбирать шрифт через меню xfontsel можно в любом порядке. Например, можно сначала через два последних пункта (**rgstry** и **encdng**) отфильтровать

только шрифты с поддержкой кириллицы, а затем уже определиться с гарнитурой, шрифтоначертанием и размером. Более того, для корректного представления шрифта часто нет нужды в определении всех параметров шрифта: некоторые из них (как, например, размер в пикселах и точках, горизонтальное и вертикальное разрешения) жестко сцеплены друг с другом: выбор размера шрифта 12 пунктов автоматически влечет установку значения 120 в точках. А такие параметры, как семейство гарнитур или их характер (пропорциональный и моноширинный), вообще однозначно вытекают из имени гарнитуры. Хотя шрифты можно фильтровать и по этим признакам: сначала отобрать все моноширинные гарнитуры, пригодные для использования в терминале, а потом уже детализировать свой выбор. Почему я и говорил, что формат описания шрифта содержит в себе некую сермяжную правду...

Интермедия: о менеджерах окон

- О менеджерах окон.
- Об организации десктопа.
- FLWM, или предел минимализма.
- Blackbox как представитель семейства.
- WindowMaker: как делать окна.
- IceWM: гибкость и настраиваемость.

Как уже было сказано, сами по себе Иксы ни для каких полезных целей использоваться не могут. Разве что запустить их одновременно с эмулятором терминала, в командной строке его и работать. Можно запускать любые, в том числе и графические, приложения в фоновом режиме, каждое из них будет открыто в собственном окне. Однако переключаться между задачами окажется практически невозможно, ведь для этого потребуется переходить из окна в окно, а встроенных средств управления окнами в Иксах-то и нет.

О менеджерах окон

Чтобы получить возможность управления окнами, над X-сервером следует запустить одну из клиентских программ, именуемых оконными менеджерами, менеджерами окон или, менее точно, их диспетчерами (Window Manager). Именно они, как нетрудно догадаться из названия, отвечают за манипуляции над открытыми окнами: их закрытие, масштабирование, минимизацию, перемещение, а также переключение между окнами. Кроме того, многие оконные менеджеры обеспечивают и дополнительные возможности: вызов контекстных меню, различные инструментальные панели и пиктограммы запуска программ на рабочем столе, а также поддержку виртуальных экранов (рабочих столов). В компетенции менеджера окон и средства для выхода из графического режима штатным образом.

Вообще, именно от оконного менеджера зависит функциональность рабочей среды в X-сеансе, почему вопросу выбора такой программы следует уделить достаточное внимание. Однако прежде надо ознакомиться с возможностями разных менеджеров окон, а для этого разобраться, как их запускать. Сделать это можно в командной строке эмулятора терминала отдачей соответствую-

щей команды, обычно совпадающей с именем оконного менеджера. После этого терминальное окно, лишенное в голых Иксах всяких управляющих элементов, волшебным образом преображается: на нем появляются строка заголовка, управляющие пиктограммы (максимизации, минимизации, закрытия и т. д.), линейки прокрутки и прочие атрибуты. С рабочего стола по щелчку кнопок мыши становятся доступными контекстные меню. А самих рабочих столов может оказаться несколько, переключение между ними обеспечивается инструментальной панелью (в нижнем правом углу). Если же выйти (посредством соответствующего пункта контекстного меню) из оконного менеджера, внешний вид X-сеанса возвращается к первозданному состоянию.

Есть, однако, более простой способ запуска менеджера окон — вписать соответствующую команду в один из файлов инициализации, общесистемный или пользовательский. Первый — /usr/X11R6/lib/X11/xinit/xinitrc — по умолчанию уже содержит некоторый набор таких команд, выглядящий следующим образом:

```
twm &
xclock -geometry 50x50-1+1 &
xterm -geometry 80x50+494+51 &
xterm -geometry 80x20+494-0 &
exec xterm -geometry 80x66+0+0 -name login
```

Эти команды обеспечивают запуск оконного менеджера twm и в фоновом режиме часов и нескольких терминальных окон. Последнее из них, открываемое командой

```
exec имя_терминала
```

запускается в активном режиме и выступает в качестве главного: закрытие его (командой `exit` в командной строке) приводит к выходу из оконного менеджера.

Пользовательский файл инициализации Иксов ~/.xinitrc автоматически в домашнем каталоге не появляется, его нужно создать (в любом текстовом редакторе). И записать в нем нечто вроде

```
exec имя_менеджера
```

предварив ее требуемыми директивами для запуска терминальных окон, программы, обеспечивающей работу колеса прокрутки мыши, и т. д. А на первых порах, для знакомства с возможностями различных управляющих программ, можно поместить их все, закрыв комментариями, которые последовательно снимать по мере изучения отдельных экземпляров этого класса.

Об организации десктопа

Существуют два стиля организации рабочего места в какой-либо графической среде. Первый — это организация многих перекрывающих друг друга окон и переключение между ними. Второй — это серия виртуальных десктопов, каждый из которых занят одним полноэкранным приложением.

У пользователя UNIX-подобных систем есть выбор между двумя стилями: в зависимости от задач, да и просто личных привычек. Любой современный оконный менеджер, не говоря уже об интегрированных средах, поддерживает виртуальные десктопы в количестве, достаточном для обеспечения потребностей большинства пользователей. И даже без оконного менеджера вообще можно реализовать подобие полноэкранной работы: определить виртуальное разрешение экрана, скажем, вчетверо превышающее реальное, в каждой "четвертушке" открыть по терминальному окну, прописав их геометрию в файле X-ресурсов, и переключаться между ними мышью.

По многим причинам мне удобнее именно многодесктопный, а не многооконный стиль организации рабочего места. Благо, любой из оконных менеджеров (не говоря уже об интегрированных средах) предоставляет для этого массу возможностей. Так что далее будем ориентироваться именно на многодесктопную организацию рабочего места.

С этих позиций и рассмотрим несколько оконных менеджеров.

FLWM, или предел минимализма

FLWM можно назвать оконным менеджером для спартанца. Ибо построен он с простотой, достойной истинного гражданина древнего Лакедемона.

После его запуска через доли секунды появляется стерильно чистый экран ядовитого синего цвета. Ни пиктограмм, ни кнопок, ни какой-либо панели задач — ничего. На щелчок левой (а также правой, да и средней тоже) кнопкой мыши этот экран реагирует появлением меню со столь же аскетическим набором функций — создания нового десктопа, нового терминала и выхода. Запуск программ — из командной строки терминала. Что, соответственно, при нескольких запущенных приложениях быстро загромождает экран. Зато виртуальных можно создать, сколько душа пожелает. И переключаться между ними из того же меню по щелчку кнопки мыши.

В отличие от большинства оконных менеджеров, управляющие элементы окна в FLWM располагаются вертикально вдоль его левого края (рис. 16И.1). Это кнопки (сверху вниз) минимизации, втягивания, разворота по вертикальной и горизонтальной осям (включение обоих приводит к полноэкранному режиму), затем строка заголовка и кнопка закрытия окна. Все. Этим интерфейсные элементы исчерпаны.

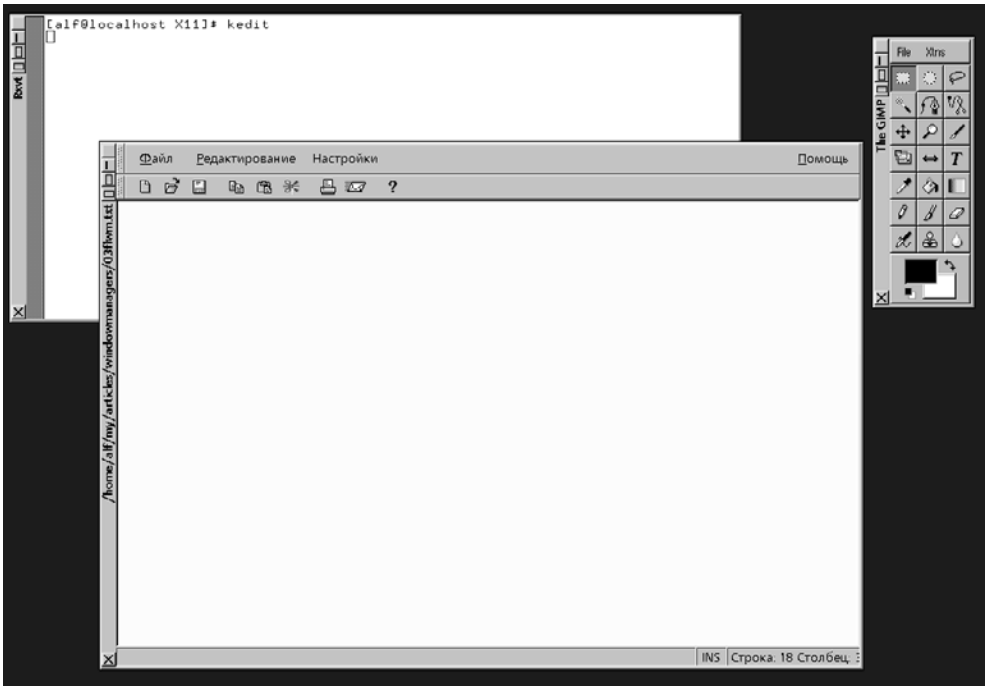


Рис. 16И.1. Оконный менеджер FLWM

Никаких визуальных средств настройки не обнаруживается. Модификация выпадающего меню не требует даже текстового редактора. Содержание его определяется составом каталога ~/.wmx/ (правда, его следует вручную скопировать из директории образцов типа /usr/local/share/flwm) или подобной. В этом каталоге обнаруживаются подкаталоги, соответствующие пунктам меню, а уже в них файлы, отвечающие подпунктам. Файлы пунктов меню — просто символические ссылки на исполняемые, запускающие соответствующие приложения. То есть добавить пункт меню можно просто командой

```
ln -s pasth2/тмя_исполняемого_файла
```

Еще в FLWM можно изменить цвет фона (по умолчанию он, как я уже говорил, отвратительно синий). Правда, делается это только через файл ресурсов (~/.Xresources). Потому что своего фона FLVM не имеет.

Все, больше ничто изменению не подлежит. Зато FLVM — практически идеальная среда для работы с большим количеством терминальных окон. Да и быстродействие его вне конкуренции. Ну, и на слабых машинах с ограниченными ресурсами FLVM может прийтись ко двору.

Blackbox как представитель семейства

Blackbox — предельно простая и очень компактная среда (размер исполняемого файла менее 200 Кбайт). Для своей установки и работы она не требует никаких специальных высокоуровневых библиотек типа Qt или Gtk. Тем не менее, возможностей для пользователя она предоставляет вдоволь (рис. 16И.2):

- неограниченное количество виртуальных рабочих столов (Workspaces), создаваемых при необходимости легким движением мыши;

Рис. 16И.2. Оконный менеджер Blackbox — общий вид

- удобные средства навигации между рабочими столами и окнами открытых приложений:
  - панель управления Toolbar;
  - контекстное меню, доступное по щелчку средней кнопкой мыши;
- легкодоступное (по щелчку правой кнопкой мыши) и элементарно настраиваемое контекстное меню для запуска приложений;
- возможность минимизации, максимизации и закрытия окон приложений, свертывания их до строки заголовка; при этом соответствующие управляющие элементы расположены удобно и привычно.

О возможностях настройки Blackbox следует сказать особо. Элементарное его конфигурирование выполняется непосредственно с рабочего стола. Здесь можно изменить модели фокусировки окон (по щелчку мышью, по перемещению курсора, автовсплытие) и их открытия (каскадное, упорядоченное в любом направлении), положение панели управления и некоторые другие параметры.

Через главное же контекстное меню настраивается и внешний вид Blackbox с помощью выбора одного из имеющихся в комплекте стилей. В это понятие входят цветовая гамма и характер заливки (сплошная, градиентная) рабочего стола и прочих интерфейсных элементов, шрифт меню, оконных титулов и прочего. Правда, свойства эти сквозные, приписать индивидуальный стиль отдельному рабочему столу не получается. Как нельзя и использовать фоновые изображения для них.

Более тонкие настройки выполняются традиционным способом — правкой конфигурационных файлов. Главным из них является ~/.blackboxrc, который при первом же запуске Blackbox автоматически образуется в домашнем каталоге пользователя. Именно в нем прописаны модели фокусировки и открытия окон, положение панели управления, количество и имена рабочих столов.

Кроме того, в ~/.blackboxrc указываются пути к файлу, определяющему состав главного меню, и к стилевому файлу, ответственному за внешний вид. По умолчанию это /usr/X11R6/share/Blackbox/menu и /usr/X11R6/share/Blackbox/styles/имя_стиля соответственно. Редактированию пользователем они, естественно, не поддаются. Однако ничто не мешает скопировать целиком /usr/X11R6/share/Blackbox/ в свой пользовательский каталог:

```
cp -R /usr/X11R6/share/Blackbox/ ~/.Blackbox/
```

и развлекаться с ним в свое удовольствие. В частности, переопределить состав главного меню можно просто в текстовом редакторе. Для этого достаточно внести в него, после строки

```
[begin] (Blackbox 0.61.x)
```

требуемые приложения в следующем формате:

```
[exec] (rxvt)     {rxvt}
```

Можно организовать и субменю любой степени вложенности:

```
[submenu] (Graphics)
[exec] (XV)           {xv}
[exec] (The GIMP)     {gimp}
[exec] (Image Magick) {display}
[end]
```

Ничуть не сложнее и переопределить стили внешнего вида, взяв за основу любой подходящий или создав свой собственный.

Ну и, конечно, нужно не забыть изменить пути к соответствующим файлам в ~/.blackboxrc. После чего вопрос с настройками Blackbox можно считать законченным.

Не знаю, удалось ли мне убедить читателя в несравненных достоинствах Blackbox. Можно отметить некоторую сложность клавиатурной навигации по открытым окнам и рабочим столам: клавишные комбинации, описанные в документации к программе, далеко не всегда работают. Ну и эстетическое впечатление ослабляется отсутствием возможности определить фоновые изображения для рабочих столов.

Однако эти ограничения изначального Blackbox полностью компенсируются многочисленными дополнительными компонентами, разработанными для повышения его функциональности и распространяемыми самостоятельно. Среди них:

- `bbrun` — удобная строка мини-терминала, поддерживающая историю команд в виде выпадающего меню;
- `bbpager` — средство визуального управления окнами, открытыми на виртуальных рабочих столах;
- `bbkeys` — система определения "горячих клавиш";
- `bbrb` — менеджер фоновых изображений.

Не обошлось и без интерактивного средства конфигурирования Blackbox, программы bbconf, позволяющей в интерактивном режиме изменять меню этого оконного менеджера, переопределять "горячие клавиши" и т. д. Аналогичное назначение имеет и пакет toolbox — комплексное средство для настройки Blackbox. Впрочем, мне все равно кажется, что настройка вручную более удобна и трудностей не доставляет...

Определение "горячих клавиш" не показалось мне очень уж удобным. Оно требует сначала запуска программы bbkeysConfigC, в которой (в терминальном режиме) нужно определить все клавишные комбинации, которые предполагается задействовать в качестве "горячих", а уж потом посредством bbconf приписать им некоторые действия (перемещения по рабочим столам, например). Однако в результате этого создается новый конфигурационный файл ~/.bbkeysrc, который легко отредактировать вручную, например, таким образом:

```
KeyToGrab(1), WithModifier(Mod1), WithAction(Workspace1)
...
KeyToGrab(#), WithModifier(Mod1), WithAction(Workspace#)
```

для легкой навигации по рабочим столам.

Чтобы постоянно пользоваться "горячими клавишами", средствами управления окнами и мини-терминалом, соответствующие программы можно за-

гружать одновременно с Blackbox. Для этого вызывающие их команды нужно внести в файл ~/.xinitrc:

```
#!/bin/sh
bbrun &
bbkeys &
bbpager &
exec blackbox
```

Функциональная ограниченность Blackbox (а главное, видимо, практическое прекращение его развития) вызвала к жизни несколько сделанных по его образу и подобию оконных менеджеров, например: openbox, fluxbox и hakedbox. По интерфейсу они весьма мало отличаются от оригинала, и, более того, способны работать с большинством его дополнительных модулей.

WindowMaker: как делать окна

Многим памятна (по крайней мере, по литературным источникам) замечательная платформа Next — детище Стива Джобса, созданное им в период отрыва от Apple с ее Macintosh. Являвшее собой диалектическое единство железа (черный кубик на базе процессора Motorolla 68040), операционной системы (базировавшейся на микроядре Mach) и графического пользовательского интерфейса (последние два компонента получили имя NextStep).

Созданная в конце 80-х, платформа Next далеко опередила свое время, как с точки зрения потребностей массового пользователя, так и возможностей. И потому довольно быстро прекратила свое существование. Но дело ее не пропало. С одной стороны, NextStep послужил прототипом Mac OS X — UNIX-подобной системы. С другой же стороны, идеи NextStep легли в основу двух свободных оконных менеджеров для системы X: AfterStep и WindowMaker. Последний и будет предметом настоящего раздела.

WindowMaker стандартно входит в большинство дистрибутивов Linux (**http://www.windowmaker.org/**). Так что на получении и установке его я останавливаться не буду. Замечу только, что перед первым его использованием требуется настройка путем запуска файла /usr/X11R6/bin/wmaker.inst от лица того пользователя, который желает иметь его своим оконным менеджером по умолчанию. Этот конфигуратор, во-первых, вносит необходимые изменения в пользовательский файл инициализации Иксов ~/.xinitrc, а во-вторых, создает в домашнем каталоге пользователя подкаталог ~/GNUstep/, содержащий основные файлы конфигурации WindowMaker с некими умолчальными его настройками.

Первое, что привлекает внимание после запуска WindowMaker (рис. 16И.3), — крупные трехмерные кнопки вдоль правого края экрана, образующие управ-

ляющую панель, именуемую доком (Dock). Двойной щелчок левой кнопкой мыши на большинстве входящих в панель кнопок запускает приложения (их набор прописан в конфигурационных файлах и с реально установленными в системе программами никак пока не соотносится) либо конфигурационные утилиты.

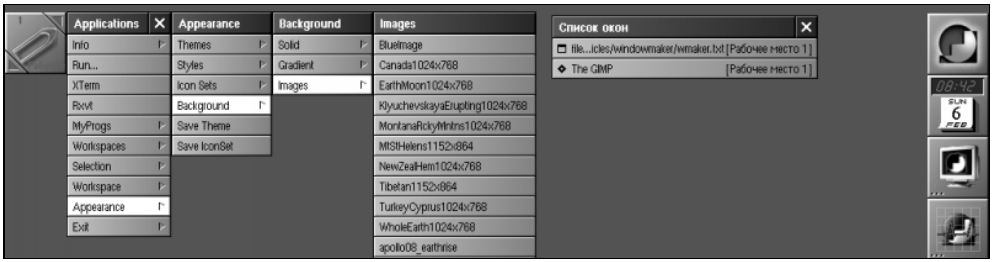


Рис. 16И.3. WindowMaker — первый запуск

Правда, верхняя кнопка (WMDock) по умолчанию ничего не запускает, а служит для управления положением панели на экране: ухватив за нее, панель можно перемещать вверх-вниз или с правой на левую сторону экрана (на верхнюю или нижнюю нельзя). Но к ней можно привязать какое-либо приложение, которое требуется запускать при старте Window Maker.

Вторая сверху кнопка (ASClock) — это просто индикатор времени и даты. Третья (xterm) запускает одноименный X-терминал (но ей можно приписать и любую другую терминальную программу). А четвертая (WMPefs) служит для конфигурирования Window Maker.

Одинокая кнопка слева вверху (**Clip**) выполняет двоякую функцию: переключения (но не создания) виртуальных экранов или, в терминологии программы, рабочих пространств (Workspace), и прикрепления (пришвартования, в терминологии оригинала) приложений.

Все остальное пространство экрана свободно и представляет собой рабочий стол. Щелчок на нем правой кнопкой мыши вызывает контекстное меню, правой — список запущенных приложений. Эти всплывающие панели не исчезают, в отличие от KDE или Windows, самопроизвольно. Щелчок левой кнопкой никакого действия на них не оказывает: чтобы закрыть всплывающую панель, следует щелкнуть мышью на заголовке и затем — на появившемся крестике. Либо щелкнуть соответствующей (правой или средней) кнопкой вне меню на рабочем столе.

Из контекстного меню создаются (в любом количестве, через пункты **Application | Workspace | Создать**) и удаляются новые виртуальные экраны. Переключение экранов сопровождается объемной анимацией их названий в

центре экрана. Однако поскольку название экрана по умолчанию русское (**Рабочее место #**), то при отсутствии соответствующего шрифта может анимироваться абракадабра. Впрочем, название экрана легко изменяется щелчком правой кнопки мыши на **Clip**. Открытые приложения по умолчанию не переходят на новый экран.

Для запуска приложений, кроме кнопок на панели и стандартного терминала, используется также контекстное меню рабочего стола. Этой цели в нем служит пункт **Run**, запускающий мини-терминал. В командной строке и вводится имя исполняемого файла нужной программы.

Тем не менее, основной способ запуска программ — через кнопки дока. Однако о пополнении управляющей панели необходимыми для работы кнопками следует позаботиться самому.

Можно запустить приложение, выбрав из контекстного меню рабочего стола упомянутый пункт **Run**. Одновременно с открытием окна приложения на рабочем столе появляется его пиктограмма, внешне аналогичная кнопкам панели. Она захватывается мышью и просто перетягивается в панель. Где и остается на веки веков, если вам не придет в голову ее истребить. Кнопка захватывается мышью и перетягивается за пределы панели на рабочий стол, где она рассыпается в прах.

С некоторыми приложениями (например, из KDE, или с программами, базируемыми на Motif, типа NEdit) этот способ не проходит. Чтобы встроить кнопки для таких приложений в док, требуется выполнить следующие действия:

1. Запустить нужное приложение
2. По щелчку правой кнопки мыши на заголовке ее окна вызвать контекстное меню.
3. Выбрать в нем пункт **Свойства окна**.
4. Перейти к подпункту **Дополнительные параметры**.
5. Отметить опцию **Эмулировать значок приложения**.
6. Сохранить эту установку.
7. Перезапустить приложение.
8. При следующем его запуске появляется та самая пиктограмма, которую можно перетащить в док.

Есть еще способ — правка вручную конфигурационных файлов, о чем скажу чуть позже.

Окно запущенного приложения по умолчанию имеет в строке заголовка два управляющих значка: минимизации слева и закрытия справа. По щелчку правой кнопкой мыши на строке заголовка окно можно развернуть на полный экран, свернуть (минимизировать), втянуть (сократить до строки заго-

ловка), выделить, переместить, закрыть (нормальное завершение программы) и убить (аварийное завершение программы, команда `kill`).

По щелчку же на титульной строке окна осуществляется доступ к свойствам и параметрам. Свойства — это атрибуты окна, такие как наличие/отсутствие строки заголовка, кнопок закрытия и минимизации, изменяемость размера и прочее, а также исходное рабочее место (программу можно жестко привязать к любому из существующих виртуальных десктопов). Параметры — это положение на рабочем столе (всегда вверху или внизу), а также присутствие везде (как я уже говорил, по умолчанию каждое приложение существует только на том виртуальном экране, на котором оно открыто).

WindowMaker имеет богатые возможности настройки, осуществляемой тремя взаимодополняемыми способами. Базовые настройки осуществляются через кнопку **WMPefs**. Она вызывает окно конфигурирования WindowMaker (рис. 16И.4), где можно настроить:

- позицию открываемых окон:
  - автоматически;
  - случайно;
  - каскадом;
  - ручками;

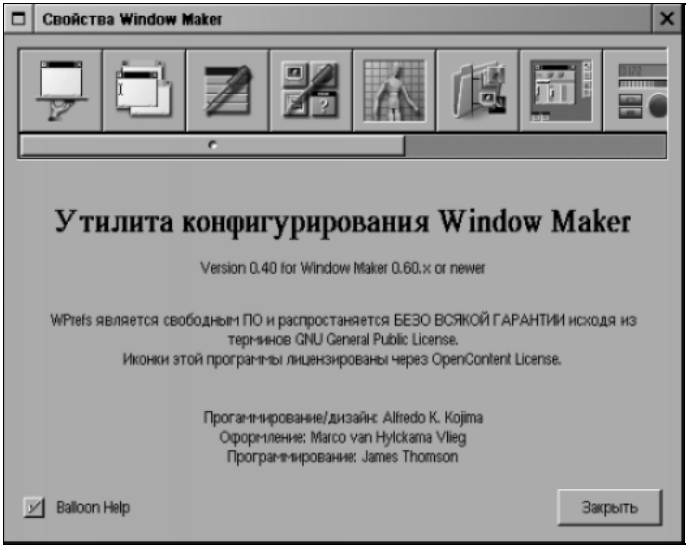


Рис. 16И.4. Окно конфигурирования WindowMaker

- фокусировку окна (щелчком или вслед за курсором мыши), а также определить время автоматического всплывания окна (от 10 до 800 мс, по умолчанию — никогда);
- выравнивание подменю в стиле Windows, ниспадающим каскадом (рис. 16И.5) или по верхнему краю (рис. 16И.6);

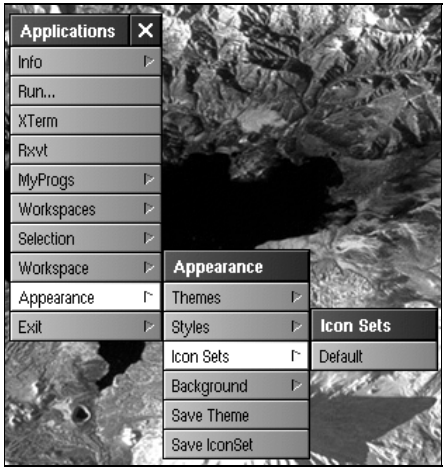


Рис. 16И.5. Настройка выпадающих подменю в стиле Windows

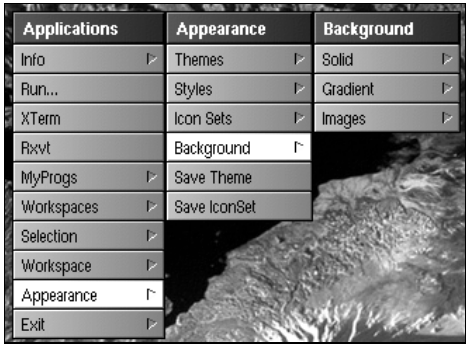


Рис. 16И.6. Настройка выпадающих подменю по верхнему краю

- расположение минимизированных окон и их выравнивание, а также размер экранных кнопок и иконок (от 24×24 до 96×96 пикселов);

- появление и характер всплывающих подсказок (по умолчанию отключены вообще);
- пути поиска графических файлов для пиктограмм и фоновых изображений);
- навигацию по виртуальным экранам (циклическую, с возвратом на первый экран после последнего, или с открытием нового экрана после последнего существующего); здесь же можно отключить **WMClip** и **WMDock** (а вот включить обратно их можно только ручной правкой одного из конфигурационных файлов).

Имеются также средства настройки внешнего вида иконок, окон, горячих клавиш, свойств мыши и пр. А главное — настройка меню приложений, о чем следует сказать подробнее.

В меню приложений (строка **Applications** контекстного меню рабочего стола) можно добавить (и, разумеется, удалить) пункты первого уровня, которые могут содержать подменю любого уровня вложенности (рис. 16И.7). И с любым количеством пунктов, которым приписываются любые команды, достаточно указать ее имя (или полный путь).

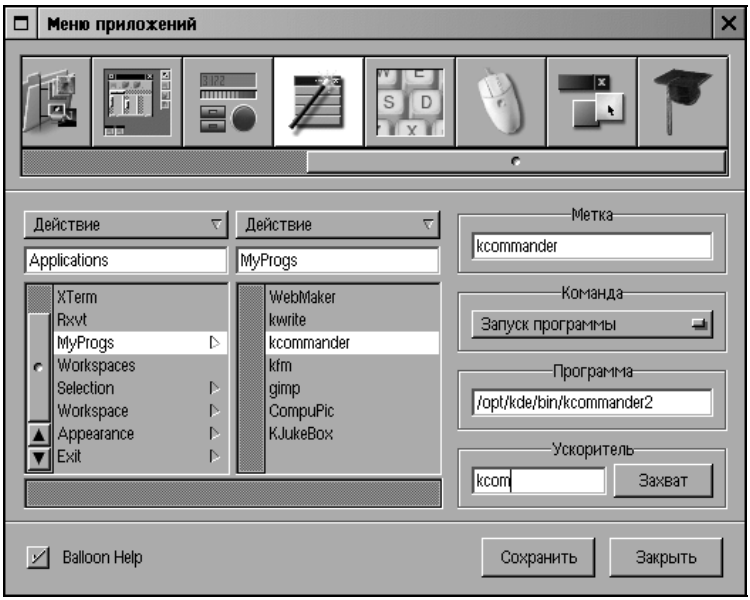


Рис. 16И.7. Настройка содержимого контекстных меню

Кроме того, здесь же, кроме выхода из WindowMaker и его перезапуска, можно (в подпункте **Switch to**) прописать вызов любого доступного окон-

ного менеджера, также с привязкой к горячей клавише. Делается это абсолютно так же, как и вызов прикладных программ. При этом переключение в другой оконный менеджер или графическую среду (например, KDE) происходит с сохранением всех открытых приложений. К сожалению, обратная процедура (то есть, скажем, возврат из KDE в WindowMaker) невозможна.

Дополнительные настройки внешнего вида WindowMaker можно выполнить из всплывающего меню **Applications** (подменю **Appearance**, а не **Workspaces**, как можно было бы ожидать). Здесь можно определить:

- тему целиком (из фиксированного набора, который легко пополнить);
- стили окон и меню (из списка в более чем две дюжины позиций);
- установки иконок;
- фоновое оформление (сплошная или градиентная цветовая заливка, изображения в любом распространенном растровом формате); фон становится общим для всех виртуальных экранов, как существующих, так и создаваемых позднее.

Из комбинации всех этих элементов легко создать собственную тему и сохранить ее в виде файла для дальнейшего использования. Отсутствует только возможность изменения базового шрифта, это делается исключительно вручную.

А Window Maker, как и любой уважающий себя элемент UNIX-системы, позволяет точную ручную настройку путем правки конфигурационных файлов. Они, в количестве пяти штук, автоматически копируются в каталог ~/GNUstep/Defaults при исполнении первичного конфигуратора.

В файле ~/GNUstep/Defaults/WMGLOBAL, как явствует из его названия, определяются наиболее общие параметры. В частности, только здесь (я, по крайней мере, другого места не нашел) можно переопределить шрифт для элементов рабочего стола.

В файле ~/GNUstep/Defaults/WindowMaker указываются стили меню и икон, цветовая палитра, пути для икон и фоновых рисунков и многие другие параметры, определяемые через WMPrefs. Через который их и лучше изменять при необходимости. Однако если вы случайно отключите WMClip и WMDock, единственный способ вернуть их — отыскать в этом файле строки

```
DisableClip = YES;
DisableDock = YES;
```

и заменить `YES` на `NO`. Или просто удалить, ибо `NO` — значения обеих этих строк по умолчанию. Только сделать это нужно либо в консольном режиме, либо переключившись в другой оконный менеджер; иначе WindowMaker при выходе восстановит значения текущего сеанса.

Файл ~/GNUstep/Defaults/WMRootMenu описывает содержание меню **Applications**, вызываемого с рабочего стола и также определяемого через **WMPrefs**. Содержание файла ~/GNUstep/Defaults/WMWindowAcodeributes ясно из названия. Ну, а файл ~/GNUstep/Defaults/WMState — описание панели **WMDock** и рабочих мест. Именно его легко отредактировать для внесения новых приложений в панель **WMDock**.

Каков же вывод из всего рассмотренного? В целом, по-моему, сугубо положительный. Конечно, на первый взгляд Window Maker выглядит непривычно для пользователя Windows. Однако этим и пробуждает здоровое любопытство. Тем более, что удовлетворение его не связано ни с какими принципиальными сложностями.

Возможных настроек почти достаточно для удовлетворения личных эстетических идеалов. А также для внесения разнообразия в производственный процесс. Устойчивость — на уровне, основные манипуляции с приложениями выполняются (при наличии минимальной привычки и после несложных настроек) легко и быстро. Для WindowMaker написано большое количество дополнительных модулей, расширяющих его функциональность, многочисленные темы для разнообразия внешнего вида и средства управления ими. Этот комплекс опциональных средств приближает его возможности к таковым интегрированных сред. Кроме того, WindowMaker принят в качестве базы для GNUStep (application framework), предполагаемого к использованию в разработках проекта GNU.

IceWM: гибкость и настраиваемость

Внешний вид IceWM весьма похож на Windows: в нижней части экрана панель запуска приложений, она же переключатель виртуальных экранов и панель задач (рис. 16И.8). Имеет аналог пресловутой кнопки **Start**, правее которой — кнопки для переключения задач и запуска нескольких своеобычных приложений (терминала, Netscape Navigator и т. д.).

Затем кнопки переключения виртуальных экранов, их по умолчанию четыре. С правой стороны панели процессорный монитор и часы. В общем, достаточно аскетический набор.

Щелчок левой кнопкой мыши на девственно чистом рабочем столе вызывает список запущенных задач и виртуальных экранов, правой — меню, аналогичное по содержанию таковому стартовой кнопки; к средней кнопке привязан список **Windows List**.

Стартовое меню содержит следующие пункты:

- пару терминалов, несколько приложений и системных утилит;
- отдельные пункты для приложений KDE и GNOME;

- переключатель оконных менеджеров;
- переключатель тем.

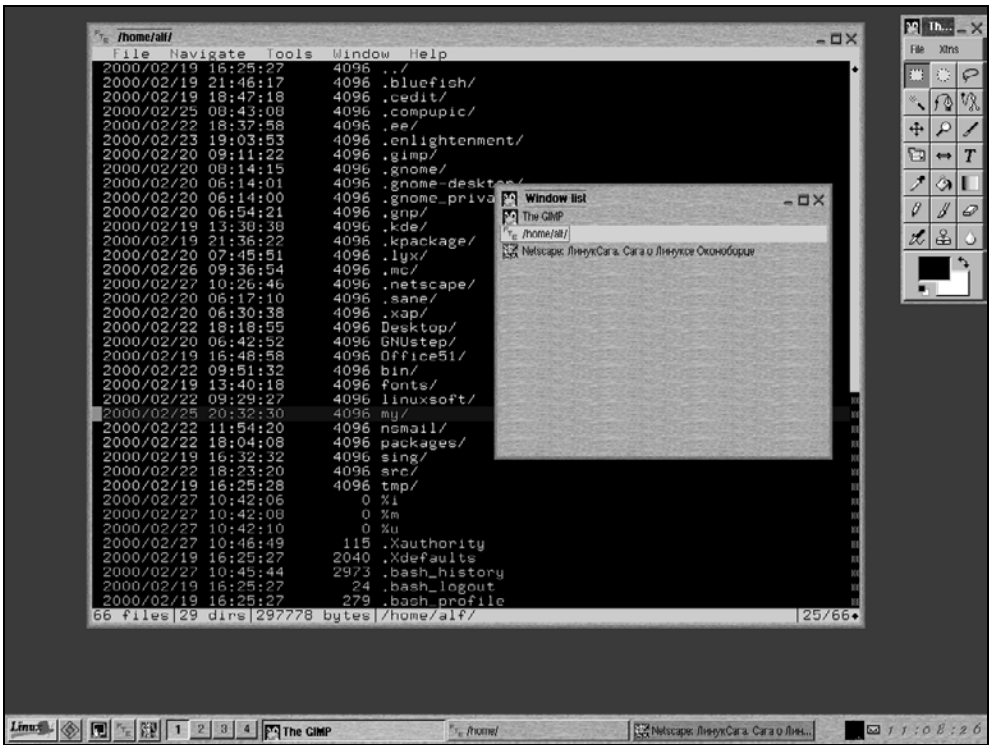


Рис. 16И.8. Оконный менеджер IceWM

Никакого средства быстрого запуска программ, типа мини-терминала из KDE, поначалу увидеть не удастся. Напрашивается вывод (сразу скажу, ошибочный), что для запуска программ следует использовать либо стартовое меню, либо окно терминала. Первое содержало лишь ограниченный набор приложений. И простых средств для его пополнения не обнаруживалось. Ну, а второй способ, очевидно, быстро приводит к загромождению доступных экранов. Способа увеличить количество последних также пока не видно.

Средства визуального конфигурирования системы сводятся к выбору из фиксированного набора тем. Изменяющих внешний вид окон и шрифты меню, в некоторых случаях — и цвет фона. Однако при перезапуске настройки тем сохраняться упорно не желают.

То есть первое впечатление от IceWM достаточно удручающее. И может и остаться таковым, если не обратиться к каталогу /usr/X11R6/lib/X11/icewm/. Где можно видеть сразу четыре конфигурационных файла: preferences, menu, toolbar и winoptions. Чтобы сделать их доступными для редактирования пользователем, эти файлы должно скопировать в свой домашний каталог (сами по себе они там не образуются). Зато потом настройке поддается практически все.

В файле preferences все пункты (а их там чуть ли не две сотни) по умолчанию отключены. Снимая значок комментария и устанавливая переключатели (1/0) в соответствующие значения, можно настроить:

- массу параметров активизации окон;
- определить шрифты и цвета всех интерфейсных элементов;
- установить автоскрытие панели задач и состав иконок на ней (таких как часы, мониторинг загрузки процессора и т. д.) и многое другое.

Очень интересная опция — возможность установки двойной высоты управляющей панели. В этом случае панель разбивается на две части по вертикали. В верхней половине — стартовая кнопка, кнопки запуска приложений и часы; между ними появляется своего рода командная строка (аналог минитерминала из KDE). В нижней части — кнопки переключения виртуальных экранов (которых можно установить вплоть до двенадцати) и значки запущенных приложений; последние могут отображаться также и на рабочем столе. И в том, и в другом случае они представлены только на родительском виртуальном экране.

Не могу не отметить чрезвычайно удобную работу с командной строкой (даже несмотря на отсутствие истории команд и поддержки автоматического дополнения табулятором). В нее можно вставить выделенный фрагмент (стандартным способом, средней кнопкой мыши) или фрагмент, предварительно скопированный (сочетанием клавиш <Ctrl>+<v>). Разумеется, можно и просто набрать команду с клавиатуры, по буквам.

В файле menu, как не сложно догадаться, можно настроить состав стартового меню, вызываемого по кнопке Linux. Делается это достаточно просто: пункт для запуска программы определяется словом `prog`, после чего следуют название приложения, имя иконки и имя исполняемого файла. А чтобы объединить несколько программ в одну папку (то есть сделать иерархическое меню), указывается слово `menu`, после которого, в фигурных скобках, перечисляются все потребные `prog`. Как это показано в примере:

```
prog rxvt xterm rxvt -bg black -cr green -fg white -C -fn 9x15 -sl 500
     prog xterm xterm xterm
     prog fte fte fte
     prog Netscape netscape netscape
```

```
prog Gimp gimp gimp
separator
menu Mandrake magic {
        prog "Mandrake Update" updates-mdk MandrakeUpdate
        prog "rpmdrake" rpmdrake rpmdrake
        prog "DrakConf" DrakConf DrakConf
        prog "Mandrake Doc" doc-mdk netscape
        /usr/doc/mandrake/index.htm
        prog "Mandrake News" news-mdk netscape
        http://www.linux-mandrake.com/en/fdoc.html
}
separator
```

Ну, а в файле toolbar, естественно, можно вывести иконки запуска приложений непосредственно с управляющей панели. Делается абсолютно так же:

```
prog rxvt xterm rxvt -bg black -cr green -fg white -C -fn 9x15 -sl
500
prog FTE fte fte
prog Netscape netscape netscape
prog WebMaker /home/alf/.icewm/icons/webmaker.xpm webmaker
prog Kedit /home/alf/.icewm/icons/kedit.xpm kedit
prog Kwrite /home/alf/.icewm/icons/kwrite.xpm kwrite
prog XNC /home/alf/GNUstep/.AppInfo/WindowMaker/xnc.XNC.xpm xnc
prog Gimp gimp gimp
prog Compupic /home/alf/.icewm/icons/cpicicon-16x16.xpm compupic
```

Наконец, в файле winoptions можно настроить параметры всяческих приложений.

Поддаются ручной настройке и темы рабочего стола. Для этого в каждой теме имеется файл default.theme (где задаются цвета, бордюры и пр.) и множество графических файлов для интерфейсных элементов. Все это совершенно элементарно поддается ручному редактированию. В частности, не составляет труда изменить цвета любых элементов, шрифты и т. п. Можно также определить фоновое изображение, причем оно отнюдь не обязано быть в формате XPM.

Для сохранения темы после окончания текущего сеанса путь к соответствующему файлу default.theme нужно прописать в preferences. Там же, при необходимости, можно разрешить/запретить центрирование фонового рисунка и его размножение. Дополнительные (помимо входящих в состав дистрибутива) темы для IceWM можно найти по адресу **hhtp://icewm.tux.nu**.

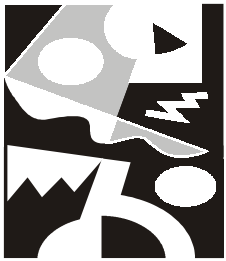
Все изложенное не могло не повлиять на мои выводы о IceWM в целом. А именно: это очень гибкая и настраиваемая система. Разумеется, ручная правка конфигурационных файлов занимает несколько больше времени, чем, скажем, перетаскивание иконок загруженных приложений в WindowMaker. Однако никаких принципиальных сложностей не представляет, все параметры конфигурации предельно прозрачны. Конечно, при эпизодическом применении IceWM она может быть неоправданна. Однако если предполагается постоянное использование этой среды, затратить полчаса на ручную доводку не сложно.

IceWM сочетает в себе следующие достоинства:

- эффективные средства запуска программ, как из удобной и компактной панели, так и из командной строки мини-терминала, без необходимости вызова терминального окна;
- удобная и разнообразная манипуляция запущенными приложениями; достаточное количество виртуальных экранов с удобными способами перемещения между ними;
- с эстетической точки зрения среда может быть обустроена как угодно.

Быстродействие IceWM близко к рекордному среди всех оконных сред. Неоднократно отмечалась также ее устойчивость (что подтверждается моими впечатлениями). Короче говоря, если есть необходимость в гибко настраиваемой системе, и при этом не пугает необходимость покопаться руками в конфигурационных файлах (преувлекательное занятие, доложу я вам), то IceWM это именно то, что нужно.

Глава 17



KDE: интеграция десктопа

- Проблема выбора.
- KDE — почему бы и нет.
- KDE как он есть.
- Установка.
- Собственные средства настройки.
- Конфигурационные файлы.
- Детали настройки клавиатуры.

Эта глава посвящена самой интегрированной рабочей среде графического режима — KDE (K Desktop Environment), а также тому, как с ним бороться, то есть настраивать. Но сначала несколько слов о том, почему эта самая борьба с KDE может стать необходимостью для пользователя.

Проблема выбора

Выбор среды обитания при работе в графическом режиме (то есть в Иксах) — дело сугубо личное, я бы даже сказал — интимное. И суть его сводится, как и практически все в POSIX-системах, к старому анекдоту про парашютиста. На вопрос, не страшно ли ему во время прыжка, тот ответил отрицательно и объяснил, почему. Позволю себе пересказать суть его объяснения в сокращенном варианте (на самом деле старыми мастерами советского анекдота все рассказывалось гораздо подробнее).

Итак, наш парашютист ответил: "Когда я прыгаю, у меня есть два выхода: или парашют раскроется, или не раскроется. Если раскроется — все хорошо, а если не раскроется, остается два выхода: или я останусь жив, или разобьюсь насмерть. Если я останусь жив — все хорошо, если разобьюсь — остается два выхода: или я попаду в рай, или я попаду в ад. Если я попаду в рай — все хорошо, а если я попаду в ад, остается два выхода: или черт меня

не съест, или черт меня съест. Если черт меня не съест — все хорошо, а если черт меня съест — ну один-то выход у меня все равно остается!"

Это объяснение каждый начинающий POSIX'ивист должен запомнить, как молитву: в любом затруднительном положении у него есть как минимум два варианта решения проблемы. И если ему покажется, что решения нет, хоть один вариант по зрелом размышлении найдется обязательно.

Так что проблема выбора графического интерфейса начинается с определения того, что же нужно пользователю: просто оконный менеджер или действительно интегрированная среда (graphic desktop environment), называемая также просто десктопом (desktop). Различия между ними очевидны: первый класс программ предоставляет пользователю возможность управления окнами: их открытия, закрытия, масштабирования, свертывания, перемещения, переключения между открытыми окнами.

В качестве дополнительных (обычно присутствующих в оконном менеджере, но отнюдь не обязательных) возможностей могут иметь место виртуальные рабочие столы или виртуальные разрешения оных, средства запуска приложений — иконки рабочего стола, контекстные меню, строки мини-терминала (то, что в KDE называется mini-cli), средства навигации по десктопам и окнам (типа панелей задач), различные дополнительные украшательства — фоны, обои и т. д. А также более или менее автоматизированные средства для конфигурирования всего этого. Однако повторяю: все, что выходит за рамки управления окнами, является сугубо опциональным и может отсутствовать в оконном менеджере, а настройка его вполне может осуществляться только прямым редактированием конфигурационного файла.

Интегрированный десктоп, разумеется, включает в себя средства управления окнами — собственные, как в KDE, или привлеченные из дружественных (то есть совместимых) оконных менеджеров, как в GNOME (понятно, что без управления окнами работа в оконной среде X попросту невозможна). Но тут уже виртуальные десктопы, средства запуска программ и навигации по ним — всякого рода панели, стартовые и контекстные меню, пиктограммы, наборы тем и прочие красивости становятся непременными атрибутами рабочей среды. Плюс более или менее обширный набор интегрированных приложений (почему он и называется интегрированной средой), как минимум: терминал, редактор, файловый менеджер. Ну и обязательным компонентом десктопа (без чего он не заслуживал бы этого наименования) — собственные (то есть графические же) средства сквозного конфигурирования его самого и всех его приложений.

Маленькое отступление

Среди старых (не по возрасту, а по стажу) пользователей POSIX-систем широко распространено мнение, что любые программы, выполненные в направлении UNIX Way (или True UNIX Way), должны настраиваться правкой конфигов в

> текстовом редакторе (а паче того, их созданием "с нуля"), все же остальное — от лукавого.
>
> С этим трудно не согласиться. Да, истинный POSIX'ивист всегда должен иметь возможность вмешаться руками в процесс настройки. Однако если речь идет о правке многих десятков конфигов (а в случае с KDE, как станет ясным из дальнейшего, именно так и есть), не проще ли в общем и целом положиться на собственный конфигуратор, а к ручной правке прибегать только в критических ситуациях? Ибо если GUI не может сам себя настроить средствами своего же GUI, то какой он Интерфейс Пользователя?
>
> И уж вообще нелепо, если с помощью GUI-конфигуратора можно настроить массу очень сложных параметров, а для какого-либо элементарного действия (например, изменения шрифта меню) требуется ручное редактирование rc-файла...

В случае выбора оконного менеджера для пользователя все хорошо. Правда, в его распоряжении оказывается огромное их количество, с интерфейсами разной степени простоты, построенными по различным принципам. Однако тут уж с проблемой выбора он должен справиться, хотя бы методом перебора вариантов.

А вот при выборе интегрированного десктопа перед пользователем два выхода. Первый — это строить такой десктоп самостоятельно, на основе более или менее простого оконного менеджера и тех приложений, которые он использует постоянно. Благо, многие из оконных менеджеров позволяют превратить себя в полноценную интегрированную среду (хотя и весьма индивидуальную) если не легким движением руки, то несложным редактированием своих конфигов. Здесь показателен пример fluxbox: благодаря механизму вкладок (tabs) совместно используемые приложения (например, терминал, текстовый редактор, браузер) легко объединяются в группы "по интересам". Так что если пользователь сделает такой выбор, все хорошо: остается только затратить должное количество времени на редактирование файла X-ресурсов, пользовательского ~/.xinitrc, конфигурационных файлов оконного менеджера и отдельных приложений. Если же это почему-либо не устраивает — остается второй выход: использование уже готового десктопа, их не так и мало. Из мне известных графических интерфейсов в этом качестве позиционируются CDE, XFce, GNOME и KDE. Однако первая — продукт коммерческий, и, насколько я знаю, не входит в состав ни одного дистрибутива Linux или свободной BSD-системы. А XFce, при всех своих несомненных (и многочисленных) достоинствах, в современном своем виде на роль интегрированной среды претендовать может с трудом: это скорее наиболее развитый (по сравнению с прочими оконными менеджерами) конструктор для собственноручного построения таковой. Так что на самом деле и тут остается только два выхода: GNOME или KDE.

Если пользователь решится на первый выбор, для него (надеюсь) все будет хорошо. Однако здесь я ему не советчик. Потому что GNOME — один из

немногих представителей класса графических интерфейсов, который мне активно не нравится.

Я вполне разделяю чувства поклонников элегантности преемников NextStep (Afterstep, WindowMaker) или строгой простоты семейства Blackbox, Openbox, Fluxbox (к слову, ничуть не менее элегантных, а последний еще и уникально функционален). Тем паче, что сам долгое время был в их числе (а периодически пользуюсь WindowMaker или Fluxbox и по сей день). Я готов понять любителей IceWM, сочетающего в себе простоту настройки с ее гибкостью. Я осознаю несравненную настраиваемость FVWM и его клонов, хотя и чисто теоретически. Мне столь же платонически очень нравятся идеи, заложенные в XFce, являющем близкий к идеальному баланс между минимализмом оконного менеджера и функциональностью полноценного десктопа. Наконец, в некоторых случаях мне представляется вполне приемлемым предельный аскетизм FLVM, которому, приложив чуть-чуть усилий, можно еще и придать некоторую элегантность.

Но, разрази меня гром, за все свои попытки общения с GNOME я не обнаружил в нем никаких привлекательных (для себя) черт. Начать с того, что это также не вполне интегрированная среда. Что, конечно, становится особенно понятным при сравнении с KDE, однако... Большая часть того, что интегрировано в GNOME и заслуживает всяческого использования, — создавалась до него, вне него и независимо от него. Тут вспоминаем GIMP, ведь именно для его разработки была придумана библиотека Gtk (что так и расшифровывается — GIMP Toolkit), послужившая базой для множества приложений, изначально с GNOME никак не связанных, от сугубо кросс-платформенного AbiWord до векторного графического редактора Sodipodi, автор которого озаботился столь же легкой интеграцией своего произведения в KDE, как и в GNOME.

Далее. Если KDE с каждой новой версией становится все быстрее, то GNOME — все задумчивее. А тенденция к разрастанию объема выражена во втором ничуть не меньше, чем в первом. Ну и, наконец, просто идеология: разработчики GNOME все больше и больше тяготеют к воспроизведению особенностей "самой великой ОС всех времен и народов".

Единственным основанием к использованию GNOME я вижу нежелание (или невозможность) плодить большое количество библиотек — так как без GIMP при мало-мальски существенной доле работы с графикой обойтись все равно не удастся, а он основан на той же библиотеке Gtk, что и GNOME. Хотя, строго говоря, как раз наоборот: библиотека Gtk создавалась специально для GIMP, а к GNOME ее прикрутили за отсутствием выбора. Впрочем, лично мне думается, что и тут XFce (основанная на той же Gtk) была бы более предпочтительна.

Впрочем, ругательных материалов о GNOME, особенно в современных его ипостасях, можно найти ничуть не меньше, чем хвалительных. Как уже бы-

ло сказано, мне лично хвалить GNOME не за что, а ругать его не возьмусь за слабым знанием.

Так что пользователь должен сделать свой выбор, опираясь на субъективные впечатления и объективные оценки источников. Однако если выбор его будет не в пользу GNOME, то еще один выход у него найдется. И выход этот — использование KDE.

KDE — почему бы и нет

Итак, KDE — единственный (повторяю, по моему мнению) по-настоящему интегрированный десктоп в мире Open Sources. Правда, базируется он на библиотеке Qt, имеющей (в том числе и) коммерческий статус. Однако ныне лицензионные споры относительно свободы ее отшумели, и лицензия, под которой Qt распространяется для некоммерческого использования, признана вполне GPL-совместимой. Так что у всех адептов чистоты Open Sources и Free Software "юридических" оснований не использовать KDE не осталось.

Другое дело, что многие не любят KDE за его масштабность. Действительно, эта среда предъявляет довольно высокие требования к аппаратуре, особенно к объему памяти: мало-мальски комфортная работа в современных версиях KDE возможна, начиная с RAM 256 Мбайт (лучше — больше, хотя начиная с 512 Мбайт, разницы уже почти не чувствуется). Однако столь ли страшны такие требования для современных машин?

Немало места занимают компоненты этой среды и на винчестере: моя обычная установка KDE (вместе с обязательной библиотекой Qt) тянет почти на 500 Мбайт. Но и это при современных объемах винчестеров не столь критично. Наконец, удовольствие от работы в KDE можно получить при разрешении экрана, начиная с 1024×768 и глубине цвета от 16 бит. Но ведь и это не проблема для нынешних видеокарт и мониторов.

Можно видеть, что при более или менее современной аппаратуре системные требования KDE не выглядят чем-то сверхъестественным. Другое дело, что это не лучший выбор для реанимации старого "железа". Так что пользователям безнадежно состарившихся (морально, не обязательно физически) машин придется обратиться в выбору между "легкими" оконными менеджерами.

Впечатление некоторой "задумчивости" при общении с KDE может дать старт самой среды и первый в сеансе запуск какого-либо приложения. Действительно, поскольку любое KDE-приложение задействует множество библиотечных функций, запуск его по определению не может быть быстрым. Однако столь ли это критично? Ведь сама среда и постоянно используемые

ее компоненты запускаются один раз в день (а на служебной машине, возможно, вообще загружены круглосуточно). А на тех программах, которые открываются и закрываются перманентно (например, терминальные окна), замедление не сказывается: повторный запуск любого KDE-приложения осуществляется на порядок быстрее, чем первый (оно и понятно: нужные библиотечные функции уже в памяти). Кроме того, в Linux, например, существует метод радикального ускорения старта KDE-приложений (и не только их) — так называемое предварительное связывание (prelinking). А в DragonFlyBSD аналогичная функция поддерживается на уровне ядра.

И вообще, если уж речь зашла о быстродействии, KDE в этом отношении ощутимо выигрывает у GNOME.

Наконец, KDE — одна из немногих программ в мировой истории, которая с каждой новой версией становится не только функциональней, но и быстрее. И это не слова, а реальность, которую не могли не заметить все пользователи, перешедшие в недавнее время с версий 3.2.X на 3.3.X (и затем — на 3.4.X и особенно на 3.5).

И последнее из широко распространенных предубеждений против KDE — его интерфейс с точки зрения визуального впечатления. Действительно, вид KDE не являет собой предел эстетического совершенства и вызывает неприятные для приверженцев True UNIX GUI ассоциации с внешностью "самой известной ОС". Однако с каждой новой версией KDE движется в сторону элегантности, и для него существует (и постоянно пополняется) большой набор тем и отдельных интерфейсных элементов, в том числе и весьма симпатичных (с ними можно ознакомиться на специальном сайте: **http://www.kde-look.org**). Ну, а родимые пятна Windows-подобия легко выводятся несложными настроечными действиями.

Да, еще, коль скоро я уж заговорил о настройках. В качестве недостатков KDE подчас отмечают сложность конфигурирования этой среды и способа ручных настроек. Это действительно так. Однако взамен среда предлагает очень развитые средства интерактивного конфигурирования. Тем не менее, способы ручной коррекции настроек и составят существенную часть этой главы: я постараюсь показать, что не так уж страшен черт, как его малюют.

До сих пор все аргументы в пользу KDE были, так сказать, от противного и сводились к тому, что этот десктоп не так уж плох, как о нем часто думают. Пора, однако, обратиться к позитиву и посмотреть, чем же он хорош.

Во-первых, функциональностью собственно оконного менеджера: средства управления окнами многочисленны и разнообразны. Во-вторых, полным ассортиментом средств запуска — от пиктограмм рабочего стола до строки мини-терминала, сохраняющей историю команд, от стартового меню в стиле Windows (K-меню) до дерева главной управляющей панели, не говоря уж о традиционных для Иксовых интерфейсов контекстных меню рабочего стола. В-третьих, удобством средств навигации между виртуальными рабочими

столами (их может быть аж 20 штук) и окнами открытых приложений, разнообразием способов "поднятия" и фокусировки окон.

Конечно, всем этим богатством современного пользователя удивить трудно, перечисленные средства в том или ином сочетании есть в любом развитом оконном менеджере. Однако лишь в редких из них можно найти полный их набор. А потом, многие из означенных возможностей впервые появились именно в KDE, например, строка мини-терминала с поддержкой истории команд (mini-cli).

Однако KDE имеет и уникальные возможности. И здесь в первую голову стоит упомянуть сквозное средство глобального конфигурирования среды обитания — Центр управления (KCC — KDE Control Center). Описанию способов настройки KDE будет посвящена изрядная часть этой главы. Пока же отмечу только, что с помощью KCC конфигурируется 99% того, что вообще может сконфигурировать самый привередливый пользователь. Оставшийся процент настроек, не поддающихся средствам KCC, легко изменяется редактированием конфигов.

Все глобальные параметры среды автоматически распространяются на любые KDE-приложения, причем даже те, что не входят в штатную поставку системы. Однако KDE-приложения можно настроить и индивидуально, и это будет иметь приоритет перед глобальными характеристиками.

Изобилие штатных приложений — другая особенность KDE: не покидая этой среды, можно найти и прекрасный эмулятор терминала, и полнофункциональный текстовый редактор, и файловый менеджер с браузером, и почтовый и FTP-клиенты, и множество графических и мультимедийных приложений, не говоря уже о мощном (и, что характерно, пригодном к использованию) наборе общесистемных утилит и средств разработки программ и Web-материалов. Число программ сторонних разработчиков, ориентированных на работу в среде KDE, наверное, учету не поддается (в этом можно убедиться на специальном сайте: **http://kde-apps.org/**) и охватывает абсолютно все области, для которых только существуют разработки Open Sources вообще. Кроме того, ряд программ имеет front end на базе библиотечных элементов Qt. А недавно появилась и возможность прикрутить KDE-интерфейс к движку Gekko — базе проприетарного Netscape, свободных браузеров Mozilla и Galeon. Все разработанные для KDE программы, включая front end, имеют стандартизированный интерфейс, который, как уже было сказано, может быть настроен глобально, вместе с параметрами самой среды.

В целом KDE выглядит даже перегруженным штатными приложениями, и это нередко отмечается как очередной недостаток данной среды. Да уж, что есть, то есть: кое-какие программы редко кем используются (ввиду наличия более продвинутых аналогов), иные же явно дублируют друг друга, и подчас дублирующие варианты, мягко говоря, далеки от совершенства (чтобы в

этом убедиться, достаточно просмотреть пункты **Графика** и **Мультимедиа** К-меню).

Однако... Вовсе не все компоненты полного набора KDE обязательны к установке, и нет препятствий к запуску из этой среды программ, основанных на других библиотеках. KDE — одна из немногих программ, новые версии которой не только обрастают новыми функциями, но и подвергаются урезанию, остаются только проверенные компоненты. Многие программы штатного комплекта принадлежат к числу лучших в своем классе (для примера — текстовый редактор kate, средство модемного соединения Kppp, почтовый клиент KMail) или просто держат пальму первенства в своей области (как тут не вспомнить konqueror — лучший файловый менеджер всех времен и народов).

Из первых двух особенностей KDE вытекает третья, и главная — ее самодостаточность. Которая, собственно, и позволяет назвать эту среду по-настоящему интегрированной. Подавляющее большинство своих задач пользователь может выполнить, не покидая KDE-десктопа, — вплоть до интерактивного редактирования общесистемных скриптов инициализации (что, впрочем, не значит, что это нужно делать, но возможность к тому вы имеете). А в грядущих версиях, по агентурным данным, KDE будет способно обходиться даже без Иксов. Впрочем, даже в современном виде KDE просматривается тенденция обходиться без общих Иксовых ресурсов (например, как станет ясно из следующего раздела этой главы, в нем имеются собственные системы управления шрифтами и клавиатурными раскладками).

И четвертое выгодное качество KDE — это стабильно поступательное, уже на протяжении многих лет, развитие. Я имел удовольствие наблюдать эту систему с самых первых версий (где-то с 1998 г.) и свидетельствую: она не только обрастала новыми функциями (это дело обычное), но становилась все стабильнее и быстрее. Совершенствуя при том свой интерфейс визуально, а эстетический момент отнюдь не последний в деле выбора среды обитания, по крайней мере, для меня.

Конечно, KDE еще не достигла состояния идеального десктопа и не свободна от некоторых недостатков. Однако я попытался показать, что они не критичны. Это позволяет рассматривать эту систему в качестве оптимального выбора при необходимости именно в интегрированной среде обитания.

KDE как он есть

Легко догадаться, что местом своего пребывания проект среда KDE имеет сайт **http://www.kde.org**, откуда может быть абсолютно безвозмездно скачана в виде исходных текстов (и бинарных пакетов для некоторых дистрибутивов Linux). В прекомпилированном же виде KDE входят в состав практически

любого дистрибутива Linux, претендующего на полнофункциональность (а во многих является и десктопом по умолчанию). Есть она в коллекциях бинарных пакетов и системах портов всех BSD-систем.

Для установки KDE необходимо озаботиться еще и получением библиотеки Qt, свободно распространяемый (для некоммерческого использования в системах XFree86 и Xorg) вариант которой берется в виде исходников с сайта разработчика (**http://www.trolltech.com/**). Впрочем, есть она и в дистрибутивах Linux и BSD-систем, нужно только следить за соответствием версий — они обычно достаточно четко коррелируют с версиями KDE.

Необходимы для работы KDE система X, а также немало дополнительных компонентов, включая главные графические и мультимедийные библиотеки, некоторые из которых в качестве зависимостей неизбежно тащат за собой библиотеки Glib и Gtk. Однако и все это хозяйство имеется в дистрибутивах и, скорее всего, устанавливается из них по умолчанию. Да и многие из дополнительных компонентов связаны с KDE "мягкими" (необязательными) зависимостями — в KDE разделение "жестких" и "мягких" зависимостей проводится очень последовательно.

Конечно, процесс самостоятельной сборки KDE — дело не из самых простых, как из-за сложности зависимостей дополнительных компонентов, так и с точки зрения временных затрат. Однако это один из тех немногих случаев, когда индивидуальная сборка способна дать значимый прирост производительности. Кроме того, она позволяет исключить явно ненужные пользователю зависимости из числа "мягких": в большинстве пакетных дистрибутивов Linux они по умолчанию задействованы в прекомпилированных сборках. Альтернатива ручной сборке — использование портов BSD-систем и портообразных систем Source Based дистрибутивов Linux, все они допускают значительные вариации персональных настроек, включая отсечение необязательных зависимостей.

Сама по себе среда KDE в виде исходников включает в себя около полутора дюжин пакетов, список и состав которых несколько варьируют от версии к версии. В современных версиях это выглядит примерно следующим образом:

```
kdeaccessibility-3.4.X.tar.bz2

kdeaddons-3.4.X.tar.bz2

kdeadmin-3.4.X.tar.bz2

kdeartwork-3.4.X.tar.bz2

kdebase-3.4.X.tar.bz2

kdebindings-3.4.X.tar.bz2

kdeedu-3.4.X.tar.bz2

kdegames-3.4.X.tar.bz2
```

```
kdegraphics-3.4.X.tar.bz2
kdelibs-3.4.X.tar.bz2
kdemultimedia-3.4.X.tar.bz2
kdenetwork-3.4.X.tar.bz2
kdepim-3.4.X.tar.bz2
kdesdk-3.4.X.tar.bz2
kdetoys-3.4.X.tar.bz2
kdeutils-3.4.X.tar.bz2
kdeWebdev-3.4.X.tar.bz2
```

Кроме того, в KDE входят пакеты arts и kdevelop, имеющие свою нумерацию версий, kde-i18n, обеспечивающую интернационализацию среды, а также пакеты локализации для отдельных языков (вида kde-i18n-язык, например, kde-i18n-ru), в числе коих — почти все мыслимые (хотя локализация для них выполнена с очень разной полнотой и качеством).

Далеко не все перечисленное обязательно к установке. Правда, во многих пакетных дистрибутивах Linux все компоненты KDE связаны кросс-пакетными зависимостями (включающими все необязательные), и их действительно приходится устанавливать все. Но во FreeBSD, SB Linux и тем более при ручной сборке можно обойтись только на самом деле нужными.

Первый и бесспорный кандидат к отчислению из списка — пакет kde-i18n (а это уже более 100 Мбайт): очевидно, что никому не нужна поддержка доброй полусотни языков одновременно. Достаточно отдельного пакета для своего, родного (в наших условиях это kde-i18n-ru), в крайнем случае — еще пары-тройки дружественных.

Далее, ясно, что пакет kdevelop только этим самым разработчикам и необходим, большинство же пользователей без него вполне могут обойтись. Ну, а решение об установке прочих пакетов пользователь должен принять самостоятельно. Для этого я вкратце опишу назначение каждого.

Для начала — о пакетах, обязательных к установке в любом случае. Как можно догадаться, их два: kdelibs и kdebase. Первый — набор специфичных для KDE библиотечных функций, дополняющих базовую библиотеку Qt. Второй же включает основные компоненты KDE: собственно оконный менеджер KWM и его аксессуары, минимальный набор тем и основные приложения — файловый менеджер, он же браузер, konqueror, набор текстовых редакторов (kwrite, kedit, kate), эмулятор терминала konsole и еще некоторые (не столь уж многочисленные).

Как ни странно, обязательным оказывается также пакет arts — собственная звуковая система KDE, используемая для воспроизведения системных звуков даже в том случае, если мультимедийные компоненты KDE не устанавливаются.

Все прочие компоненты сугубо опциональны, поэтому охарактеризую те, что использую сам и в порядке важности для меня лично:

- kdenetwork — сетевые компоненты, среди которых прекрасное средство дозвона kppp и kget (FTP-клиент, сам по себе ничем не выдающийся в ряду своих аналогов, но в интеграции с файловым менеджером/браузером konqueror весьма способствующий облегчению жизни);
- kdepim — пакет персональных помощников, в который не вполне, на мой взгляд, логично, включены почтовый клиент kmail и дополняющая его адресная книга;
- kdeWebdev — пакет разработки Web-материалов, основу которого составляет превосходный редактор HTML-кода Quanta Plus, гармонично дополняемый вспомогательными средствами, такими как средство проверки целостности ссылок (klinkstatus) и создания ссылочных карт на изображениях (kimagemapeditor);
- kdegraphics — пакет включает пару просмотрщикаов графических файлов, очень удобную программу съемки скриншотов, простенький, но неплохой редактор графики kolourpaint;
- kdemultimedia — в составе пакета очень приличный медиаплеер noatun, микшер, звуковой рекордер и т. д.;
- kdeutils — в его состав, помимо всякой бижутерии, входит калькулятор, штука нужная;
- kdeadmin — пакет утилит системного администрирования, требующих, за редким исключением, полномочий суперпользователя; я им не пользуюсь, хотя почему-то обычно устанавливаю.

Еще два пакета я также ставлю всегда: это kdeaddons (включает, помимо прочего, plug-in для редактора kate, konqueror и пр.) и kdeartwork, в который входят дополнительные украшательства рабочего стола (обои, иконы и т. д.), жизнь без которых была бы скучна. А еще kdegames, в нем можно найти несколько пасьянсов и пару вариаций на тему бессмертного тетриса, а больше ни в какие игры я не играю.

Осталось упомянуть пакеты, которые я никогда не ставлю и потому о содержании их имею смутное представление: kdeedu — как можно догадаться, нечто имеющее отношение к образованию (чего или кого — не знаю), kdetoys, kdeaccessibility — так называемые спецвозможности, ну и kdesdk — набор скриптов и утилит для разработчика.

Установка

В большинстве случаев пользователю не приходится как-то особенно устанавливать KDE, с этой задачей справляется либо инсталлятор его дистрибу-

тива, либо штатная система управления пакетами. Некоторая проблема возникает только в том случае, если предлагаемая схема установки почему-либо не устраивает, например, вследствие заведомой избыточности: я уже отмечал, что ряд пакетов из набора KDE практического смысла для большинства пользователей не имеет. И тут может оказаться целесообразным прибегнуть к ручной сборке требуемых компонентов.

Сам по себе процесс сборки KDE также особых сложностей не представляет, нужно только выдерживать последовательность сборки пакетов: сначала собирается библиотека Qt, потом звуковая система arts, потом — библиотека kdelibs, а за ней — kdebase. Прочие пакеты, при необходимости, собираются после этих остальных — и более-менее в любом порядке.

Для сборки библиотеки Qt тарбалл ее исходников распаковывается в тот каталог, в котором мы хотим ее видеть в дальнейшем, по умолчанию это /usr/local, но последнее время в Linux-дистрибутивах Qt часто помещается в каталог /opt. В любом случае в результате образуется подкаталог вида qt-version. Его надлежит переименовать:

```
$ mv qt-version qt
```

А в профильный файл командной оболочки root — внести коррективы значений переменных описания путей (подробно это описано в сопроводительной документации). Для sh-совместимого семейства это выглядит примерно так:

```
QTDIR=/usr/local/qt
PATH=$QTDIR/bin:$PATH
MANPATH=$QTDIR/man:$MANPATH
LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
export QTDIR PATH MANPATH LD_LIBRARY_PATH
```

А в C-shell (и `tcsh`) аналогичный результат достигается такими строками:

```
setenv QTDIR /usr/local/qt
setenv PATH $QTDIR/bin:$PATH
setenv MANPATH $QTDIR/doc/man:$MANPATH
setenv LD_LIBRARY_PATH $QTDIR/lib:$LD_LIBRARY_PATH
```

Эти переменные потребуются уже на стадии конфигурирования, поэтому необходимо либо авторизоваться заново, либо перечитать профильный файл предусмотренным образом (например, через source /etc/profile). Затем дается пара команд

```
$ ./configure && make
```

и Qt можно считать установленной. По умолчанию бинарники библиотеки собираются в тот же каталог, что и исходники, так что необходимости в команде `make install` нет.

Но при желании с помощью обычной опции

```
--prefix
```

при конфигурировании можно заказать установку в какой-либо иной каталог.

Из остальных опций конфигурирования нелишней представляется минимум

```
-qt-gif
```

включающая поддержку GIF-графики (по умолчанию, как ни странно, это не подразумевается).

Интересно, что при сборке Qt игнорируются любые флаги `gcc`, кроме неких изначально прописанных в ее настроечных файлах, так что оптимизацией этой библиотеки можно не увлекаться. По умолчанию Qt собирается с уровнем оптимизации `-O2`, что вполне достаточно. Однако если покажется мало, придется лезть достаточно глубоко в конфиги, зависимые от операционной системы.

Сборка каждого из пакетов собственно KDE выполняется посредством трех традиционных сакральных действ

```
$ ./configure && make && make install
```

из которых особого внимания заслуживает первое. Особенность конфигурационных сценариев пакетов KDE в том, что они обычно исполняются до конца, даже в случае нарушения зависимостей. При этом четко различаются зависимости "жесткие", без которых сборка и функционирование пакета невозможны, и "мягкие" зависимости, добавляющие ему необязательных (но часто задействованных по умолчанию) функций.

Если с "жесткими" зависимостями все ясно (они подлежат непременному удовлетворению), то в отношении зависимостей "мягких" пользователю предоставляется выбор. Например, в числе "мягких" зависимостей пакетов KDE обнаруживаются система печати cups и система сканирования sane. Однако вполне возможно, что пользователь в этих функциях не нуждается (например, по причине отсутствия соответствующих агрегатов). И потому вполне может отключить их указанием должных опций конфигурационного сценария, а полный их список, как обычно, получается командой

```
$ ./configure --help
```

По умолчанию Qt и пакеты KDE устанавливаются в собственные ветки каталога /usr/local (/usr/local/qt и /usr/local/kde). Однако в последнее время во многих дистрибутивах Linux просматривается тенденция перемещать такие крупные программные комплексы в каталог /opt (и это приветствуется Стандартом иерархии файловой системы Linux, *см. главу 10*). Для следования этой тенденции при выполнении конфигурационного скрипта необходимо задать соответствующие опции:

```
$ ./configure --prefix=/opt --with-qt-dir=/opt/qt
```

После сборки Qt и kdelibs (и перед сборкой остальных пакетов KDE) необходимо сделать соответствующие библиотеки общедоступными для всех приложений. Для этого нужно вписать в файл /etc/ld.so.conf строки

```
/opt/qt/lib
/opt/kde/lib
```

и запустить команду

```
$ ldconfig
```

Теперь об оптимизации. KDE (как и Qt) написана на Си++ и, соответственно, для оптимизации требуется не флаг `CFLAGS`, а `CXXFLAGS`. Впрочем, их можно приравнять друг другу:

```
export CXXFLAGS="$CFLAGS"
```

Ну, а о возможных значениях `CFLAGS` вдоволь говорится в другом месте (*см. главу 14*).

Собственные средства настройки

Настройка KDE — процесс многогранный. С одной стороны, пользователь может использовать этот десктоп в виде по умолчанию, с другой — затратить массу времени на доведение его до немыслимого совершенства. Мы же попробуем избрать некоторый промежуточный путь.

В предыдущих разделах этой статьи уже говорилось о том, что почти все параметры внешнего вида и поведения KDE можно настроить через KCC — Центр управления KDE. И это штатный способ конфигурирования: теоретически рассуждая, у пользователя не должно возникать потребности обращаться к прямому редактированию конфигов (за редкими исключениями, о которых речь пойдет далее). Так что с общего обзора KCC мы и начнем.

Вызов KCC осуществляется из стартового K-меню — через пункт **Центр управления**, после чего на экране возникает картина, подобная приведенной на рис. 17.1.

Можно видеть, что окно Центра управления состоит из двух фреймов, левый заключает в себе меню, правый же — расшифровку его основных пунктов и настроечные панели к отдельным из них.

Пункты меню отсортированы по алфавиту и в русскоязычном варианте имеют следующий порядок:

- Безопасность и конфиденциальность;
- Внешний вид и темы;
- Звук и мультимедиа;
- Компоненты KDE;

- Периферия;
- Рабочий стол;
- Региональные и специальные возможности;
- Сеть и Интернет;
- Системное администрирование;
- Управление питанием.

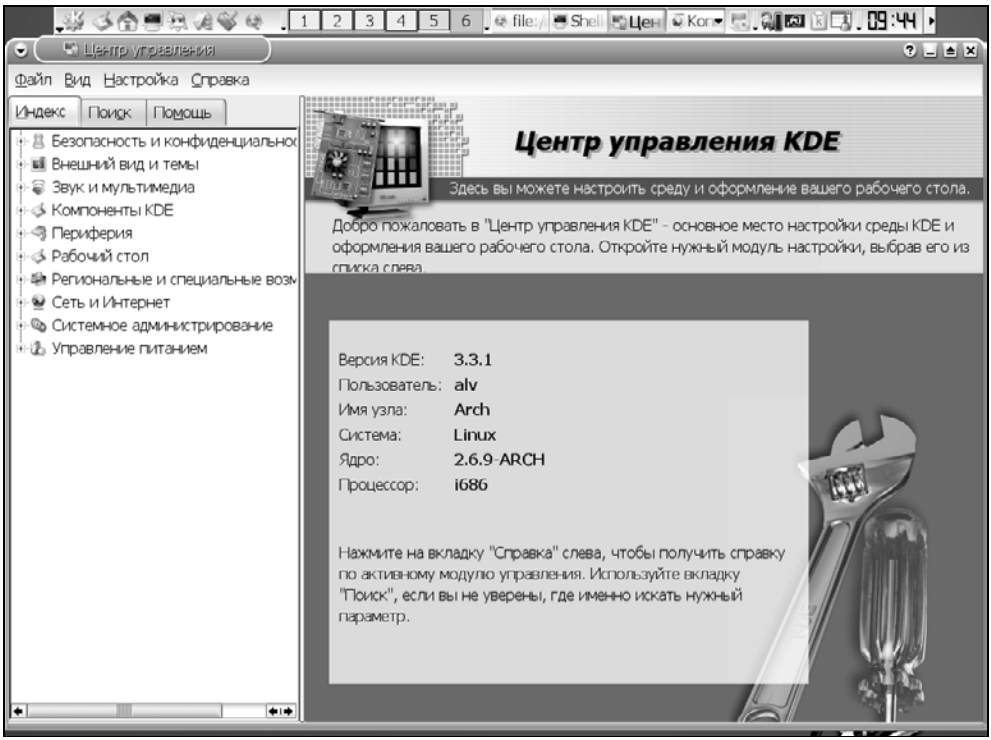


Рис. 17.1. Центр управления KDE — общий вид после запуска

Рассмотрим все эти пункты последовательно.

Безопасность и конфиденциальность

Смысл компонентов этого пункта вполне ясен из вводных комментариев к ним (рис. 17.2): это всякого рода очистка кэшей, шифрование, изменение некоторых параметров пользовательского акаунта. Несколько слов стоит сказать только о так называемом "бумажнике" (kwallet).

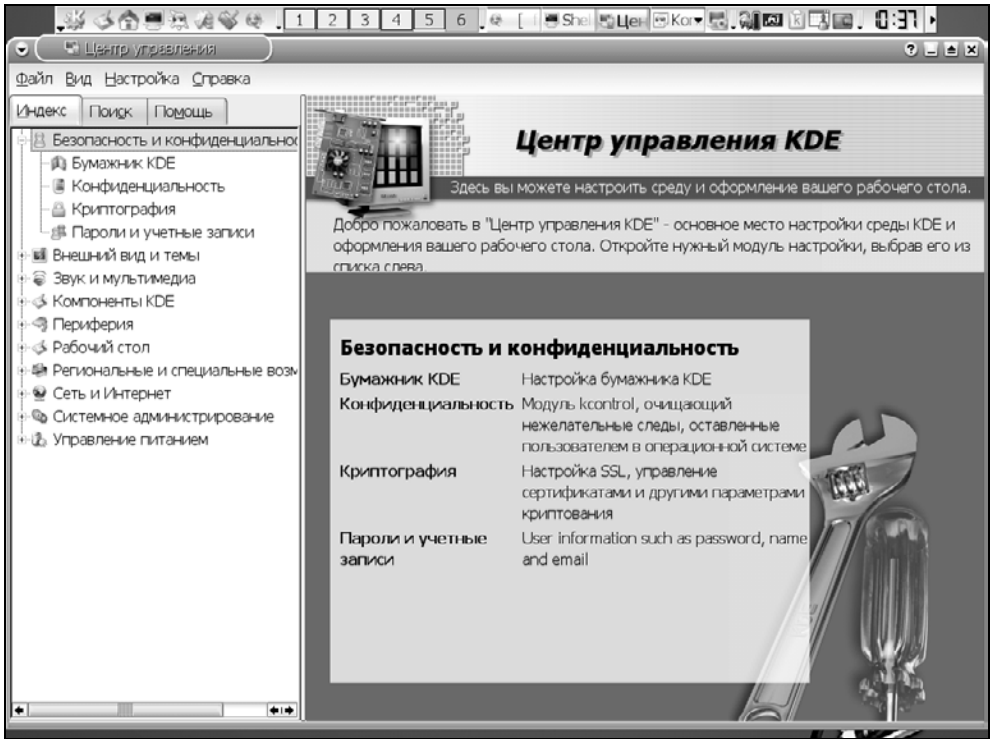


Рис. 17.2. Безопасность и конфиденциальность

Программа kwallet (это оригинальное ее название) представляет собой базу данных для хранения всяческих пользовательских паролей, как локальных, так и удаленных (например, для доступа к FTP-серверам, регистрации на форумах, и т. д.), избавляя от необходимости запоминать их. С доступом по отдельному паролю, разумеется, только его и следует помнить. Не знаю уж, насколько эта система действенна для по-настоящему конфиденциальных паролей (и жестких условий обеспечения секретности), но вот для хранения регистрационных данных для всякого рода онлайновых сервисов подходит вполне.

Внешний вид и темы

Это очень обширный пункт меню (рис. 17.3), через который можно настроить практически все визуальные элементы интерфейса KDE, как-то:

- **Декорации окон**;
- **Запуск приложений**;
- **Значки**;

- **Менеджер тем**;
- **Стиль**;
- **Фон**;
- **Хранитель экрана**;
- **Цвета**;
- **Шрифты**;
- **Экран-заставка**.

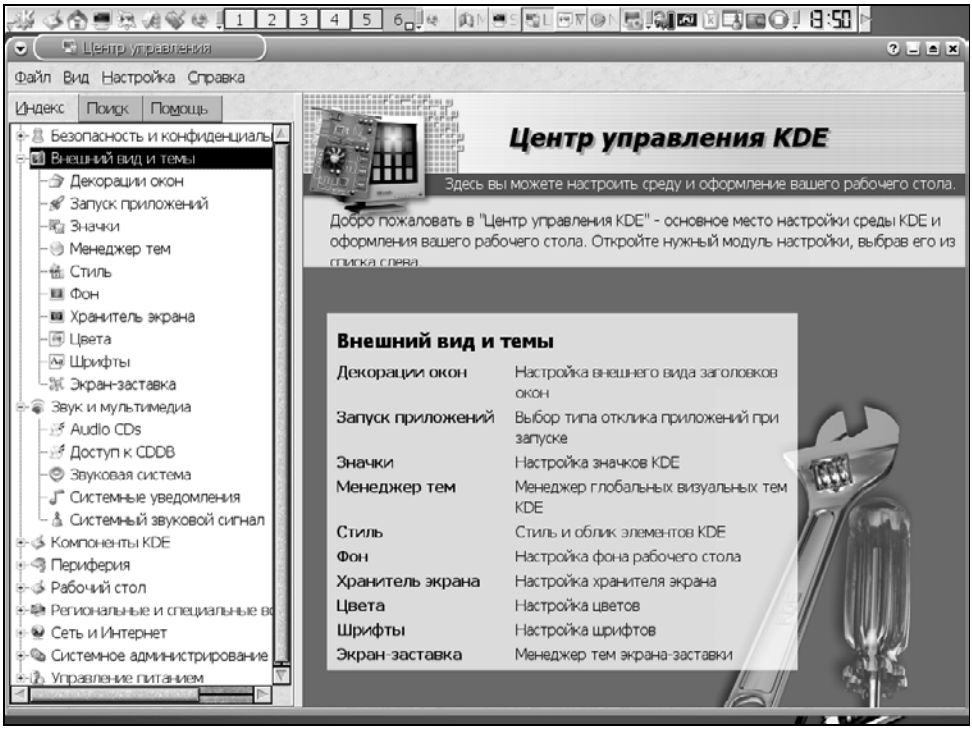


Рис. 17.3. Меню настройки внешнего вида KDE

Назначение элементов более или менее понятно из их имен. К сожалению, иерархия их несколько запутана, что усугубляется русским переводом соответствующих подпунктов меню. Поэтому здесь можно говорить или очень много, или ничего (в расчете, что пользователь со временем сам во всем разберется). Для начала замечу только, что здесь определяются такие визуальные свойства KDE, как пиктограммы управления окнами, цвет и фоновый рисунок рабочего стола, гарнитуры, размер и цвет глобально используемых шрифтов и многое другое.

Главным подпунктом этого меню является **Менеджер тем**: именно выбор темы определяет характер всех интерфейсных элементов. Большинство из них могут после этого быть настроены независимо — с помощью остальных пунктов. Правда, никаких тем (кроме той, что подразумевается по умолчанию), в комплекте KDE и нет. Их нужно отыскать (например, на упоминавшемся ранее сайте **http://kde-look.org**) и установить самостоятельно. Далее эту тему можно распространить либо на все интерфейсные элементы, либо лишь на некоторые (стиль, например, декорации окон или пиктограммы). В общем, все это легче понять, попробовав что-то изменить, чем внятно описать, как и что именно следует менять (тем более, что у каждого будут собственные предпочтения).

Звук и мультимедиа

В этом пункте (рис. 17.4) можно настроить параметры воспроизведения аудиокомпактов и доступа к базе данных, изменить характер системных сообщений в ответ на всякого рода события и, конечно же, сконфигурировать собственно систему воспроизведения звука. Почти все это — интуитивно понятно, только последний пункт заслуживает пары слов.

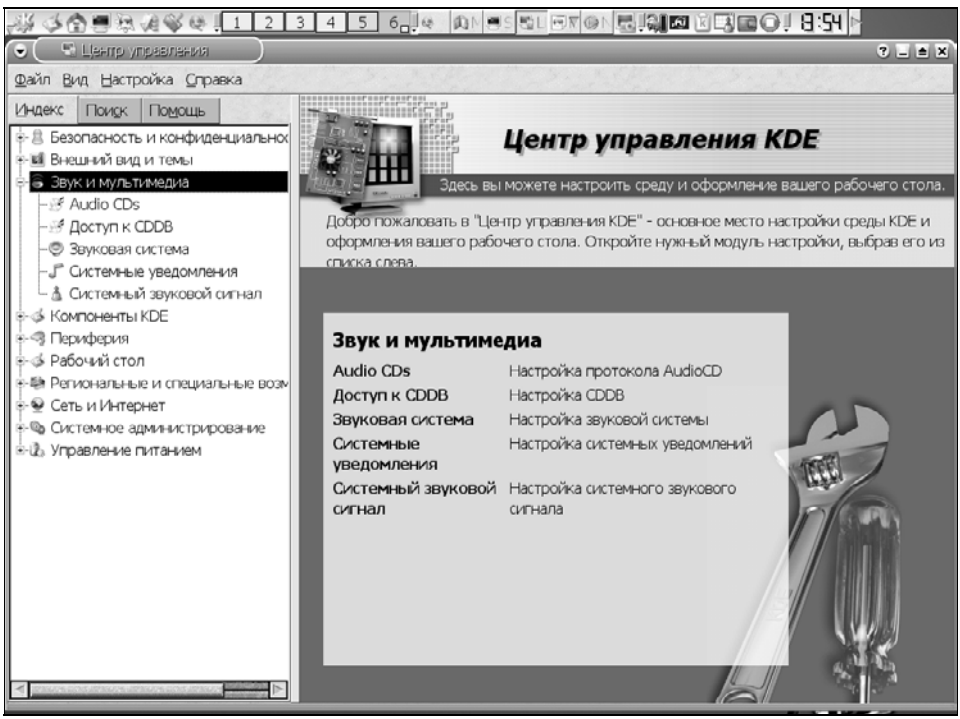


Рис. 17.4. Настройка звука

Конечно, здесь можно просто включить/выключить воспроизведение звука (по умолчанию включено) и установить приоритет воспроизведения. Но главное — это выбор звукового устройства (рис. 17.5).

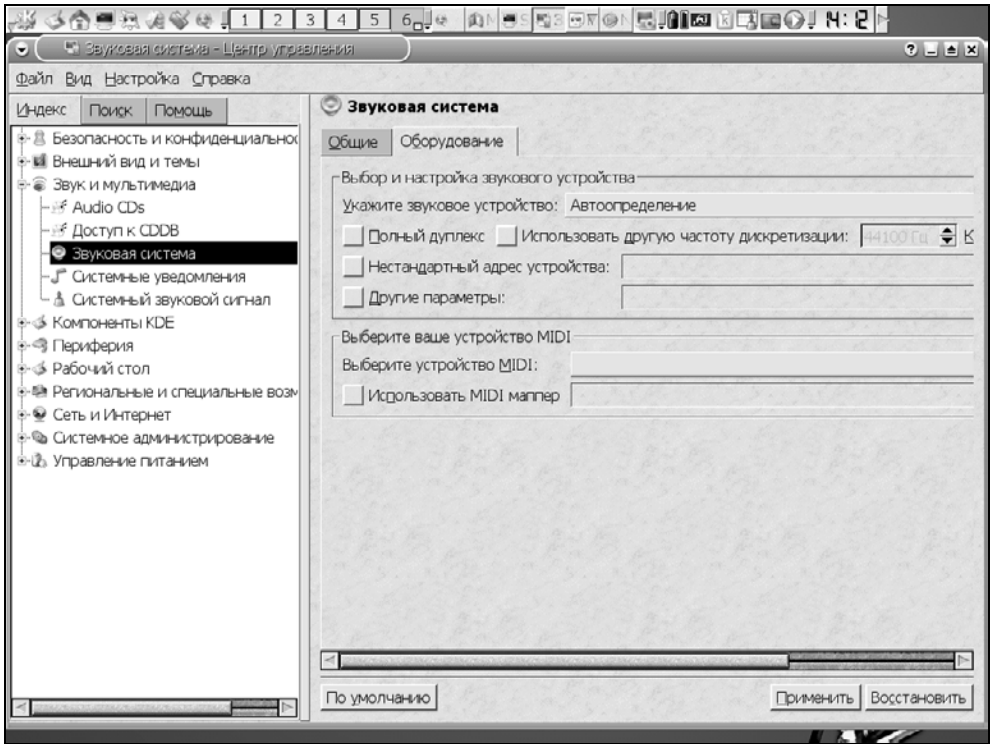


Рис. 17.5. Выбор звукового устройства

По умолчанию звуковое устройство (точнее, драйвер) определяется автоматически. И, скажем, во FreeBSD это не составляет никаких проблем. Однако в современных ядрах Linux при использовании звуковой системы ALSA автоопределение может привести к конфликтам с собственной звуковой системой KDE (пакетом arts), так что, возможно, ALSA должна быть выбрана вручную из выпадающего списка.

Компоненты KDE

- Настройка компонентов KDE весьма разнообразна по своему смыслу (рис. 17.6). **Быстродействие KDE** — режим кэширования (в соответст-

вующей панели именуется минимизацией использования памяти, при достаточном ее объеме это можно отключить).

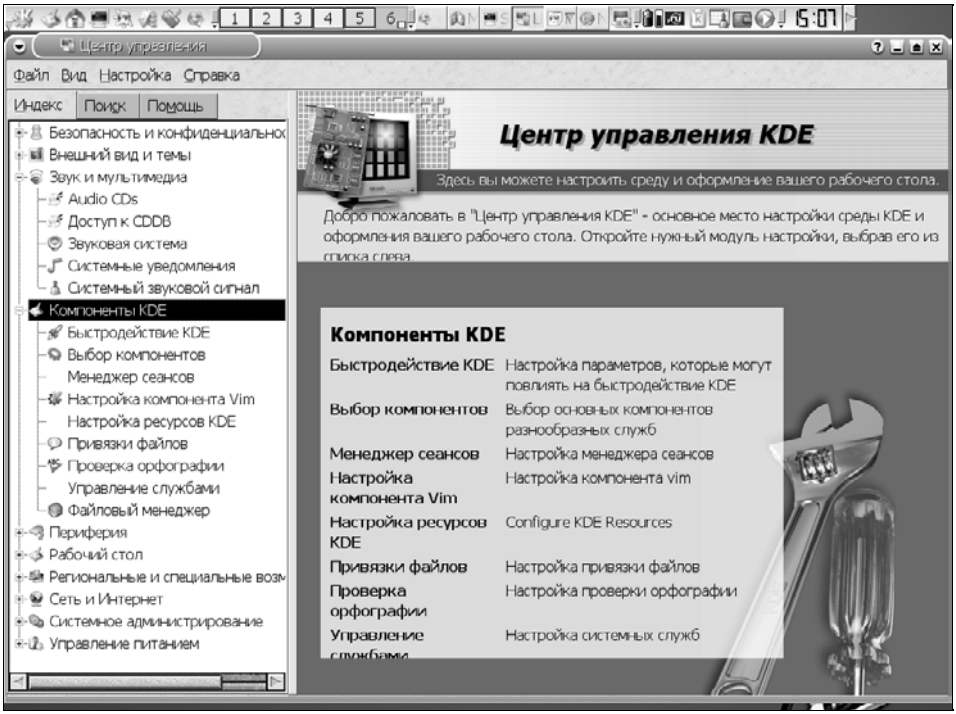


Рис. 17.6. Меню компонентов KDE

- **Выбор компонентов** — подразумевает смену браузера, текстового редактора, клиента электронной почты, терминала на другие программы, хотя и не любые. Правда, это именно тот пункт, в который вмешиваться я бы не стал: все используемые тут по умолчанию программы принадлежат к числу лучших в своем классе.
- **Менеджер сеансов** — устанавливает, нужно ли выводить предупреждение при выходе из сеанса KDE, восстанавливать ли при следующем запуске текущее состояние (включая окна открытых приложений и загруженные в них файлы), и что, собственно, должно происходить по выходе из KDE (только ли завершение сеанса, перезагрузка системы или даже выключение машины).
- Рассмотрение пункта **Настройка компонента Vim** предоставляется в качестве самостоятельного упражнения тем из читателей, кто этот редактор использует (причем в виде gvim или kvim).

- **Настройка ресурсов KDE** — относится ко всякого рода календарным записям.
- **Привязки файлов** — устанавливает связь определенных типов (точнее, масок имен) с тем или иным приложением, например, файлов с расширением txt — с текстовым редактором, а avi — с медиаплеером (рис. 17.7).

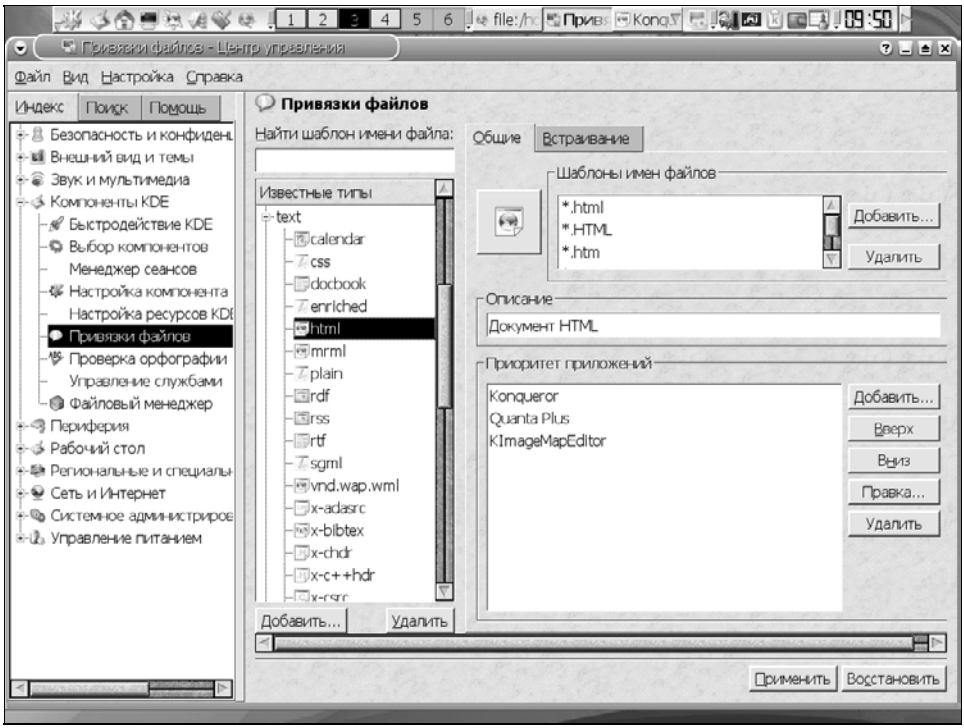


Рис. 17.7. Привязка файлов

Большинство таких привязок уже выставлены по умолчанию, необходимость в их изменении возникает, скорее всего, при установке новых приложений (например, mplayer в дополнение к штатным noatun и kaboodle). Одна и та же маска файла может быть связана с несколькими приложениями — например, для HTML-файлов это могут быть браузер konqueror и Web-редактор Quanta; в этом случае приоритет связи определяется порядком перечисления программ и может быть легко изменен.

- **Проверка орфографии** — устанавливается программа ispell или aspell, язык словаря (русский) и набор его символов (например, KOI8-R или CP1251). Очевидно, что соответствующая программа для спеллинга должна быть установлена в системе, сама по себе она в состав KDE не входит. Но раз

задействованная через Центр управления, она будет использоваться во всех приложениях KDE, в которых спеллинг имеет смысл (в текстовых редакторах, почтовом клиенте, HTML-редакторе).

- **Управление службами** — выводит панель со списком KDE-специфичных демонов (таких как упоминавшийся ранее kwallet) и их статусом. Однако как раз управлять-то большинством из них не получится...
- **Файловый менеджер** — позволяет определить некоторые свойства программы konqueror в этом качестве. Некоторые из них (например, гарнитура, размер и цвет шрифта) дублируются собственными настройками konqueror, о чем я подробно напишу в ближайшей интермедии. Иные же (как кэширование операций копирования и перемещения файлов) присутствуют, как будто бы, только здесь.

Периферия

В этом разделе можно посмотреть и изменить свойства таких устройств, как экран, мышь, клавиатура (рис. 17.8).

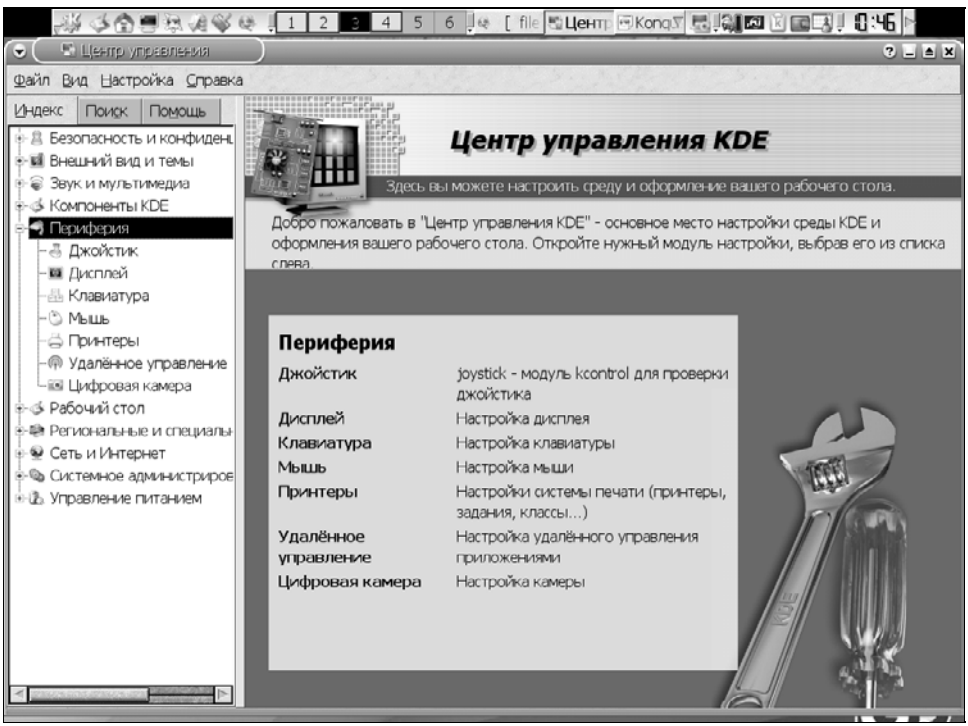


Рис. 17.8. Настройка периферийных устройств

Правда, изменению поддается не так уж много параметров. Для экрана это его разрешение, баланс цветов и включение/выключение энергосбережения.

Для клавиатуры здесь мы не найдем возможности смены раскладок (таковые имеют место быть совсем в другом месте), можно лишь включить задержки ее автоповтора и изменить статус клавиши <NumLock> при запуске KDE. Для мыши можно переопределить левую/правую кнопки, сменить одинарный щелчок для открытия файлов и каталогов на двойной (как это принято в Windows), изменить вид курсора и установить его акселерацию.

Рабочий стол

Этот раздел посвящен оформлению рабочего пространства в среде KDE как целостности (рис. 17.9): количества рабочих столов, вид и местоположение главной управляющей панели, представление стартового K-меню (отредактировать его состав можно также через этот пункт), привязку контекстных меню рабочего стола к кнопкам мыши и, наконец, правила поведения окон.

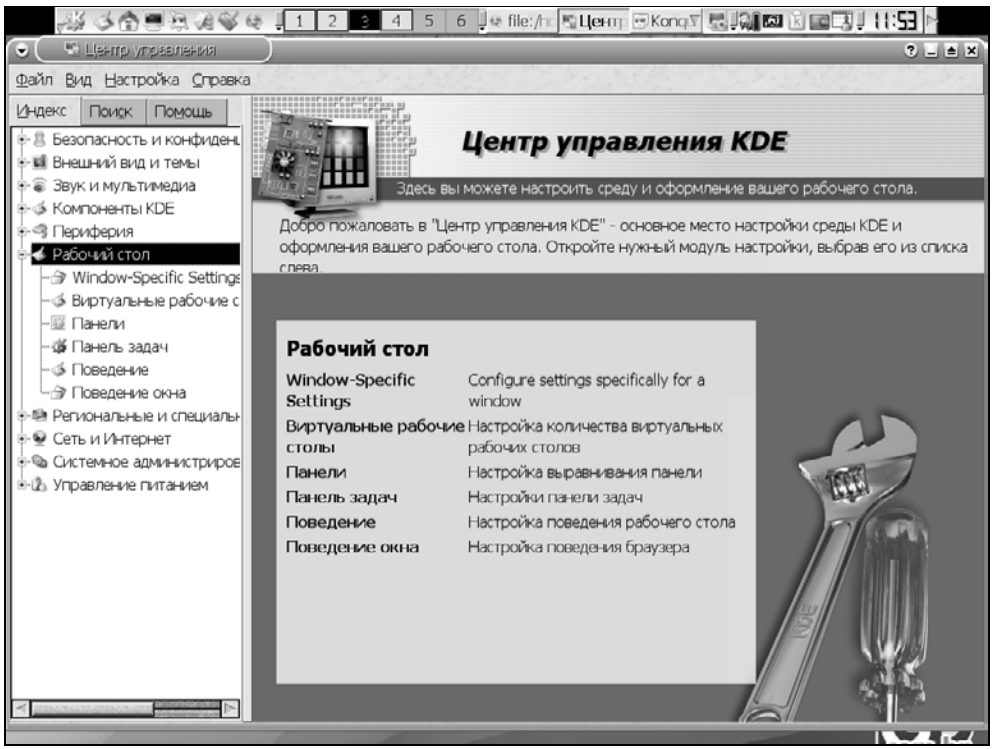


Рис. 17.9. Настройка оформления и поведения рабочего стола

Обязательно нужно сказать чуть подробнее о правилах "поднятия" и фокусировки окон. В Windows активизация (готовность реагировать на "мышиные" и клавиатурные события — выделение, ввод и т. д.) и "поднятие" окна (то есть вывода его на первый план) — понятия в большинстве случаев идентичные. В Иксах же окно может быть активным, даже будучи погребено под геологическими напластованиями других окон. Причем и поднятие, и фокусировка окна не обязательно требуют щелчка на нем мышью, при соответствующих настройках оконного менеджера достаточно навести на окно курсор мыши.

Оценить такую возможность может каждый, кому приходилось, например, редактировать находящийся в окне первого плана HTML-документ, одновременно перетаскивая в него копии ссылок из открытого в "фоновом" окне браузера. Впрочем, различение фокуса окна и его плана может доставить еще массу дополнительных удобств.

Так вот, в KDE по умолчанию принята схема обращения с окнами по типу Windows, то есть фокус и поднятие окна одновременно достигаются щелчком в любом его месте. Однако в пункте **Поведение окна** раздела **Рабочий стол** такое положение легко изменить: можно установить, чтобы фокус следовал за мышью, при этом окно может как оставаться на заднем плане, так и всплывать. Возможны и иные варианты настройки поведения окон, которые читателю предлагается изучить методом ползучего эмпиризма.

Региональные и специальные возможности

Этот раздел также очень важен и содержит такие пункты (рис. 17.10):

- **KHotKeys** — модуль для настройки клавиатурных комбинаций ("горячих клавиш"), служащих для выполнения сложных команд и вызова приложений;
- **Привязка клавиш** — определение "горячих клавиш" для выполнения простых повседневных действий, таких как переключение рабочих столов, навигация по открытым окнам и т. д.;
- **Раскладка клавиатуры** — позволяет менять раскладку клавиатуры и подключать еще до двух дополнительных;
- **Специальные возможности** — предназначен для настройки звуковых сигналов, а также использования "залипающих" и "замедленных" клавиш;
- **Страна/область и язык** — установка локально-зависимых параметров.

В комментариях тут нуждаются три пункта. Во-первых, привязка клавиш (рис. 17.11). Здесь для начала можно выбрать привычную схему "горячих клавиш" — в стиле Windows, например, Mac OS, UNIX и т. д. Например, я всегда беру за основу очень удобную схему WindowMaker, при которой переключение между виртуальными десктопами осуществляется комбинацией клавиш <Alt> и номера соответствующего рабочего стола. Затем в рамках

выбранной схемы можно скорректировать привычные клавиатурные комбинации для обыденных действий. Для этого достаточно перейти на закладку **Последовательности привязок**, выбрать нужное действие из списка, отметить переключатель **По выбору** и нажать желательную клавишу и комбинацию.

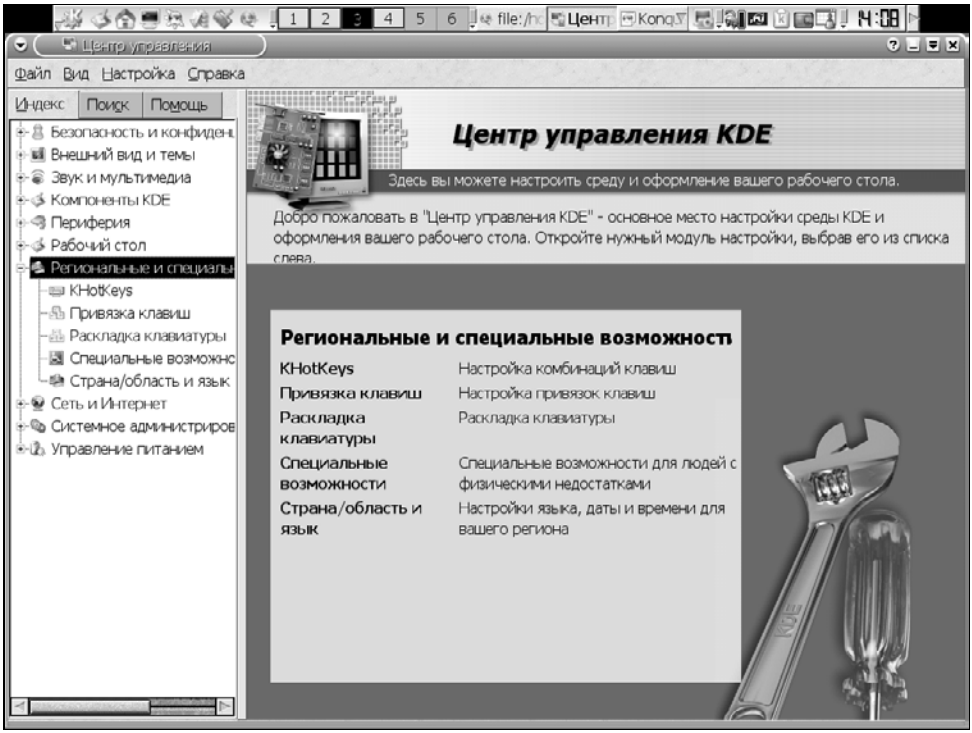


Рис. 17.10. Региональные и специальные возможности

Здесь следует учесть, что большинство удобных клавишных комбинаций уже определены по умолчанию в любой из предлагаемых на выбор схем. В прежних версиях KDE для того, чтобы переопределить уже использованную комбинацию, нужно было сначала "отвязать" ее от старого действия: выбрать его и отметить переключатель **Нет**. Однако, начиная с версии 3.3.1, в этом больше нет необходимости, при назначении "занятой" комбинации сразу предлагается отменить ее прежнюю привязку.

Во-вторых, раскладка клавиатуры. Эта тема будет предметом специального разговора. А пока отмечу, что модуль управления клавиатурой KDE — kxkb — полностью аннулирует все общие настройки клавиатуры в файле (/etc/X11/XFree86.conf или /etc/X11/xorg.conf), позволяя в принципе вообще обходиться без таких настроек.

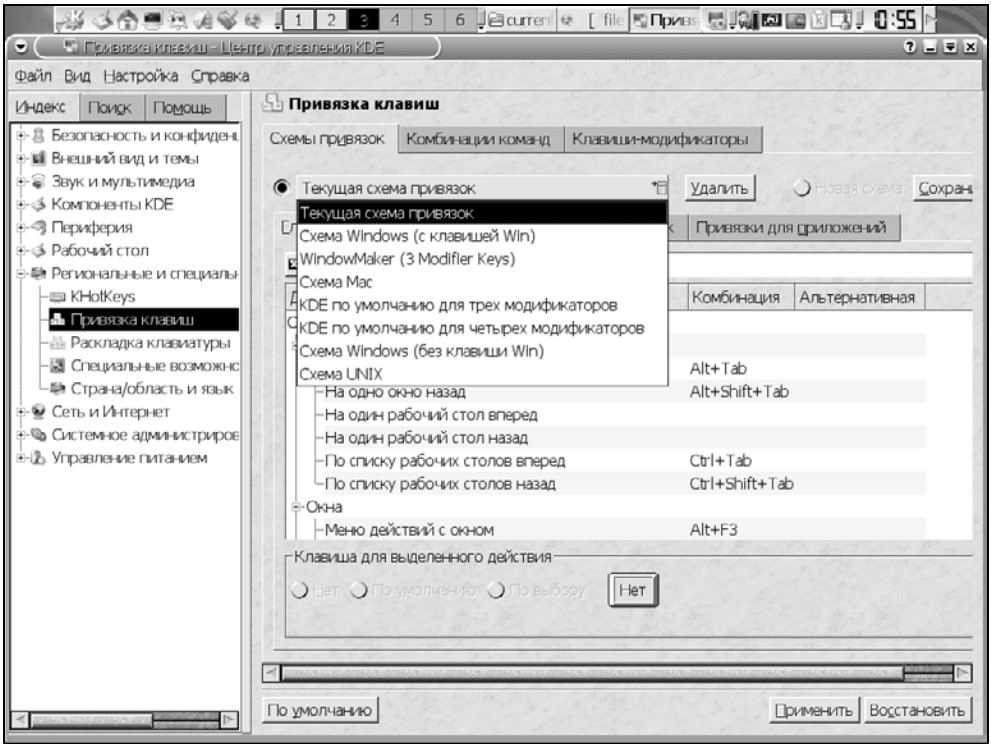


Рис. 17.11. Привязки клавиш

Внимание!

Лучше не пытаться пока переопределять Иксовые клавиатурные раскладки, иначе очень легко остаться без русских клавиш вообще. Это, конечно, в дальнейшем поправимо, но прочтение следующего раздела избавит от лишних телодвижений.

И в-третьих — страна, область и язык. Кроме собственно страны (например, России) и языка (скажем, русского, используемого для меню, вывода сообщений и прочего), здесь же определяются денежная единица, десятичные разделители, формат даты и времени, плюс кое-какие дополнительные параметры: стандартный формат бумаги (A4 или Letter), система единиц измерения (метрическая или английская), число десятичных знаков после запятой.

Все локальные параметры настраиваются независимо друг от друга. То есть можно определить страну как **Россия**, однако, при стойком отвращении к русскоязычным меню, языком по умолчанию назначить английский (вводу/выводу русских букв это не препятствует), в качестве десятичного раздели-

теля указать точку вместо запятой (это требуется некоторым счетным программам) и т. д.

Локально-зависимые параметры KDE далеко перекрывают переменные, описываемые в рамках системной локали. Однако (по крайней мере, для русского языка) они отнюдь не избавляют от необходимости корректного определения системной локали в обычном пользовательском окружении (через профильные файлы данного акаунта или, как это принято во FreeBSD, через определение класса пользователя). В противном случае возможны всякие неожиданности, вплоть до исчезновения символов кириллицы в терминале `console` и даже невозможности клавиатурного ввода при переключении на русскую раскладку.

Сеть и Интернет

На описании этого раздела я останавливаться не буду — представление о нем можно получить из входящих в него пунктов (рис. 17.12).

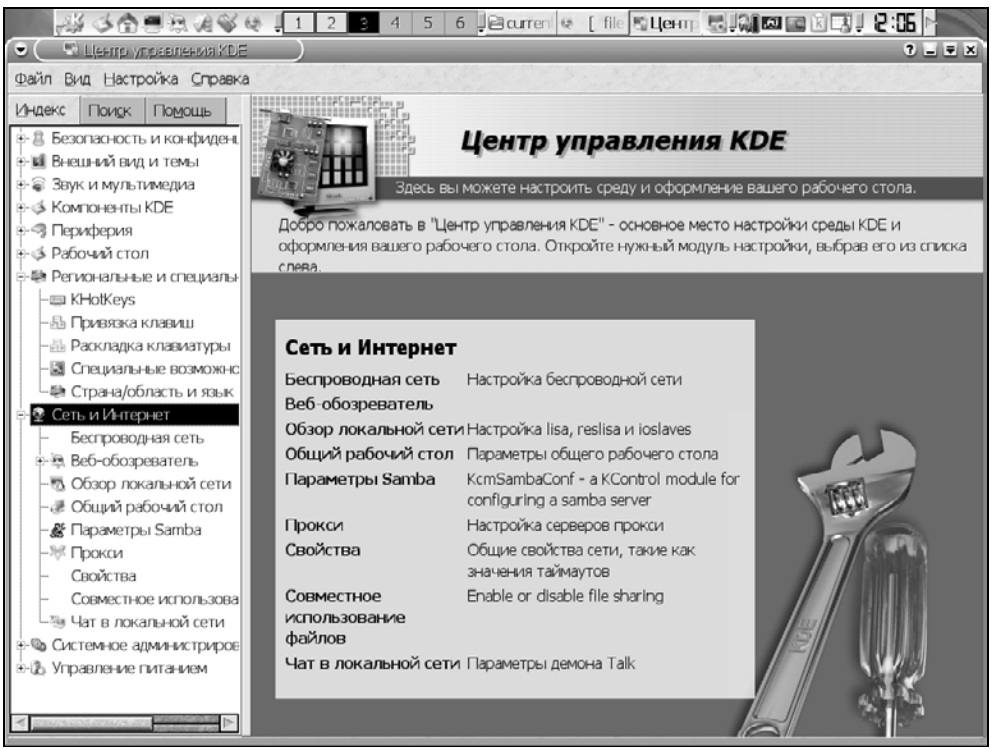


Рис. 17.12. Настройка сети и подключения к Интернету

Системное администрирование

Настройка любого параметра из всех предыдущих разделов Центра управления могла быть выполнена от имени обычного пользователя. В этом же разделе некоторые действия потребуют прав администратора. Впрочем, об этом выводится соответствующее предупреждение, а затем и предложение ввести суперпользовательский пароль. А сами действия (рис. 17.13) или вполне тривиальны (установка даты/времени, настройка менеджера входа в систему), или очень специальны (относясь к некоторому специфическому "железу"), или обычно не востребованы (конфигурирование ядра Linux, например, особенно если KDE работает поверх FreeBSD).

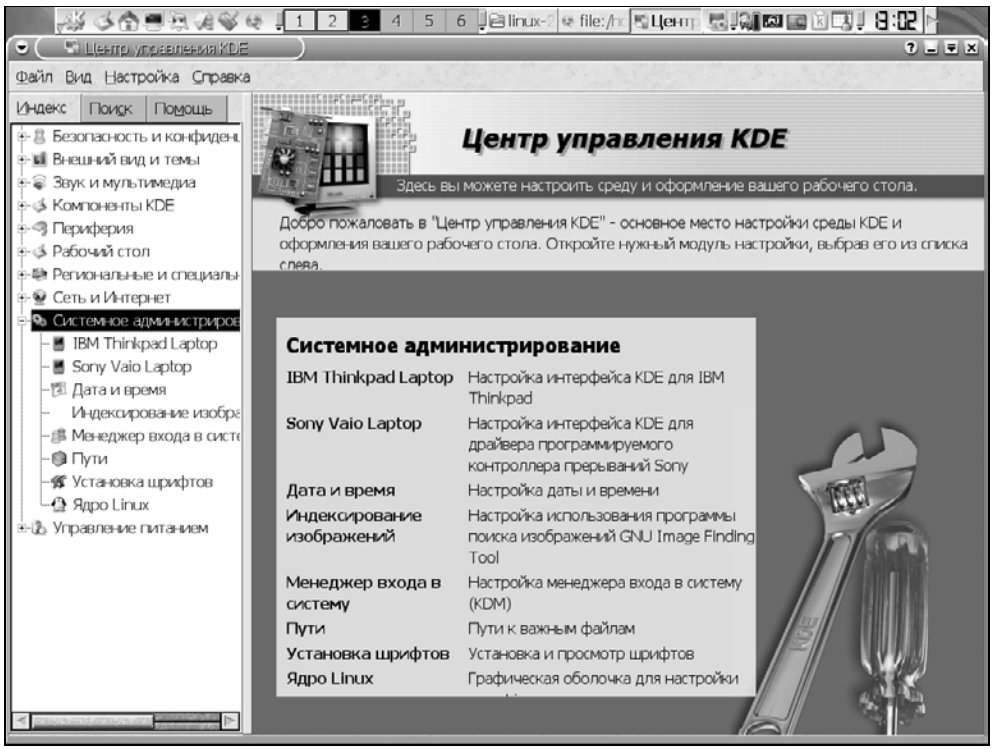


Рис. 17.13. Системное администрирование

Здесь, однако, хотелось бы мне заострить внимание на пункте **Установка шрифтов**. Конечно, обычно это проделывается для Иксов в целом — записью шрифтовых файлов в подкаталоги каталога /usr/X11R6/lib/X11/fonts и соответствующих путей — в файл /etc/X11/XF86Config(xorg.conf). Это, одна-

ко, требует прав администратора. Система же управления шрифтами KDE (kfontinst на рис. 17.14) позволяет обходиться без типичных шрифтовых настроек и инсталлировать необходимые шрифты от имени обычного пользователя (и только для его личного употребления — установленные таким образом шрифты оказываются в каталоге $HOME/.fonts). Еще один шаг на пути полной автономии KDE от оконной системы X...

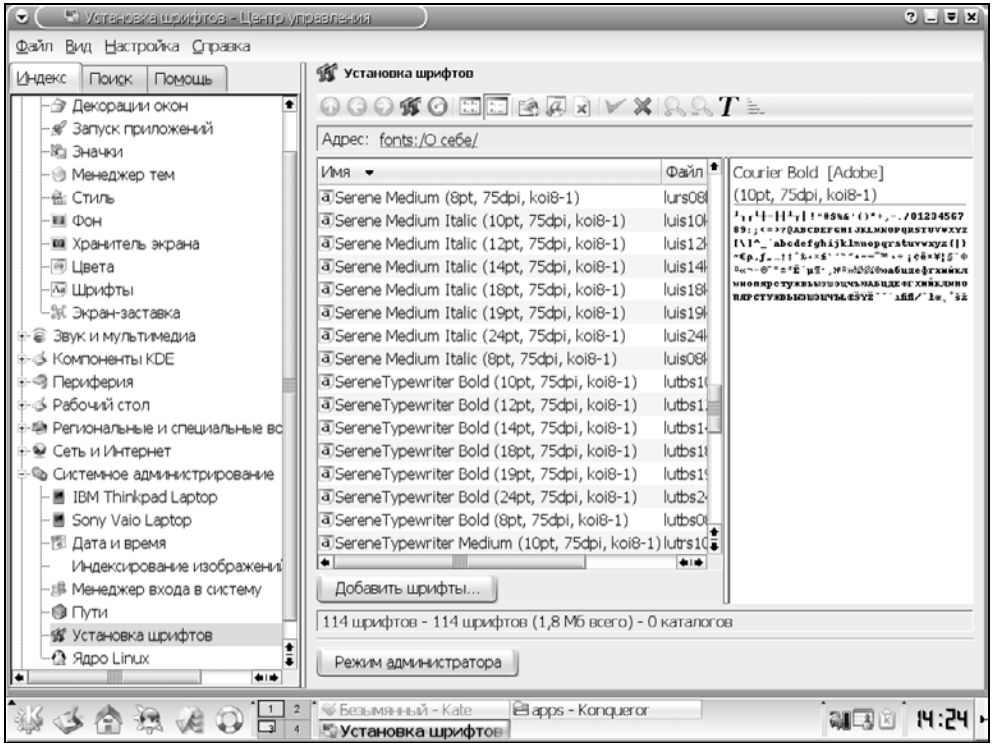


Рис. 17.14. KDE допускает установку шрифтов обычным пользователем

Лирическое отступление

Некоторые из моих знакомых время от времени пишут мне письма на русском языке, но латиницей. Это мотивируется тем, что, работая на UNIX-машинах, они не имеют административных прав и, соответственно, возможности установить русские шрифты. Так вот, KDE с помощью модуля kfontinst позволяет легко решить эту проблему.

Управление питанием

Этот раздел содержит единственный пункт **Аккумулятор ноутбука** и, соответственно, только для ноутбуков и предназначен. Здесь можно настроить довольно много вещей (рис. 17.15) — от вида пиктограмм-индикаторов зарядки аккумулятора до времени предупреждения о скорой разрядке и соответствующих им действий (исполнение команд, подача звукового сигнала).

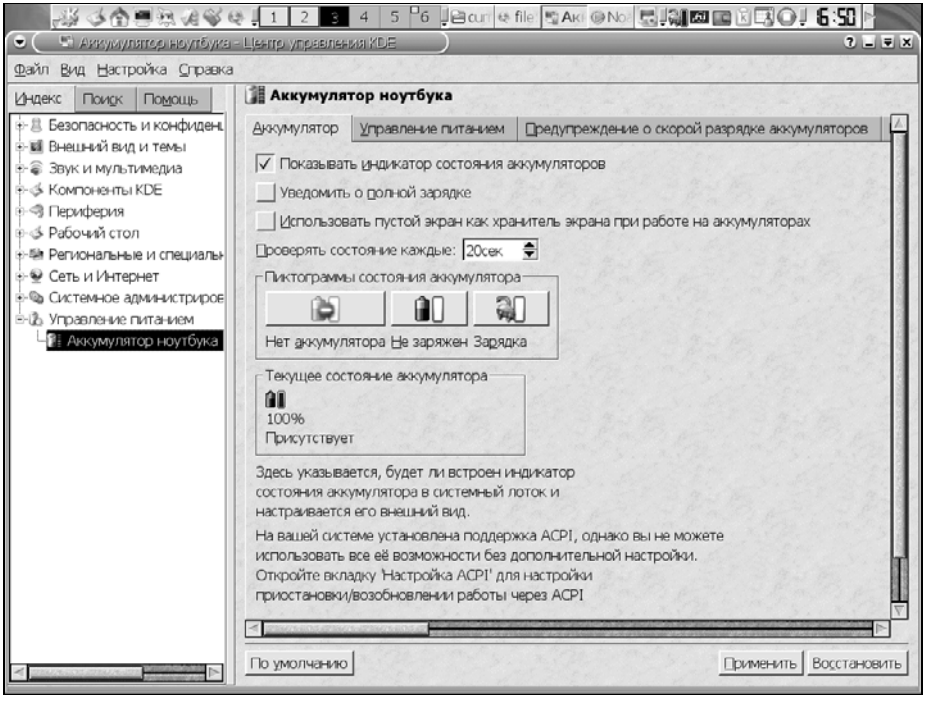


Рис. 17.15. Управление питанием

Теоретически тут присутствует и вкладка **Настройка ACPI**, однако мне ее стандартные опции оказались недоступными, а попытка включить дополнительные опции вызвала грозное предупреждение, коему я счел за благо внять.

Конфигурационные файлы

Интерактивные настройки через Центр управления KDE, естественно, находят свое отражение в соответствующих конфигурационных файлах. И настало время рассмотреть, хотя бы вкратце, их структуру.

Общесистемные конфиги KDE собраны в подкаталоге share корневого его каталога (обычно — /usr/local/kde или /opt/kde). Правда, как обычно редактировать их не рекомендуется, да они и недоступны для изменения обычным пользователем. Тем не менее, на всякий случай запомним это местоположение: кое-что из содержимого подкаталога ~/kde/share/apps нам со временем потребуется при настройке отдельных KDE-приложений.

Ну, а пользовательские настройки KDE, как всегда, определяются rc-файлами в домашнем каталоге каждого пользователя. Тут, в первую голову, следует упомянуть основной конфигурационный файл $HOME/.kderc. Правда, содержится в нем не так уж и много — имена шрифтов меню, главной панели, рабочего стола и тому подобных интерфейсных элементов, имя моноширинного шрифта, используемого в терминале konsole и терминальном окне konqueror (о последнем будет говориться в ближайшей интермедии). Здесь же можно видеть строки, описывающие цвета шрифтов, фона и переднего плана активных и неактивных окон — словом, кое-что из того, что настраивается через меню **Внешний вид и темы** Центра управления KDE.

Далее, некоторое отношение к конфигурации KDE имеет каталог $HOME/.fonts, именно в нем, как уже говорилось, помещаются шрифты, устанавливаемые через Центр управления обычным пользователем.

Главные же настройки KDE сосредоточены в каталоге $HOME/.kde. Он возникает автоматически при первом запуске этой среды пользователем и постепенно заполняется всякого рода следами жизнедеятельности — кэшами, сокетами, временными файлами. Собственно к конфигурации же KDE имеют отношение два подкаталога — $HOME/.kde/share/config и $HOME/.kde/share/apps. Первый, как легко догадаться, содержит собственно конфиги (rc-файлы) всех входящих в KDE приложений, второй же — их более специфические настройки.

В ближайших заметках представление об этих файлах понадобится нам для корректной настройки клавиатурных раскладок KDE и для индивидуального конфигурирования некоторых KDE-приложений.

Детали настройки клавиатуры

Настройка клавиатуры — благо и проклятие начинающего пользователя KDE. Благо — потому, что возможности этой среды по управлению клавиатурными раскладками существенно превосходят базовые функции, предоставляемые самой оконной системой Икс. Проклятием же это дело может показаться в том случае, если пользователь, впервые берущийся за KDE и выполнивший все формально предписанные действия, вдруг с удивлением обнаруживает, что он не только не получил тех самых дополнительных воз-

можностей, но и утратил способность переключаться с латиницы на кириллицу (или, что еще хуже, наоборот — с кириллицы на латиницу) вообще.

Тем не менее, все не так страшно, как может показаться на первый взгляд. И если ранее (например, в руководствах Алексея Новодворского к первым версиям Mandrake Russian Edition) резонные люди советовали вообще не трогать модуль настройки клавиатуры KDE, во избежание потери базовой русификации Иксов, то ныне им вполне можно пользоваться. Если это, конечно, нужно.

А нужно ли это народу? Как обычно, ответ будет вполне дипломатичный — кому нужно, а кому и нет. В обоснование чего вспомним, как достигается базовая русификация клавиатуры в Иксах, и что она дает.

Как явствует из *главы 16*, сами по себе Иксы (то есть X-сервер) обеспечивают пользователю подключение кириллической раскладки, и не более того. Причем раскладки — одной-единственной, примерно соответствующей фабрично русифицированным клавиатурам с DOS-маркировкой клавиш. Что не следует путать с DOS-кодировкой клавиатурного ввода — это название исторически закрепилось за некогда распространенными (а когда-то и единственно наличествующими) в продаже клавиатурами, у которых в кириллице самые востребованные знаки препинания (точка и запятая) располагались на верхнем регистре цифрового ряда клавиш, как на механических пишущих машинках отечественного производства...

Далее, Иксы, позволяя подключить кириллическую раскладку, парадоксальным способом не обеспечивают возможности переключения на нее (или, напротив, с нее на латиницу). Ибо эта самая кириллическая раскладка штатно такого переключателя не содержит.

Историческое отступление

Некогда русская раскладка переключатель имела, и привязан он был к клавише <CapsLock>. Как сказали бы в Одессе, и кому это мешало? Кому-то, видимо, мешало, потому что, начиная, если память не изменяет, с версии XFree86 3.3.4, штатный переключатель из русской раскладки выкинули. То есть пользоваться ею по прямому назначению стало невозможно, ведь трудно представить себе самого исконно-кондового патриота, не нуждающегося в латинице вообще. Да, если бы не модуль расширения Иксов, именуемый Xkb Extensions (или просто Xkb).

Его подключение (а при любом способе конфигурирования Иксов оно выполняется по умолчанию) обеспечивает такие дополнительные возможности, как:

- использование варианта раскладки (в частности, варианта winkeys, соответствующего маркировке ныне продающихся фабрично русифицированных клавиатур);

- выбор клавиши-переключателя (в отличие от Windows, выбор этот весьма обширен — допустимы фиксируемые модификаторы <CapsLock> и <ScrollLock>, почти любые сочетания клавиш <Alt>, <Ctrl> и <Shift> (да еще с различением правых и левых), а также так называемые Win-клавиши (полный список можно посмотреть в файле /usr/X11R6/lib/X11/xkb/rules/xfree86.lst);
- назначение индикатора альтернативной (в нашем случае — русской) раскладки, каковым может быть любой "огонек" на клавиатуре — <CapsLock>, <NumLock> или <ScrollLock>, вне зависимости от того, какая клавиша выступает переключателем;
- возможность подключения третьей раскладки клавиатуры;
- всякого рода переопределения клавиш-модификаторов, в том числе и Win-клавиш.

В итоге, казалось бы, непосредственно в Иксах можно получить все, что нужно для счастья русскому (пардон, русскоязычному) человеку. За двумя исключениями:

- переключение раскладок клавиатуры в Иксах, в отличие от Windows, имеет силу для всех приложений сразу; что не всегда может быть удобно — например, при одновременной работе в командной строке (требующей латиницы) и текстовом редакторе с русскоязычным документом;
- с помощью Xkb можно задействовать только три раскладки клавиатуры, и если по ходу дела в русско-латинский (или русско-английский) текст требуется вставлять еще и немецкие умляуты с французскими аксантами, а то и изобильную скандинавскую символику, их оказывается явно мало.

И вот тут-то и приходит на помощь интегрированный в KDE аналог Xkb, который, как нетрудно догадаться, носит имя kxkb. Так что если ни множество раскладок, ни привязка переключения клавиатуры к некоему приложению (или даже отдельному окну приложения) вам лично не нужны, дальше можете не читать, забыв об `kxkb` навсегда и ограничившись корректной настройкой Xkb.

Если же хоть одна из описанных возможностей кажется вам необходимой или полезной — приступим к настройке kxkb. Только следует учесть: дело это придется доводить до победного конца. Потому что включение этого модуля автоматически аннулирует все настройки, заданные в конфигурационном файле Иксов, и некорректное использование kxkb может, как уже говорилось, лишить возможности ввода русских букв вообще.

И еще: если необходимо использовать множество клавиатурных раскладок, дело не ограничится настройкой kxkb, а потребует еще и использования unicode-шрифтов и локали UTF-8. Что, впрочем, к теме настоящей заметки не относится.

И последнее: все описанное далее основано на личном опыте общения с последними версиями KDE — 3.3.1 и 3.3.2. В более ранних версиях все было чуть-чуть по-другому, и не исключено, что и в следующих версиях ситуация будет меняться. Поэтому к моему тексту нужно относиться творчески...

Действуем через KCC

Итак, мы впервые приступаем к настройке клавиатурных раскладок. Для этого запускаем KCC (Центр управления KDE) и следуем по пунктам меню: **Региональные и специальные возможности | Раскладки клавиатуры**. В правом фрейме окна KCC появляется панель с тремя вкладками: **Раскладка**, **Параметры переключения**, **Параметры XKB** (рис. 17.16), которые мы и рассмотрим последовательно.

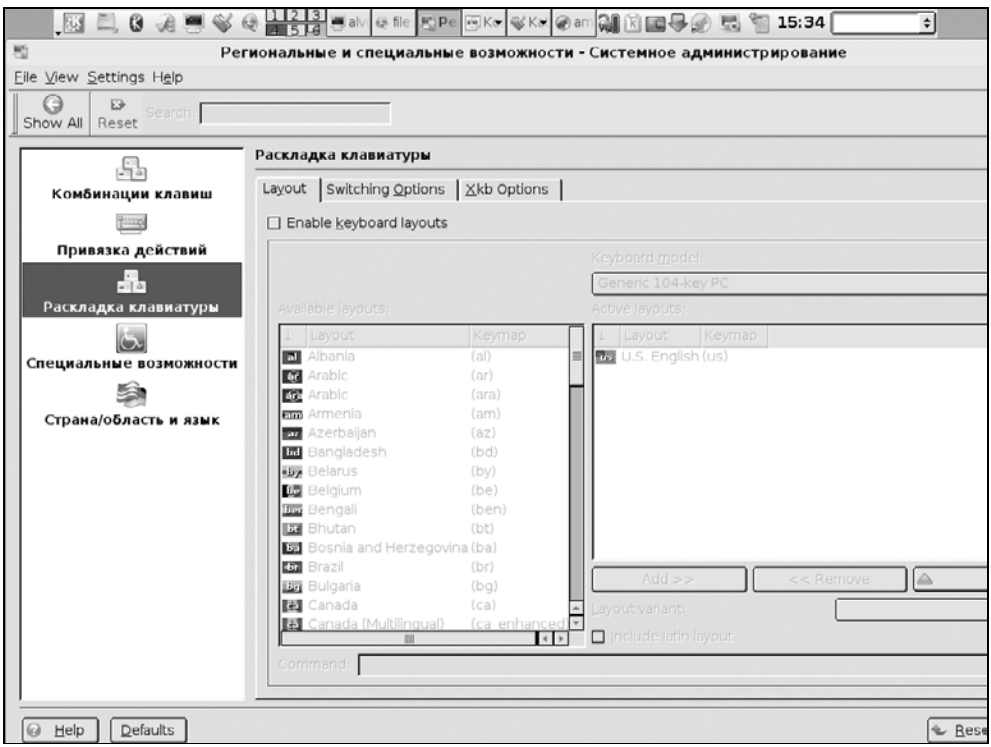


Рис. 17.16. Настройка раскладок клавиатуры через KCC

Как можно видеть из рис. 17.16, в момент первого запуска KCC раскладки не активизированы. Чтобы сделать их доступными для загрузки, нужно в первую очередь отметить переключатель **Включить раскладки клавиатуры**.

Что в данном случае следует понимать как подключение модуля kxkb. После чего из длинного списка слева можно выбрать требуемые раскладки, появляющиеся тем самым в списке задействованных справа (рис. 17.17).

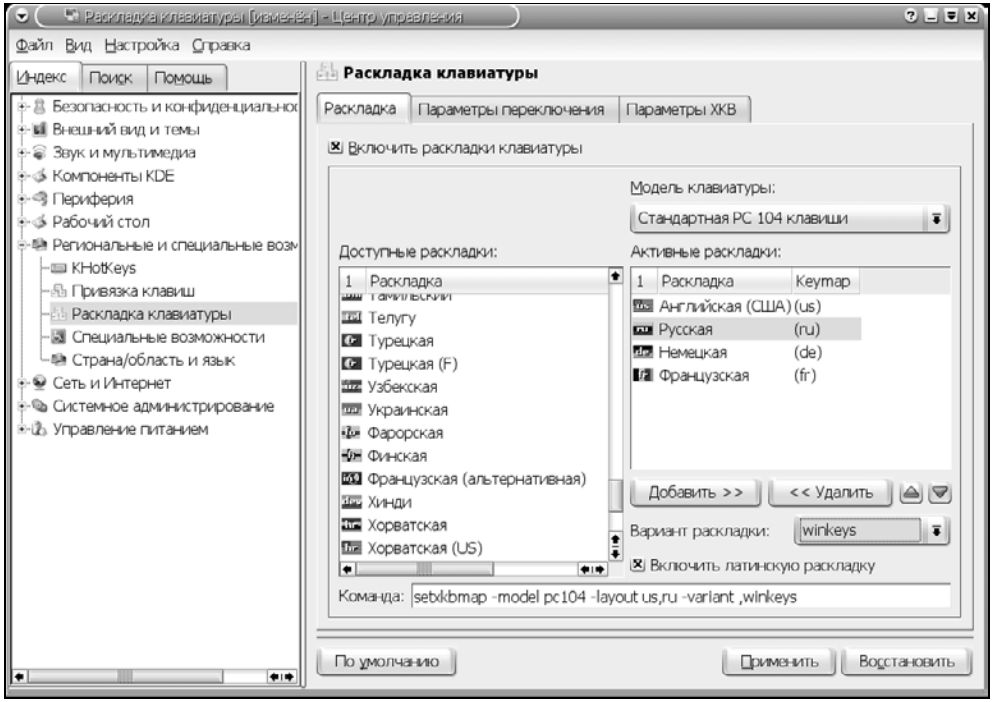


Рис. 17.17. Выбор необходимых раскладок клавиатуры

В настоящей заметке я ограничусь рассмотрением самого простого случая — включения латинской и русской раскладок. Вариант множественных клавиатурных раскладок остается в качестве самостоятельного упражнения для заинтересованных лиц.

Для начала из выпадающего меню **Модель клавиатуры** при необходимости вместо **Стандартная PC 104 клавиши** выбираем что-либо более подходящее имеющимся реалиям. Впрочем, в большинстве случаев обычных настольных клавиатур делать этого не следует: эмпирически замечено, что kxkb лучше всего работает именно со стандартной клавиатурой. И к перебору вариантов приходится прибегать только при наличии чего-либо уж очень экзотического (например, всяких своеобычных клавиатур).

Затем двойным щелчком мыши в левом списке добавляем к раскладке по умолчанию **Английская США (us)** нужную нам, в данном случае это будет

Русская (ru). А затем из нижнего выпадающего меню **Вариант раскладки** выбираем искомый — **winkeys**. При этом в "командной строке" в нижней части панели высвечиваются выбранные параметры. В нашем примере они будут выглядеть так:

```
setxkbmap -model pc104 -layout ru -variant winkeys
```

Далее переходим к вкладке **Параметры переключения** (рис. 17.18). Здесь, в соответствии с нормами современной русской (точнее, российской) орфографии, игнорируем переключатель, отвечающий за букву "ё" (хорошо это или плохо — другое дело, но в книгах и прочих печатных изданиях России принято именно так). Зато, при желании, ставим переключатель **Отображать флаг страны**. Который, встраиваясь в дерево главной панели KDE, эмулирует индикатор текущей раскладки клавиатуры — в виде якобы американского и российского флагов соответственно.

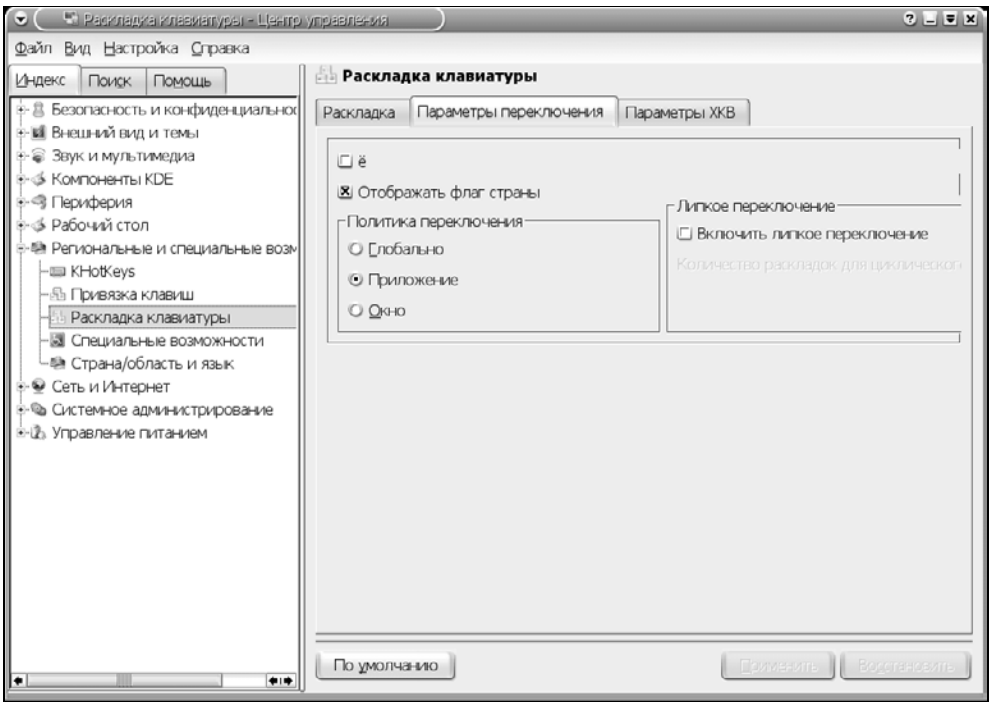


Рис. 17.18. Настройка параметров переключения раскладок

Правда, пользы от этого немного, и по двум причинам: я лично только при большом напряжении фантазии готов признать появившийся значок за какой-либо флаг вообще, а уж опознавать страну его принадлежности не взял-

ся бы вообще; работает этот индикатор обычно только при переключении раскладки мышью, напрочь игнорируя (по крайней мере, у меня так бывало всегда) переключение по назначенной клавише. Так что, поскольку место в дереве ограничено, а там и так уже собирается немало значков (и подчас более важных), флаг страны можно проигнорировать.

Теперь **Политика переключения**, одна из причин, по которой вообще стоит связываться с kxkb. По умолчанию отмечен вариант **Глобально** — то есть переключение раскладки имеет силу для всех открытых (и вновь открываемых) приложений (если кто помнит, так же было и в Windows 3.1). Что, как я уже говорил, не во всех случаях удобно. И потому можно выбрать одну из следующих возможностей — **Приложение**, когда переключение раскладки распространяется на данную программу (и все ее экземпляры, открытые в собственных окнах, как это принято в Windows), или **Окно** — в этом случае каждый экземпляр программы будет автономен в отношении текущей раскладки.

После этого переходим к последней вкладке — **Параметры XKB** (рис. 17.19).

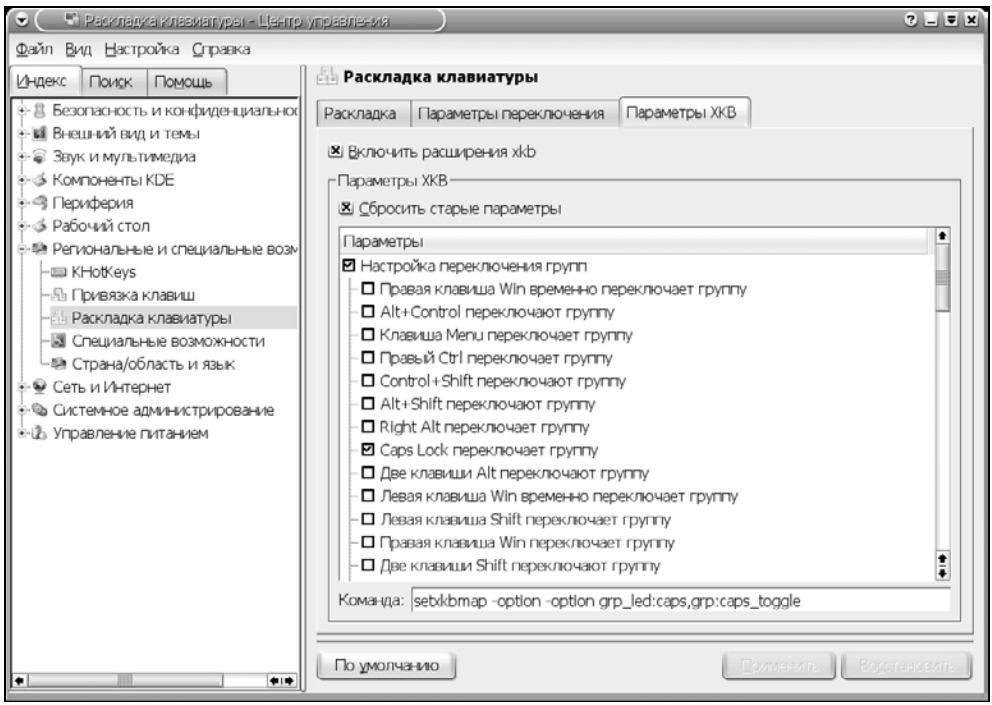


Рис. 17.19. Настройка переключателя раскладок, их индикатора

Здесь для начала нужно включить расширения xkb — я уже говорил, что собственно опции Иксового Xkb Extentions аннулируются при подключении kxkb (точнее, будут аннулированы в момент, когда мы нажмем кнопку **Применить**). И их следует заменить аналогами из нашего модуля KDE.

Для этого перво-наперво отмечаем переключатель **Сбросить старые параметры**, под коими следует понимать опции Xkb, прописанные в файле /etc/X11/XF86Config (или /etc/X11/xorg.conf) — чтобы не мешались. Вообще-то, говорят, что это не обязательно (или обязательно не всегда). Но у меня kxkb нормально работал (после ручной доводки, о которой речь пойдет далее) только при сбросе старых параметров.

Теперь последовательно выбираем клавишу-переключатель (или — комбинацию клавиш, вариантов тут — дюжины полторы, я предпочитаю традиционный CapsLock), светодиод для индикации альтернативной раскладки (например, тот же **CapsLock**) и при желании прочие опции (я их не использую и потому не разбирался с ними).

Вот и все — остается только нажать экранную кнопку **Применить**, чтобы сделанные настройки вступили в силу. И после этого с удивлением обнаружить, что выбранный переключатель ничего не переключает, светодиод — ничего не индицирует, и вообще переключение раскладок возможно только щелчком мышью на том самом флаге, который мы то ли встроили, то ли не встроили в дерево... И возникают сакраментальные русские вопросы: кто виноват, и что делать...

На первый отвечать я категорически отказываюсь, хотя подозреваю, что дело тут все же в некоторых недоработках, не столько даже самого модуля kxkb, сколько способа его интерактивной настройки. Странно только, что недоработки эти тянутся из одной версии KDE в другую, а воз и ныне там. Хотя некоторые положительные тенденции и просматриваются.

А на вопрос "что делать", как обычно, можно ответить двояко: снести kxkb или бороться до победы ручной правкой конфигурационного его файла.

Переходим к мануальной терапии

Возможно, вам покажется, что овчинка (возможность независимого переключения раскладок в разных программах) не стоит выделки (ручной правки конфига). И тогда проще всего отказаться от kxkb. Только не нужно пытаться делать это через Центр управления: отключая раскладки клавиатуры и расширения Xkb — провозитесь вы долго, а результата, скорее всего, не будет ни малейшего.

И потому самое время обратиться к каталогу $HOME/.kde/share/config, помните, в прошлом разделе я говорил, что он нам еще понадобится. А по-

том отыскать в нем файл $HOME/.kde/share/config/kxkbrc — конфиг модуля `kxkb`, просто-напросто удалить его (или переименовать) и перезапустить сессию KDE. Сначала среда эта выругается на отсутствие удаленного файла, но Иксовое переключение раскладок придет в норму, а при следующем старте KDE даже и ругани уже не последует...

Тем не менее, первое решение — это капитуляция, а русские, как известно, не сдаются (особенно русские POSIX'ивисты). Трудностями их тем более не испугать, так что обращаемся ко второму варианту.

Очевидно, что правке подлежит тот же самый файл $HOME/.kde/share/config/kxkbrc. Открываем его в любом текстовом редакторе и внимательно изучаем результаты наших предыдущих действий, воплощенные в опции конфигурации. А выглядит он примерно так:

```
[Layout]
Additional=ru
EnableXkbOptions=true
Includes=
Layout=us
Model=pc104
Options=grp_led:caps,grp:caps_toggle
ResetOldOptions=true
ShowFlag=true
ShowSingle=false
StickySwitching=false
StickySwitchingDepth=1
SwitchMode=WinClass
Use=true
Variants=us(intl),ru(winkeys)
```

Несмотря на некоторую бессистемность списка опций, на первый взгляд, в этом файле все нормально: есть и строка, предписывающая использовать раскладки клавиатуры (`Use=true`), модель клавиатуры (`Model=pc104`), и добавленная русская раскладка (`Additional=ru`) вместе с определением варианта для нее (`Variants=us(intl),ru(winkeys)`), и включение аналогов Xkb (`EnableXkbOptions=true`), и сброс Иксовых настроек (`ResetOldOptions=true`), и демонстрация флага (`ShowFlag=true`), и назначение клавиши-переключателя вкупе с индикацией альтернативной раскладки светодиодом (`Options=grp_led:caps,grp:caps_toggle`), и привязка раскладки к приложению, как в Windows (`SwitchMode=WinClass`). И, тем не менее, никакого переключения не происходит...

И тут мы видим строку

```
Layout=us.
```

Так вот где таилась погибель kxkb — переключать-то не на что! И действительно, стоит только изменить ее таким образом:

```
Layout=us,ru(winkeys)
```

перезапустить KDE-сессию — и все приходит в норму. Правда, как я уже говорил, на флаг-индикатор переключение раскладки назначенной клавишей никакого влияния не оказывает. Но по мне — так и светодиода для индикации русской раскладки вполне достаточно.

Обращаю внимание, что в строке `Layout` для русской раскладки нужно обязательно в скобках указать — `winkeys`, иначе то, что мы определили вариант раскладки в одноименной строке, не возымеет никакого действия.

Вот и все. Единственное, что еще следует помнить, — после ручной доводки не следует перенастраивать клавиатурную раскладку через KCC — результат может быть непредсказуемым, и сеанс мануальной терапии придется повторить. А в остальном, прекрасная маркиза, все не просто хорошо, а так даже замечательно.

Интермедия: универсальный konqueror

- Введение.
- О древовидниках и двухпанельниках.
- Konqueror в его величии.
- Лики konqueror.
- Konqueror в ипостаси браузера.
- Пустячок, но приятный: программа krename.

Эта интермедия посвящается центральной программе KDE — konqueror, универсальному инструменту для управления файлами, выполняющему также роль браузера.

Введение

Каждому пользователю любой ОС, будь то Windows или Linux, DOS или Solaris, не избежать операций с файлами: их просмотра, копирования, перемещения, а иногда — страшно сказать — даже удаления. Хорошо юниксоидам, привыкшим к консольному режиму: для всех этих действий им достаточно командной оболочки, сиречь шелла (shell) и десятка команд. Данные с необходимыми опциями и в должных сочетаниях, команды эти, подобно конторе Кука, предоставят полную информацию о файловой системе, дадут возможность отобрать из изобилия созданных документов те, что потребны в данный момент, и произвести над ними те действия, которые позволят прийти к поставленной цели кратчайшим путем.

А как быть пользователям, с младых ногтей подвергшимся тлетворному влиянию графических интерфейсов, которым командная строка кажется столь же таинственной, как "Книга Мертвых" древних египтян? Конечно, радикальный выход для них — скорее ознакомиться с командами управления файлами (*хотя бы в объеме интермедии к главе 8*). Но пока суть да дело, им на помощь придут программы, именуемые файловыми менеджерами.

О древовидниках и двухпанельниках

Не скажу за Windows (каюсь, по незнанию), но в POSIX-системах (и в этом пользователь должен четко отдавать себе отчет) файловые менеджеры не делают ничего такого, что не могли бы сделать встроенные команды оболочки и штатные утилиты операционной системы (классические UNIX-утилиты). Более того, ни на что большее они не способны в принципе: ведь при любой файловой операции они используют те же базовые функции ядра ОС (системные вызовы), специально для этой цели предназначенные. Которые уже испокон веков и по полной программе задействованы в файловых утилитах POSIX-систем. А подчас файловые менеджеры просто являют собой надстройки над шелл-командами — front-end. Однако красивые интерфейсы и удобные меню могут создать впечатление если не богатства возможностей, то хотя бы простоты их использования.

Все файловые менеджеры можно условно разделить на две категории: двухпанельники и древовидники, по преобладающему представлению в них файловых систем. В силу некоторых причин начнем со вторых.

Древовидные файловые менеджеры ведут свое начало от старинной, ныне почти забытой DOS-программы XTree Gold. И исходят они из метафоры дерева файловой системы. Типичным их современным представителем является Windows Explorer (хотя как раз в нем понятия древовидной организации файловой системы и нет). Преобладающий (и адекватный представлению) способ манипуляции файлами здесь — перетаскивание их мышью из одного каталога в другой, действия с помощью клавиатурных комбинаций занимают подчиненное положение. Может, потому и не получили они популярности во времена "черного DOS" и столь же черной текстовой консоли UNIX, к которым мышь приходилось прикручивать пассатижами...

Двухпанельные файловые менеджеры апеллируют к другой метафоре — списку файлов в каталоге. А поскольку плоский список (в сущности, ничем не отличающийся от вывода команды `ls` в шелле) давал, казалось, немного простора для представления, появилась резонная мысль вывести одновременно два независимых списка файлов, между которыми и осуществляется взаимодействие. Причем здесь определяющей оказывается обычно роль клавиатурных комбинаций (что, конечно, не значит, что в двухпанельниках запрещен Drag&Drop, а в древовидниках нельзя использовать "горячие клавиши"). Впервые двухпанельная метафора была реализована в знаменитом Norton Commander, почему их часто называют еще менеджерами командирского стиля.

В отличие от древовидников, давших достаточно мало (по числу представителей, но не пользователей — вспомним Windows Explorer) отростков, двух-

панельная идея породила множество продолжателей. Тут можно перечислить и отечественный Volkov Commander (без него по сию пору не обходится ни один Reanimator), и столь же родной FAR, и двухпанельник в квадрате Pie Commander, и множество других. Апофеозом же двухпанельников стал Windows Commander, не так давно трасформировавшийся в Total Commander.

В мире POSIX-систем древовидная идея вообще произрастала довольно хило (хотя несколько файловых менеджеров, сделанных по образу и подобию XTree Gold, и существует, но я почти не знаю тех, кто ими реально бы пользовался). А вот двухпанельная идея обрела здесь благодатную почву. И хотя количественно их оказалось немного, один из них — Midnight Commander (mc) — занял в Linux господствующее положение среди всех файловых менеджеров (не считая командной строки, конечно). В мире же BSD-систем получил некоторое распространение отечественный продукт — deco (Demos Commander).

Интересно, что доминирование двухпанельников над древовидниками особенно явно выражено среди отечественных пользователей: популярность на Руси и старого NC, и современных FAR и Total Commander вкупе с mc (и с поправкой на ОС) далеко превосходит общемировой уровень. Достаточно заметить, что все отечественные дистрибутивы Linux непременно включают mc в набор устанавливаемых приложений. Чего в дистрибутивах заграничных эта программа удостаивается не так уж и часто.

Объяснение феномена двухпанельной любви можно найти в изысканиях историков из Екатеринбурга, опубликованных на сайте Neosoft (**http://www.neosoft.ru**).

Исторически сложилось так, что я никогда не мог причислить себя ни к двухпанельникам, ни к древовидникам в чистом виде. XTree Gold прошел как-то мимо меня, развесистые баобабы Windows Explorer наводили ужас. Во времена "черного DOS" я, конечно, прибегал к NC. Однако и тогда набрать к командной строке что-нибудь типа

```
copy from to
```

мне казалось проще, чем рыскать стрелками по панелям и запоминать "горячие клавиши". А уж в Linux или BSD, с их непревзойденными возможностями автоматизации действий в шелле, необходимость в mc или deco вообще возникала достаточно редко, и только на раннем этапе освоения командного интерфейса.

Так и остался бы я лишним на празднике жизни файловых менеджеров, если бы в один прекрасный день не сделал замечательное открытие, имя которому — konqueror.

Konqueror в его величии

Конечно, о существовании konqueror я знал со дня его появления. Ведь эта программа — одно из штатных средств интегрированной графической среды KDE (начиная с версии 2.0), функционирующей поверх любого дистрибутива Linux (собственно, во многих она представляет собой десктоп по умолчанию), FreeBSD или любой другой BSD-системы. И устанавливается эта программа вместе со всей средой, являясь ее непременным и неотъемлемым компонентом.

При первом запуске konqueror (делается это щелчком на пиктограмме **Home** на рабочем столе или панели KDE) не производит впечатления выдающегося произведения программистской мысли: обычный Explorer-подобный файловый менеджер с преобладанием древовидных черт в его облике (рис. 17И.1).

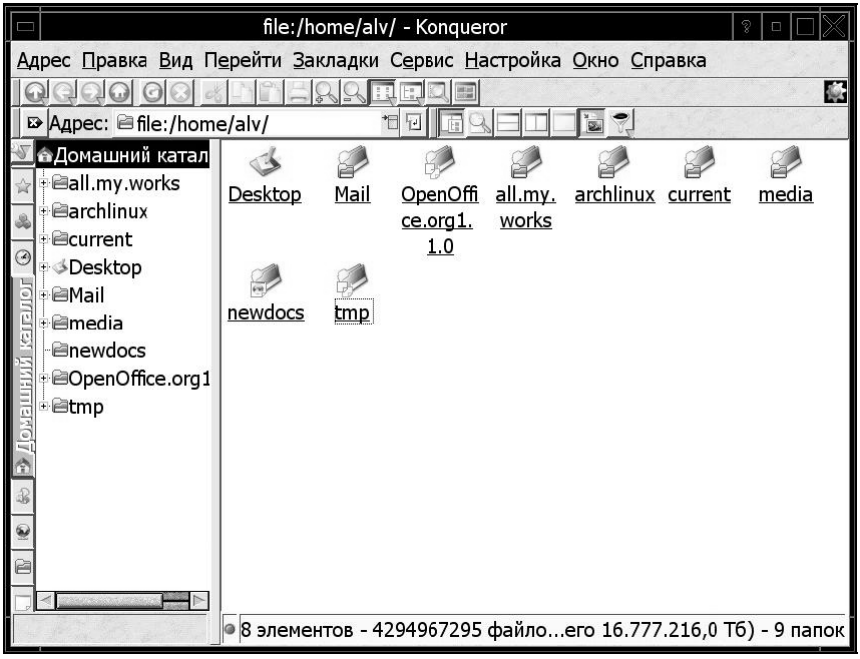


Рис. 17И.1. Вид konqueror по умолчанию.
Грубо слепленный клон Windows Explorer, не так ли?

Конечно, внимательный взгляд задержится на богатых инструментальных панелях: основной (вверху, сразу под меню), дополнительной (чуть ниже — вровень с окном адреса) и навигационной (вертикально по левому краю).

Однако вот на них-то мы останавливаться и не будем, благо, получить представление о функциях каждой кнопки можно из всплывающей подсказки. Потому что более нас интересует вопрос о том, а чего же такого революционного привнес в этот мир konqueror.

Повторяю, на первый взгляд — ничего. И если случайно (или прочитав эту интермедию) не залезть в его настройки, то так до конца жизни можно остаться в неведении относительно бездонных их возможностей. Правда, надо отметить, что настройки эти не вполне логично разбросаны по трем пунктам главного меню: **Вид**, собственно **Настройка** и **Окно**. Может быть, потому они и не получили пока всенародной известности.

Начать с того, что Explorer-подобная ипостась konqueror, хотя и загружается по умолчанию, лишь одна из многих доступных. Зайдя в меню **Окно**, видим, что можно легко отключить файловое дерево навигационной панели, получив просто плоскость с пиктограммами каталогов и файлов. Жизни нашей это отнюдь не украсит, поэтому преобразуем ее в список с именами файлов и их атрибутами.

Правда, для этого потребуется отправиться в меню **Вид** с его пунктом **Режим просмотра**, где среди многочисленных подпунктов выбрать пункт **В виде подробного списка** (или **В виде текста**, что уберет пиктограммы слева от имен файлов). Получаем список имен файлов, а уж отрегулировать подробность представления атрибутов можно в пункте **Показывать подробно** того же меню **Вид**.

Здесь можно последовательно включить/отключить демонстрацию размера и типа файла, времени его изменения (атрибут `mtime`) и последнего к нему доступа (атрибут `atime`), имя владельца и группу, которой он принадлежит, а также атрибуты доступа — в том порядке, в каком мы их подключаем. Что-то напоминает (рис. 17И.2), не так ли? Совершенно верно, да это же практически вывод команды `ls`, только в несколько ином порядке.

К слову, если порядок колонок в нашей таблице не устраивает — любую из них можно просто перетащить мышью, ухватившись за заголовок. Правда, отключить лишние все равно придется через те же пункты меню. А отсортировать файлы по любому параметру (и в любом порядке) можно щелчком мыши по заголовку.

Теперь для пущей двухпанельности остается только вывести параллельно два независимых таких списка. Этого достигаем, вернувшись в меню **Окно** и приказав — **Разделить панель по вертикали**. После чего, скорректировав вывод атрибутов, получаем вполне Norton-подобный вид (рис. 17И.3).

По аналогии с NC, легко догадаться, что между панелями возможен обмен файлами: копирование, перемещение и т. д. Делать, однако, это пока придется либо методом Drag&Drop, либо через пункты главного меню.

Адрес Правка Вид Перейти Закладки Сервис Настройка Окно Справка

Адрес: e/alv/current/free/freebook

Имя	Размер	Тип файла	Изменен	Использован	Владелец	Группа	Права доступа
book01.0.html	3,9 Кб	Документ HTML	04/05 11:48	02/05 13:26	alv	alv	rw-r--r--
book01.1.html	11,5 Кб	Документ HTML	04/05 11:47	02/05 13:26	alv	alv	rw-r--r--
book01.2.html	6,1 Кб	Документ HTML	04/05 11:46	02/05 13:26	alv	alv	rw-r--r--
book01.3.html	5,3 Кб	Документ HTML	04/05 11:45	02/05 13:26	alv	alv	rw-r--r--
book01.4.html	7,1 Кб	Документ HTML	04/05 11:44	02/05 13:26	alv	alv	rw-r--r--
book01.5.html	5,9 Кб	Документ HTML	04/05 11:43	02/05 13:26	alv	alv	rw-r--r--
book01.6.html	5,2 Кб	Документ HTML	04/05 11:42	02/05 13:26	alv	alv	rw-r--r--
book01.7.html	5,8 Кб	Документ HTML	04/05 11:41	02/05 13:26	alv	alv	rw-r--r--
book02.0.html	3,7 Кб	Документ HTML	04/05 11:40	02/05 13:26	alv	alv	rw-r--r--
book02.1.html	4,4 Кб	Документ HTML	04/05 11:39	02/05 13:26	alv	alv	rw-r--r--
book02.2.html	15,9 Кб	Документ HTML	04/05 11:38	02/05 13:26	alv	alv	rw-r--r--
book02.3.html	11,5 Кб	Документ HTML	04/05 11:36	02/05 13:26	alv	alv	rw-r--r--
book02.4.html	9,5 Кб	Документ HTML	04/05 11:35	02/05 13:26	alv	alv	rw-r--r--
book02.5.html	7,9 Кб	Документ HTML	04/05 11:33	02/05 13:26	alv	alv	rw-r--r--
book02.6.html	17,9 Кб	Документ HTML	04/05 11:32	02/05 13:26	alv	alv	rw-r--r--
book02.7.html	7,0 Кб	Документ HTML	04/05 11:31	02/05 13:26	alv	alv	rw-r--r--
book03.0.html	3,7 Кб	Документ HTML	04/05 11:30	02/05 13:26	alv	alv	rw-r--r--

book01.0.html (3,9 Кб) Документ HTML

Рис. 17И.2. Представление konqueror в виде плоского списка — по информативности ничуть не ниже, чем вывод команды `ls` с максимально задействованными опциями

file:/home/alv/current/free/freebook - Konqueror

Адрес Правка Вид Перейти Закладки Сервис Настройка Окно Справка

Адрес: e/alv/current/free/freebook

Имя	Размер	Тип файла
book01.0.html	3,9 Кб	Документ HTML
book01.1.html	11,5 Кб	Документ HTML
book01.2.html	6,1 Кб	Документ HTML
book01.3.html	5,3 Кб	Документ HTML
book01.4.html	7,1 Кб	Документ HTML
book01.5.html	5,9 Кб	Документ HTML
book01.6.html	5,2 Кб	Документ HTML
book01.7.html	5,8 Кб	Документ HTML
book02.0.html	3,7 Кб	Документ HTML
book02.1.html	4,4 Кб	Документ HTML
book02.2.html	15,9 Кб	Документ HTML
book02.3.html	11,5 Кб	Документ HTML
book02.4.html	9,5 Кб	Документ HTML
book02.5.html	7,9 Кб	Документ HTML
book02.6.html	17,9 Кб	Документ HTML
book02.7.html	7,0 Кб	Документ HTML
book03.0.html	3,7 Кб	Документ HTML
book03.1.html	10,3 Кб	Документ HTML
book03.2.html	11,8 Кб	Документ HTML
book03.3.html	7,7 Кб	Документ HTML
book03.4.html	12,5 Кб	Документ HTML
book03.5.html	12,4 Кб	Документ HTML
book04.0.html	4,3 Кб	Документ HTML
book04.1.html	10,7 Кб	Документ HTML
book04.2.html	53,1 Кб	Документ HTML
book04.3.0.html	3,7 Кб	Документ HTML
book04.3.1.html	3,8 Кб	Документ HTML
book04.3.2.html	32,3 Кб	Документ HTML
book04.3.3.html	20,7 Кб	Документ HTML

Имя	Размер	Тип файла
/ chronics	512 б	Каталог
/ files	512 б	Каталог
/ freebook	1,5 Кб	Каталог
/ freebsd	512 б	Каталог
/ freegeo	512 б	Каталог
/ freenotes	1.024 б	Каталог
/ freeprogs	512 б	Каталог
/ links	512 б	Каталог
/ otherbsd	512 б	Каталог
/ otheros	512 б	Каталог
/ sb-linux	512 б	Каталог
/ statistics	1,5 Кб	Каталог
/ templates	512 б	Каталог
/ tmp	512 б	Каталог
/ toolbars	512 б	Каталог
about.html	4,9 Кб	Документ HTML
free.webprj	12,2 Кб	Обычный текст
index.html	5,8 Кб	Документ HTML
links.txt	0 б	Обычный текст
logo.gif	277 б	Рисунок gif
sitemap.html	20,1 Кб	Документ HTML
team.html	3,5 Кб	Документ HTML

book01.0.html (3,9 Кб) Документ HTML chronics/ Каталог

Рис. 17И.3. Konqueror — почти Norton-подобный вид

Есть возможность настроить комбинации "горячих клавиш". Для чего в меню **Настройка** предусмотрен специальный пункт — **Комбинации клавиш**. Он вызывает список доступных действий (рис. 17И.4).

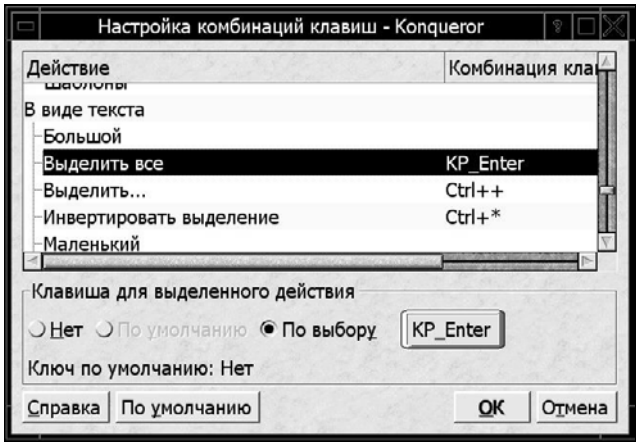


Рис. 17И.4. Настройка "горячих клавиш": еще один шаг к полному Norton-подобию

Аналогично можно поступить и с прочими привычными клавишами — привязать к <F3> функцию просмотра файла, к <F4> — вызов редактора для его модификации (причем не обязательно встроенный, которым по умолчанию KDE выступает KWrite, а любой имеющийся в системе, в том числе и консольный), к <F5> и <F6> — копирование и перемещение файлов соответственно. Правда, тут нужно оговориться: некоторые из клавишных комбинаций могут быть уже задействованы как умолчания самой KDE. Однако и их изменить — не проблема, благо, в нынешних версиях ее сделать это предлагается сразу же (лезть в настройки клавиатуры среды стало не нужно).

Однако сила командира Нортона была отнюдь не только в двух его панелях. А еще и в собственной командной строке, еще в приснопамятные времена далеко превосходившей по своей функциональности COMMAND.COM. И где она здесь? — спросите вы меня.

Отвечу легко: командная строка командира нам тут не потребуется. Ибо, поворотившись обратно же в меню **Окно**, мы увидим там пункт **Показать эмулятор терминала**. И включение его даст нам не просто командную строку, а полноценное терминальное окно с запущенным экземпляром вашей пользовательской оболочки, настроенной в полном соответствии с тем, как это описано в ее профильных файлах (типа ~/.bash_profile — но *это было*

темой главы 15). В этом окне доступны будут и автодополнения команд и путей (нажатием клавиши табулятора), и командная история (прокручиваемая стрелками управления курсором), и управление заданиями (стандартными шелл-операторами типа `&`), и перенаправления ввода/вывода, и вообще все тридцать три удовольствия чисто консольной жизни — но в графическом режиме.

И при этом одну из панелей можно синхронизировать с терминалом: для этого нужно отметить переключатели в нижнем правом углу панели и терминального окна. И тогда мы получаем всю гибкость работы в командной строке плюс полную визуализацию результатов своих действий, именно ее часто не хватает начинающему пользователю при использовании команд типа `cp`, `mv` или `rm`. Вплоть до возможности перетаскивания файлов в строку шелла методом Drag&Drop — для задания их имен как аргументов команд.

Вид терминала может быть настроен в очень широких пределах. Достаточно щелкнуть в его поле правой кнопкой мыши, и из появившегося контекстного меню можно изменить (рис. 17И.5):

- размер и гарнитуру шрифта (на шрифте панелей это никак не скажется — тот изменяется независимо);
- так называемый тип терминала, что в русской версии озаглавлено почему-то как **Клавиатура**;
- цветовую схему (черным по белому или наоборот, а можно — даже и с прозрачным фоном, сквозь который будут просвечивать любимые обои рабочего стола).

Нужно только не забыть сохранить настройки, добившись оптимального результата.

Пару слов специально об установке типа терминала. Для нормального поведения клавиш управления курсором здесь настоятельно рекомендую **Linux console**: даже во FreeBSD это даст привычное поведение кнопок типа **Home** и **End** в большинстве командных оболочек. А вот для tcsh добиться того же мне удавалось только при типе терминала XTerm (XFree 4.x.x).

Вообще говоря, терминальное окно konqueror по своим свойствам и функциям идентично программе эмулятора терминала из комплекта KDE — konsole. Так что все сказанное относится и к ней, избавляя меня от необходимости введения дополнительного раздела. Отмечу только, что сама по себе konsole — очень мощная и удобная программа, позволяющая создавать множество связанных окон с навигацией по ним с помощью вкладок или клавишных комбинаций. А если последние определить как <Alt>+<F#>, то можно добиться полной иллюзии работы в обычной текстовой консоли.

И еще к слову о настройках вообще. Большинство из выполненных нами ранее действий будут иметь силу только для запущенной в данный момент

копии konqueror. Чтобы сделать их перманентными, требуется сохранить настройки в профиле.

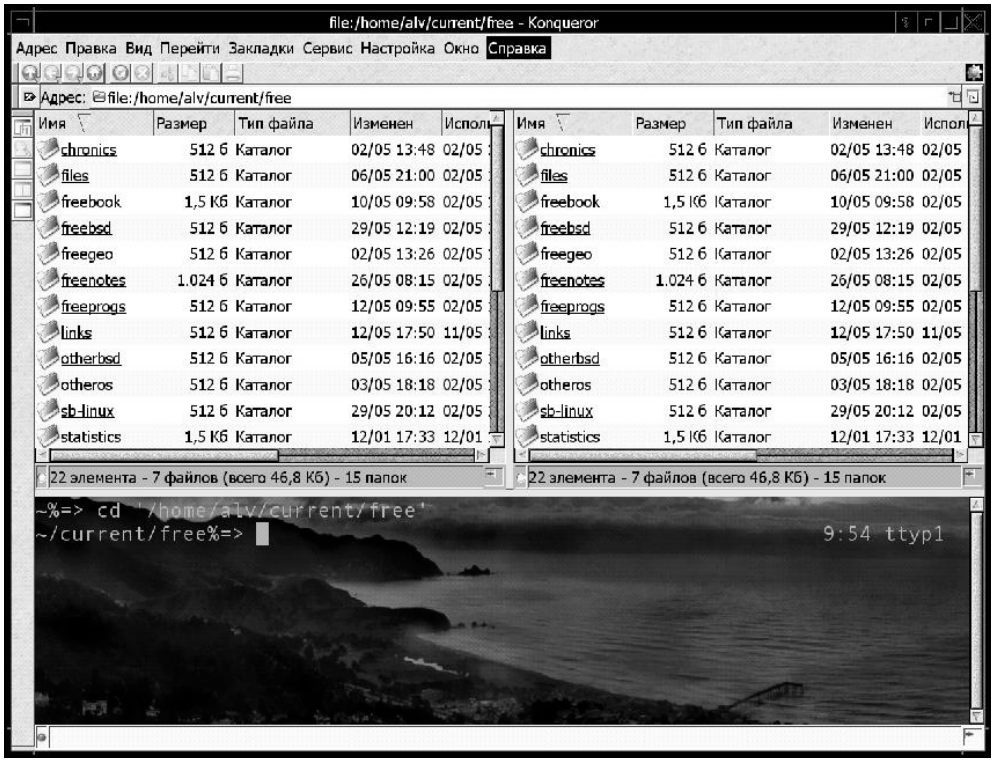


Рис. 17И.5. Konqueror — итоговый вариант: и терминал бывает украшением рабочего

Лики konqueror

В списке предлагаемых профилей их можно обнаружить несколько, в том числе цель наших предыдущих манипуляций (имитацию внешности Midnight Commander, а мы затратили на это столько трудов!), а также интересный профиль предварительного просмотра файлов, в котором мы имеем (рис. 17И.6) навигационную панель, панель содержимого каталогов и панель просмотра файлов.

Ничто не в силах помешать нам создать и собственный профиль konqueror, взяв за основу один из иных существующих.

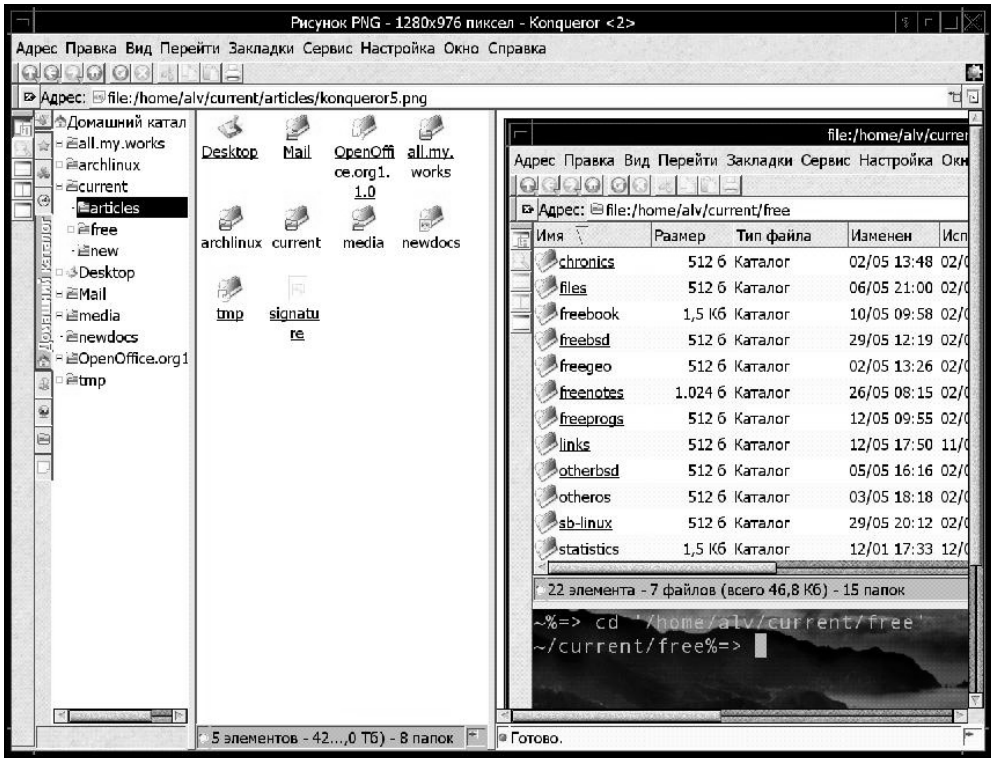


Рис. 17И.6. Предварительный просмотр файлов, слева направо: навигационная панель, фиксация на которой выбирает каталог, панель содержимого каталога — в ней мы выбираем файл, отображаемый в панели просмотра

Для этого ему нужно только присвоить имя в момент сохранения. Правда, щелчком по все той же пиктограмме **Home** по-прежнему будет вызываться прежний профиль по умолчанию. Можно ли это изменить?

Ну, конечно же, можно.

1. Отправляемся в панель запуска KDE.
2. Отыскиваем там нужный нам значок (в большинстве тем это в той или иной степени стилизованное изображение домика).
3. Щелкаем на нем правой кнопкой мыши.
4. В появившемся контекстном меню выбираем пункт **Свойства**.
5. В возникшей панели переходим на вкладку **Приложение**.
6. Отыскиваем на ней поле **Команда** (рис. 17И.7).

7. В поле этом остается просто заменить имя вызываемого профиля, выступающего как аргумент команды `kfmclient` (в оригинале это имеет вид `kfmclient openProfile filemanagement`), на имя, придуманное ранее, при сохранении профиля (типа `kfmclient openProfile my_profile_name`).

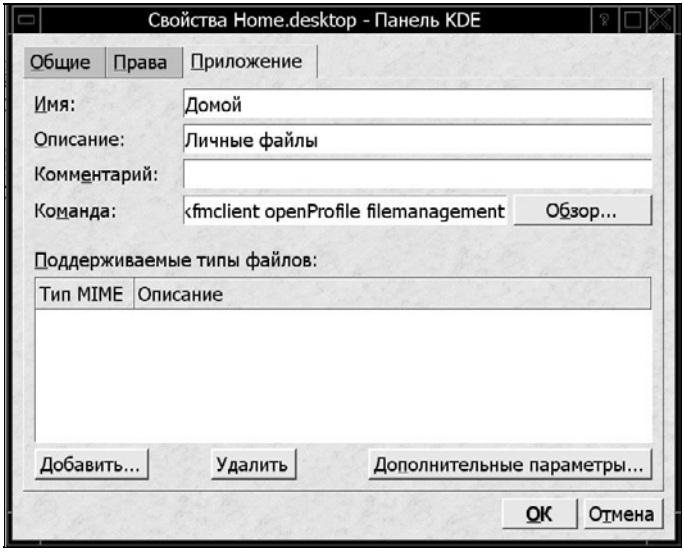


Рис. 17И.7. Создание собственного профиля konqueror, вызываемого по умолчанию через главную стартовую панель KDE

Konqueror в ипостаси браузера

К слову сказать, среди имен предопределенных профилей можно обнаружить и такое — webbrowsing. Да-да, konqueror — не только файловый менеджер-организатор, но и Интернет-агитатор — Web-браузер. Причем, начиная с KDE версии 3.0, вполне сравнимый по функциональности с такими браузерами, как Mozilla или Opera (и далеко превосходящий Internet Explorer). Что немаловажно в наших условиях — без проблем справляющийся с любыми кириллическими кодировками, каковые могут быть определены автоматически или установлены вручную. Правда, автоматическое определение кодировки основывается на значении метатега `charset`, и при его отсутствии на странице (а многие Web-мастера не утруждают себя указанием набора символов) не сработает.

Если для полноценного Web-серфинга возможностей konqueror-браузера может и не хватить (хотя, по моим потребностям, их немного больше, чем

вдоволь), то уж для локального просмотра HTML-файлов (например, документации) трудно придумать что-либо более удобное — вследствие быстродействия, простоты использования и интеграции с файловым менеджером. Ведь в последнем Web-документ открывается (в той же панели) простым щелчком мышью по имени соответствующего файла. А если этот щелчок выполнить нажатием правой кнопки нашего грызуна, то появляется выбор: открыть ли нужный файл в новой вкладке (уже без разделения на панели) или новом окне.

Разумеется, запустить konqueror в реинкарнации Web-браузера можно и сам по себе, для этого на стартовой панели KDE по умолчанию имеется соответствующая пиктограмма (нечто вроде земного шарика, обрамленного шестеренкой). Просмотр ее (то есть пиктограммы) свойств показывает, что стартует браузер командой `kfmclient openProfile webbrowsing`, то есть различия с файловым менеджером только в аргументе запускающей команды. Из чего заключаем, что и для него можно создать любой собственный профиль.

А поводов для этого konqueror-браузер дает сколько угодно. Потому что в нем, как и в файловом менеджере, можно настроить все, что душе угодно: и шрифты для отображения Web-страниц, и вид и состав инструментальной панели, и положение панели вкладок, и многое, многое другое.

При желании можно придать браузеру двухпанельный вид или отобразить в нем окно терминала (рис. 17И.8). Такое представление оказывается очень не лишним при работе с FTP-архивами: файлы из них можно копировать точно так же, как это делается на локальной машине (перетаскиванием мышью, горячими клавишами, или просто из командной строки). Правда, при условии, что в konqueror будет интегрирован собственный FTP-клиент KDE — kget (который, напомню, входит не в основной kdebase, как konqueror, а в отдельный пакет kdenetworks). Впрочем, он это проделывает сам собой при первом же запуске (если специально не отказаться от этой возможности). А можно из командной строки воспользоваться каким-либо консольным FTP-клиентом, типа wget или lftp — возможности и того, и другого далеко выходят за рамки функций большинства "качалок" графического режима. В этом случае и kdenetworks не понадобится...

Подробным описанием возможностей konqueror в роли браузера я заниматься не буду. Как и его настройками (выполняются они очень просто и по аналогии с настройкой konqueror — файлового менеджера). Хотя на отдельных деталях позволю задержать ваше внимание.

Одна из привлекательных черт konqueror в роли браузера — использование концепции вкладок (Tabs). Появившись впервые в Opera (основанной, к слову сказать, на той же библиотеке Qt), эта концепция была внедрена чуть ли не во все браузеры графического режима. И нынче ей можно удивить

разве что пользователей Internet Explorer. Однако в konqueror она реализована очень полно — практически также, как в матушке-Opera.

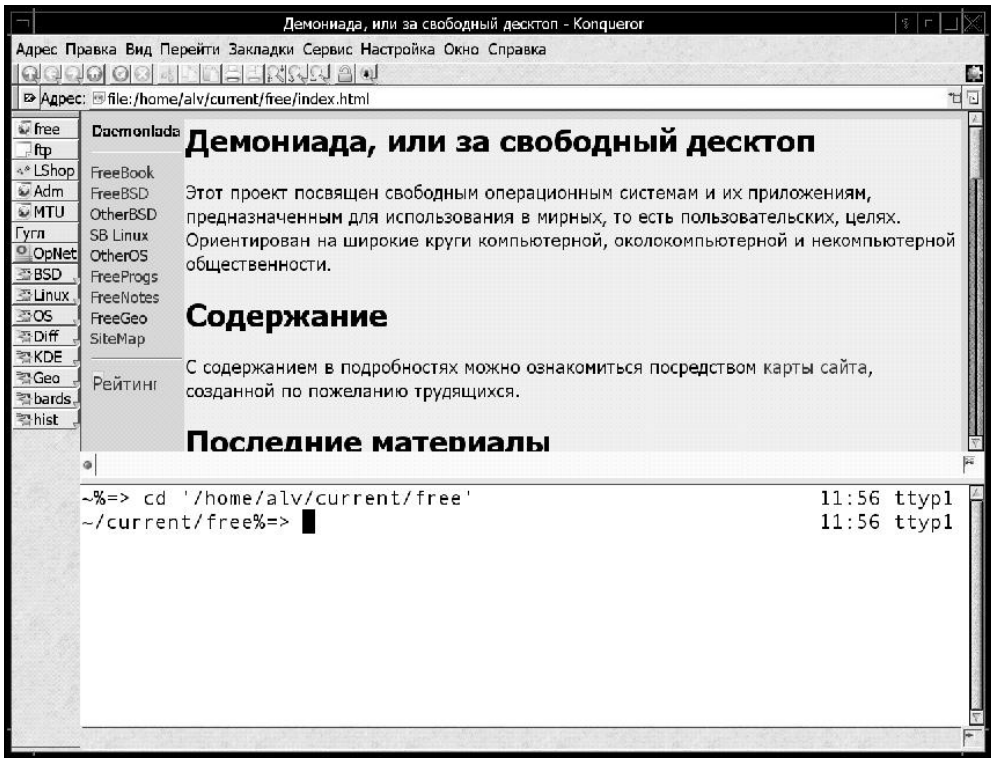


Рис. 17И.8. Konqueror в ипостаси браузера и с терминальным окном. Для чего? Да для запуска консольного FTP-клиента, например...

Кого из нас не раздражали сайты, авторы которых склонны злоупотреблять атрибутом `target=_blank` в тегах гиперссылок? В результате чего любой щелчок по ссылке приводит к открытию нового окна, а те мгновенно заполоняют собой весь экран, сколь бы большим он не был. Конечно, с этим можно бороться: щелкнув на ссылке правой кнопкой и выбрав из контекстного меню пункт **Открыть в новой вкладке**. Однако можно и решить эту проблему раз навсегда. Для чего отправиться в меню **Настройка**, выбрать там пункт **Настроить konqueror** и в панели **Поведение Web** отметить переключатель **Открывать ссылки в новой вкладке, а не в новом окне**. А через кнопку **Дополнительные параметры** распространить действие этой опции даже на всплывающие окна (pop-up).

Можно, конечно, запретить и загрузку по умолчанию графических элементов страницы — особенность, весьма ценная с развитием городских сетей, услуги которых оплачиваются не по времени, а по трафику. К сожалению, в konqueror нет замечательной возможности, реализованной в Opera: одним щелчком мыши отменить авторский стиль сайта, заменив его собственным. Однако глобально такую подмену выполнить можно — через вкладку **Стили CSS** в том же меню **Настроить conqueror**.

Пустячок, но приятный: программа krename

А в заключение этой интермедии я хотел бы рассказать о замечательной программе, имеющей прямое отношение к проблеме управления файлами, хотя и не входящей в konqueror и даже в комплект KDE вообще, но созданной для работы в этой среде. Программа эта — krename, и предназначена она для массового переименования файлов. Чтобы оценить ее удобство, представим себе такую ситуацию.

Дано: массив старых файлов, имена которых записаны в формате DOS в виде 8.HTM, которые нужно включить в новый сайт и, соответственно, переименовать для единообразия в вид имя.html. Пользователя Windows сама мысль о количестве кликов вгонит в дрожь, не придут на помощь тут и всякого рода FAR и Total Commander.

Для записного юниксоида решение лежит на поверхности: нужно написать простенький скрипт, выполняющий такую операцию за один раз. И это действительно несложно: я для этой цели пользовался оператором `for`, вероятно, можно придумать и другие способы. Однако такое сиюминутное решение (согласитесь, ведь предложен далеко не самый сложный случай из реально возможных) будет применимо только к файлам с определенными масками имен. Можно, конечно, поднатужиться и придумать чуть более сложный сценарий, универсальный, допускающий задание произвольных масок, как для заменяемых, так и для заменяющих имен. А это и будет база программы вроде rename — так не лучше ли положиться на нее, снабженную к тому же удобным графическим интерфейсом, нежели изобретать велосипед?

Если последнее покажется более простым, остается только установить программу любым способом. Она есть в портах и пакетах FreeBSD, вероятно, в "больших" дистрибутивах Linux, на худой конец — просто собрать из исходников (**http://kde-apps.org**), потому что текущая в данный момент версия krename не всегда собирается с KDE произвольной версии. И потому это тот случай, когда проще воспользоваться штатными средствами дистрибутива в надежде на то, что в нем совместимость KDE и krename протестирована.

А теперь запускаем krename. При первом запуске он предложит выбор режима: использование мастера (wizard) или режим эксперта. Для начала выберем первый, полуавтоматический режим: при необходимости переключиться позднее в режим эксперта труда не составит.

Теперь остается выбрать файлы для переименования. Они могут находиться в произвольных местах файловой системы, чтобы включить в список файлы из другого каталога, достаточно прибегнуть к кнопке **Добавить**. А закончив с отбором, обратиться к кнопке **Далее**, чтобы перейти к следующему шагу — выбору условий переименования. Ибо krename позволяет (рис. 17И.9) сохранить файлы в исходном каталоге (каталогах) под новыми именами с сохранением копий под старыми именами или без оного, скопировать или переместить переименованные файлы в произвольный каталог, а также создать сценарий отмены переименования (или использовать существующий), для чего потребуется только задать его имя — сам сценарий запишется автоматически. А выполнить его можно будет через меню **Дополнительно | Откат старого переименования**.

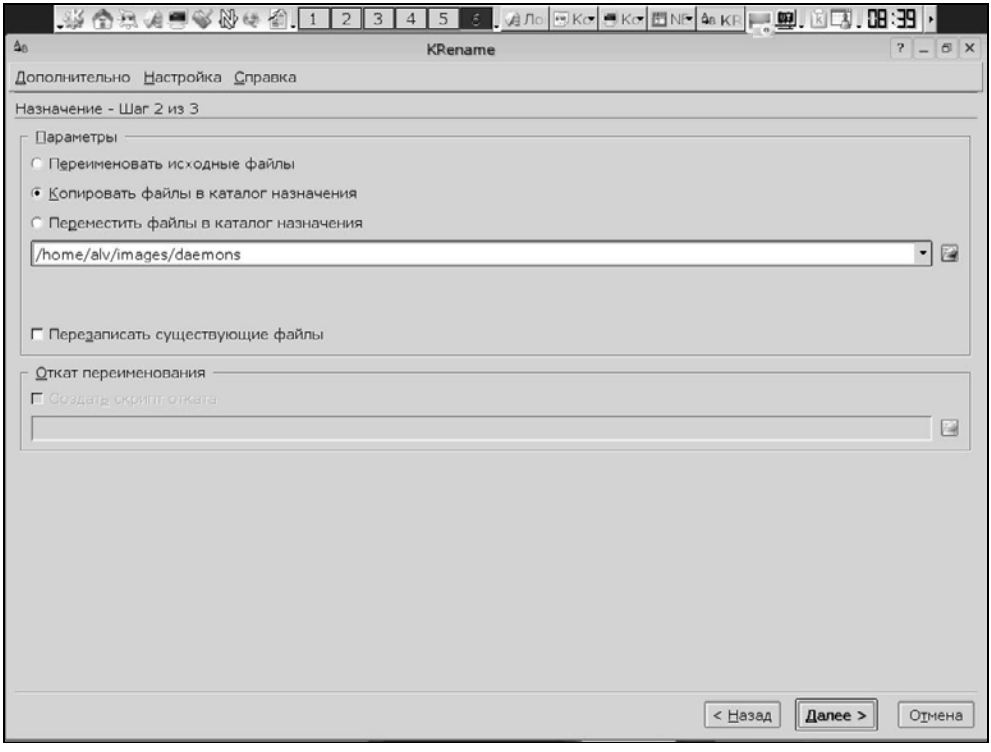


Рис. 17И.9. Krename, выбор условий переименования

На следующей же стадии задается собственно схема переименования (рис. 17И.10). При этом можно частично использовать их старые имена и "расширения", дополнив их суффиксами и (или) префиксами; при этом символы старых имен могут быть преобразованы в верхний или нижний регистр. В качестве суффиксов и префиксов могут использоваться номер, дата или просто произвольный набор символов.

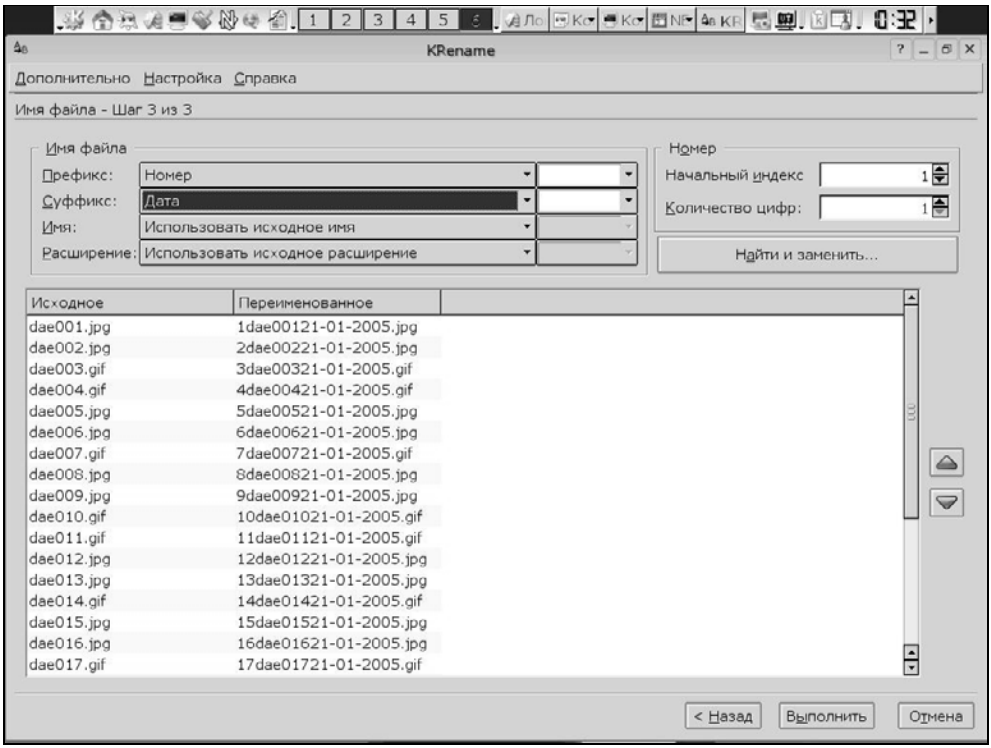


Рис. 17И.10. Определение схемы переименования файлов

С помощью кнопки **Найти и заменить** можно предварительно в именах всех выбранных файлов заменить одни их части на другие, в том числе и с использованием регулярных выражений (рис. 17И.11). А можно вообще отказаться от сохранения исходных имен файлов, задав для всего массива собственную маску.

Таким образом легко превратить рассеянные по разным каталогам и бессистемно именованные файлы изображений в единый массив иллюстраций к некоему материалу вида ris01.tiff ... ris#.png, собрав их заодно в едином подходящем каталоге (например, ~/book/ill). Однако это не все, что

может делать krename: в режиме эксперта она обретает дополнительные возможности.

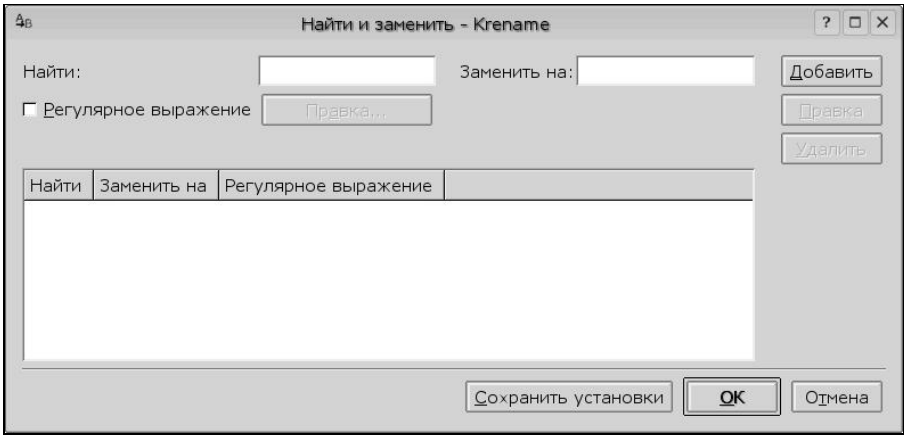


Рис. 17И.11. Частичная коррекция имен исходных файлов перед их переименованием

Переход в режим эксперта выполняется через меню **Настройка | Настроить krename**, где во вкладке **Интерфейс** следует отметить переключатель **Использовать вкладки** (для опытных) (рис. 17И.12).

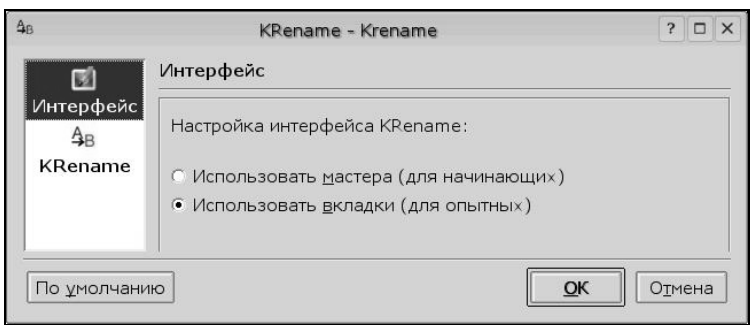


Рис. 17И.12. Переключение режимов krename

В режиме мастера интерфейс krename преобразуется в вид с вкладками (рис. 17И.13). В первой из них (**Файлы**) осуществляется отбор файлов для переименования, во второй (**Назначение**) — определяются каталоги для помещения переименованных файлов — все точно так же, как было описано ранее.

krename

Дополнительно Настройка Справка

Файлы | Назначение | Модули | Имя файла

/home/alv/images/daemons/daemon01.jpg
/home/alv/images/daemons/daemon02.jpg
/home/alv/images/daemons/daemon03.gif
/home/alv/images/daemons/daemon04.gif
/home/alv/images/daemons/daemon05.jpg
/home/alv/images/daemons/daemon06.jpg
/home/alv/images/daemons/daemon07.gif
/home/alv/images/daemons/daemon08.jpg
/home/alv/images/daemons/daemon09.jpg
/home/alv/images/daemons/daemon10.gif
/home/alv/images/daemons/daemon11.gif
/home/alv/images/daemons/daemon12.jpg
/home/alv/images/daemons/daemon13.jpg
/home/alv/images/daemons/daemon14.gif
/home/alv/images/daemons/daemon15.jpg
/home/alv/images/daemons/daemon16.jpg
/home/alv/images/daemons/daemon17.gif
/home/alv/images/daemons/daemon18.jpg

Добавить...
Удалить
Удалить все
Сортировка: без сортировки
Предосмотр
Отображать имя
Файлов: 27

Выполнить Отмена

Рис. 17И.13. Режим эксперта, вкладка **Файлы**

krename

Дополнительно Настройка Справка

Файлы | Назначение | Модули | Имя файла

Шаблон: ris##

Функции...

Использовать расширение исходного файла

Расширение файла начинается с: Последняя точка

Найти и заменить... Нумерация... Вставить часть имени файла...

Исходное	Переименованное
daemon01.jpg	ris01.jpg
daemon02.jpg	ris02.jpg
daemon03.gif	ris03.gif
daemon04.gif	ris04.gif
daemon05.jpg	ris05.jpg
daemon06.jpg	ris06.jpg
daemon07.gif	ris07.gif
daemon08.jpg	ris08.jpg
daemon09.jpg	ris09.jpg
daemon10.gif	ris10.gif

Выполнить Отмена

Рис. 17И.14. Режим эксперта, вкладка **Имя файла**

Схема переименования задается во вкладке **Имя файла** (рис. 17И.14). Можно видеть, что, кроме обычной замены по маске, как в режиме мастера, прибавились такие возможности, как переопределение "расширения" (за начало его можно принять не последнюю точку в имени, а первую), при использовании номеров в именах (сделать нумерацию не сквозной, а с пропуском произвольных вписанных номеров), а также задействовать некие функции.

Среди функций программы krename — полтора десятка встроенных, общего назначения, частично задействованных и в режиме мастера (смена регистра символов в именах, например), но большей частью доступных только в режиме эксперта (создание промежуточных каталогов для помещения переименованных файлов, манипуляция с единичными символами в именах). Кроме того, имеются функции специально для обработки контента файлов множества форматов — графических, мультимедийных, текстовых (включая PostScript) и пакетных (RPM, DEB). Так, функции для всех графических форматов позволяют поменять глубину цвета и разрешение (в том числе раздельно по вертикали или горизонтали), для аудиофайлов — создать список песен в HTML-формате и т. д.

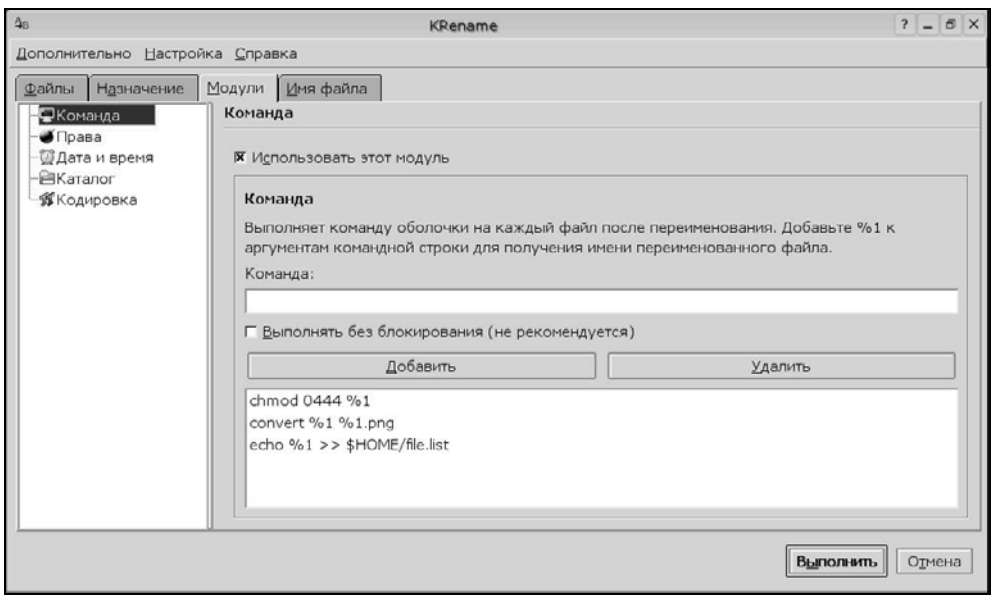


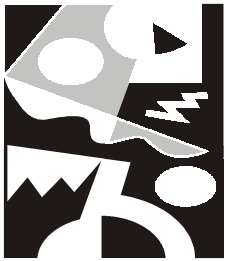
Рис. 17И.15. Режим эксперта, закладка **Модули**

Наконец, вкладка **Модули** позволяет произвести над переименованными файлами дополнительные действия (рис. 17И.15): выполнить в их отноше-

нии команду оболочки (из списка или произвольную), изменить атрибуты принадлежности и доступа (в рамках полномочий данного пользователя, конечно), а также времени (atime и mtime), отсортировать переименованные файлы в каталоге и даже изменить набор используемых в именах символов (например, KOI8-R на UTF8).

Таким образом, возможности программы krename далеко выходят за рамки простых сценариев оболочки. И их реализация путем шелл-скриптинга потребовала бы весьма изощренного программирования (мне, например, такое было бы не по силам). Так что использование ее вполне оправданно даже для опытных пользователей: время, сэкономленное на сочинении собственных скриптов переименования файлов, лучше употребить на совершенствование контента оных...

Глава 18



Текстовые редакторы

- Вводные замечания.
- Nano: входной билет в мир редакторов.
- Несколько слов о ее.
- Редакторы vi и Vim: введение в тему.
- Joe: гармония простоты и функциональности.
- NEdit — текстовый редактор для Иксов.

Думается, не будет большим преувеличением сказать, что из всех приложений POSIX-подобных ОС важнейшими являются текстовые редакторы. Тем более что они лежат на грани программ общесистемных и пользовательских. Ведь с помощью редактора (а иногда и исключительно при его посредстве) настраивается система, редактируются общесистемные и пользовательские конфиги, пишутся скрипты и сценарии. А некоторые пользователи, подобно вашему покорному слуге, даже используют текстовый редактор и по прямому назначению — просто для составления повествовательных текстов.

Вводные замечания

Пользователи Windows в качестве универсального средства для работы с текстами в большинстве случаев используют программы, именуемые word-процессорами. Однако в POSIX-системах именно редакторы — традиционный инструмент всех UNIX-систем.

Все текстовые редакторы POSIX-мира можно разделить на два класса: редакторы командного стиля и меню-ориентированные редакторы. В первых навигация по тексту и его обработка осуществляются отдачей прямых директив, вроде: перейти на пять слов вперед, удалить пятую снизу строку, заменить строку номер пятнадцать и т. д. Действия в меню-ориентированных редакторах, как и следует из названия, осуществляются более интерактивно (и более привычно для пользователя Windows-редакторов).

Разделение редакторов на командные и меню-ориентированные, отчетливо выраженное для консольных представителей этого семейства, несколько сглаживается при переходе к работе в графическом режиме. Потому что в Иксах даже типичные командные редакторы обретают часто (при соответствующих опциях сборки) элементы визуального интерфейса. Однако командная сущность gvim или kvim не меняется от того, что они обретают меню в стиле Gtk или KDE соответственно. А, например, NEdit являет собой вполне гармоничное сочетание командного и визуального стилей редактирования, хотя без соответствующих настроек об этом можно и не догадаться.

Текстовые редакторы обладают рядом преимуществ перед процессорами, по крайней мере, в той области, для которой они предназначены, то есть при создании и обработке текстов. Главное из них — это универсальность. Поскольку выходной материал в редакторах представляет по определению чистый ASCII-файл, он может быть прочитан в любой среде и на любой платформе, не требуя специальных конверторов (и, тем более, программ, в которых этот материал создавался). Это особенно важно для документов с символами кириллицы. Кроме того, с помощью редакторов можно готовить HTML-страницы, осуществлять верстку документов для TeX, править конфигурационные файлы, писать тексты программ и многое другое.

Кроме того, редакторы существенно удобнее процессоров для составления длинных структурированных текстов нарративного характера, каковое, особенно при претензиях на оригинальность, требует сосредоточенности, трудно достижимой при изобилии форматирующих возможностей процессоров. Впрочем, это мое субъективное мнение. Однако остается фактом, что подготовленный в редакторе документ в дальнейшем может быть трансформирован в любой текстовый процессор или систему верстки, где его можно подвергать любому оформлению.

Следует заметить, что большинство традиционных UNIX-редакторов принадлежат к командному классу. Среди его представителей, с одной стороны, наиболее простые и легковесные (с точки зрения требований к ресурсам) программы. К ним принадлежат, например, редакторы nano и ee, описанные в ближайших разделах.

С другой стороны, именно командными редакторами являются наиболее мощные инструменты комплексной обработки текста, такие как vi (вернее, его современная модификация vim) и emacs.

Наконец, существует категория редакторов, занимающих по своей функциональности промежуточное положение между мощными средствами типа vim и совсем простыми редакторами класса nano. Среди них типично командный редактор для консоли joe.

Чем же руководствоваться при выборе редактора? Если стоит задача достижения максимальной функциональности, с emacs или vim мало кто в со-

стоянии конкурировать (не зря же emacs удостоился звания операционной среды — того же титула, который некогда присвоила Microsoft своему Windows 3.x). А вот если основная задача выбираемого редактора — правка конфигурационных файлов, возникает вопрос, а почему бы не ограничиться программами легчайшей весовой категории?

К тому же проблема выбора редактора, как и все в жизни, имеет оборотную сторону: чем богаче возможности, тем больше времени требуется на их освоение. А если претензии на первое время невелики, но это самое первое время ограничено, выбор редакторов-легковесов как бы напрашивается.

Поэтому в настоящей главе будут кратко охарактеризованы простейшие редакторы nano и ee, инструменты конфигурации во многих дистрибутивах Linux и в BSD-системах. Столь же кратким будет введение в работу с vim, потому что это то средство, которое будет под рукой в любой UNIX-системе. Далее рассмотрению подвергнется joe, как типичный представитель консольных редакторов промежуточной категории. И, наконец, подробное описание NEdit — самого мощного из текстовых редакторов графического режима.

Nano: входной билет к мир редакторов

Редактор nano вполне может сыграть роль своего рода амортизатора для начинающего пользователя. С задачей конфигурирования справляется успешно. А в освоении и обращении прост, как грабли. Не случайно во многих Source Based дистрибутивах он предлагается в качестве общесистемного. В Gentoo Linux же, где при установке необходимость в ручном редактировании конфигурационных файлов возникает весьма часто, так это просто единственный редактор, доступный на стадии инсталляции системы.

Итак, представляю: редактор nano, или, точнее, GNU nano. Официальным местопребыванием имеет сайт **http://www.nano-editor.org**. Генетически связан с pico — текстовым редактором, входящим в почтовый пакет pine, но распространяется на условиях лицензии GPL (и, что немаловажно, не тянет за собой почтовой системы, возможно, не всем нужной). Характеризуется авторами как маленький и дружелюбный. Что в целом соответствует истине.

Редактор nano чисто консольный и запускается из строки шелла одноименной командой, можно — с указанием имени файла, существующего или нового (в последнем случае, как обычно, файл с таким именем будет создан). Поддерживается несколько опций командной строки, как-то:

- `-T #`, устанавливающей величину (в символах) табуляции;
- `-i`, включающей автоматические отступы;
- `-w`, отключающей режим переноса строк на границе экрана (что очень важно при редактировании конфигурационных файлов).

Полный их список можно посмотреть посредством

```
$ man 1 nano
```

После запуска nano перед глазами возникает вверху титульная строка, в которой выводятся номер версии программы, имя открытого файла и в правом углу сообщение о том, что файл был изменен. В нижней части экрана можно видеть зону подсказки — список основных из управляющих клавишных последовательностей с пояснениями.

Область между титульной строкой и зоной подсказки — рабочая, в ней осуществляется ввод и редактирование текста. В nano предусмотрен только один режим работы: текст вводится обычным образом, а для вызова команд предусмотрены управляющие последовательности.

Существуют два вида управляющих последовательностей: <Ctrl>+символ и <Meta>+символ. Посредством первых (частично дублируемых функциональными клавишами) осуществляется редактирование текста и операции с файлами. <Meta>-последовательности предназначены для изменения настроек редактора (тот же результат достигается и опциями командной строки).

Напомню, что на клавиатуре PC роль клавиши <Meta> выполняет обычно нажатие клавиши <Alt> или нажатие и отпускание клавиши <Esc>.

Собственно, это и все. Функциональные возможности nano отнюдь не производят впечатления исключительно богатых. Однако со своей ролью — несложной правкой небольших конфигурационных файлов — он вполне справляется. И, к тому же, в нем предусмотрено еще и внешнее средство конфигурирования — пользовательский конфиг ~/nanorc. Выполнив в нем некоторые манипуляции, можно несколько расширить функциональность редактора, в частности, обеспечить подсветку синтаксиса.

Несколько слов о ee

Редактор ee (Easy Editor) прост и незатейлив, и именно поэтому рекомендуется как базовый во FreeBSD. Он входит в состав базовой системы (Distributions) и, собственно, является там чуть не единственным представителем этого семейства. В базовой системе он замаскирован под классический vi и потому функционально ограничен и поначалу не особенно легок в использовании. Так что на первом этапе освоения FreeBSD (и DragonFlyBSD) ee может быть столь же естественным выбором, как и nano для начинающего пользователя Linux.

Запускается ee одноименной командой

```
$ ee file,
```

где `file` (необязательный аргумент) — имя файла, подлежащего редактированию. После этого открываются два окна редактора. В нижнем — редакти-

руемый текст (если файла с указанным именем не существует, или ее запущен без аргумента, окно это остается пустым), в верхнем — краткая, но практически исчерпывающая справка по использованию программы.

Навигация по тексту осуществляется или обычным способом — клавишами управления курсора либо специфичными для ее управляющими (или командными) комбинациями клавиши <Ctrl> с какой-либо из литерных. Так, комбинация <Ctrl>+<f> (от forward) перемещает курсор на один символ вправо (вперед), <Ctrl>+<b> (от backward) — на один символ влево (назад) и т. д. Причем возможности управляющих комбинаций шире, например, они позволяют перейти к следующему слову (<Ctrl>+<z>), перейти в начало (<Ctrl>+<t>) или конец (<Ctrl>+<u>) экрана.

Аналогично и редактирование текста можно выполнять двояким образом: посредством клавиш редактирования (<Backspace>, <Delete>) или такими же управляющими комбинациями. Причем последний способ, как и в случае навигации, более эффективен, позволяя удалить не только отдельный символ (<Ctrl>+<d>), но и целиком слово (<Ctrl>+<w>) или строку (<Ctrl>+<k>).

Редактор ее не является истинно командным, и основные действия в нем осуществляются через меню, вызываемое нажатием клавиши <Esc>. Пункты главного меню — следующие (рис. 18.1):

```
+------------------------+
| a) выйти из редактора  |
| b) подсказка           |
| c) операции с файлами  |
| d) обновить экран      |
| e) параметры           |
| f) поиск               |
| g) разное              |
+------------------------+
```

Рис. 18.1. Пункты меню

Смысл всех пунктов вполне понятен, и пользователь легко освоит их эмпирическим методом (тем более, что многого осваивать и не требуется). Скажу только, что при выходе из редактора выдается запрос на сохранение изменений в файле, буде таковые производились (рис. 18.2).

И еще следует заметить, что вывод подсказок и сообщений на русском языке (как в приведенном примере) происходит только при правильной локализации, установке русской локали (locale).

```
+------------------------+
| Файл изменён!          |
|                        |
| a) сохранить и выйти   |
| b) не сохранять        |
|                        |
| для отмены нажмите Esc |
+------------------------+
```

Рис. 18.2. Запрос на сохранение

Редакторы vi и Vim: введение в тему

Редактор vi — непременный атрибут всех UNIX-систем, и потому любой пользователь должен иметь о нем представление, хотя для повседневной работы какой-либо иной редактор может показаться ему более подходящим.

Поначалу vi может показаться порождением больного ума с садомазохистскими наклонностями. Однако достаточно осознать внутреннюю его логику, и начинаешь понимать, что более быстрого инструмента для обработки текста человеческий разум еще не придумал. А многочисленные возможности для его настройки обеспечивают должную функциональность такой обработки.

Существует несколько редакторов, основанных на vi, включающих дополнительные возможности, но полностью совместимых с ним по системе базовых команд. И потому знание vi обеспечит возможность работы с любым из его клонов. Более того, если в некоей UNIX-системе вместо vi используется какой-либо из редакторов-клонов, например, Vim или elvis, в каталоге /usr/bin всегда будет представлен файл vi, являющийся жесткой или символической ссылкой на реальный исполнимый файл этого редактора. Поэтому vi или любой его аналог всегда может быть запущен командой

```
$ vi имя_файла
```

Реально при этом запускается, например, Vim. Тем более, что сам по себе vi в свободных POSIX-системах не используется из-за лицензионных соображений. Так что далее термины vi и Vim используются как синонимы, но везде имеется в виду Vim (Vi Improved).

Редактор Vim может быть запущен из командной строки оболочки с именем файла в качестве аргумента или без такового. В первом случае открывается

файл, если он существует, или создается новый в каталоге, определенном в пути к нему. Например, команда

```
$ vim ~/mytext/newtext.txt
```

создаст новый текстовый файл в подкаталоге ~/mytetxt домашнего каталога пользователя. Конечно, при этом каталог ~/mytetxt должен уже существовать, иначе при записи файла последует сообщение об ошибке.

Команда `vim` без имени файла откроет редактор Vim и выведет заставку следующего содержания (рис. 18.3):

```
                    VIM - Vi IMproved
~
~                           version 6.1b BETA
~                        by Bram Moolenaar et al.
~              Vim is open source and freely distributable
~
~                     Help poor children in Uganda!
~             type  :help iccf       for information
~
~             type  :q               to exit
~             type  :help  or    for on-line help
~             type  :help version6   for version info
```

Рис. 18.3. Заставка

Она сразу подсказывает направление дальнейших действий: можно либо немедленно начать работу, либо получить справку по использованию редактора.

Настоятельно рекомендую воспользоваться последней возможностью, так как работа в Vim окажется весьма непривычной для пользователя, привыкшего к текстовым редакторам DOS/Windows. В частности, попытка немедленно начать ввод текста успехом не увенчается. И потому следует сначала ознакомиться с режимами работы vi.

В vi существуют три принципиально различных режима работы:

- командный, или визуальный (visual command mode), именуемый также нормальным;
- режим ввода (edit mode);
- ex-режим, или режим построчного редактирования (ex mode или colon mode), который по-русски именуется еще и режимом ввода команд.

Кроме того, в Vim (но не в классическом vi) дополнительно выделяется еще режим визуального выделения, или выбора.

Уже само по себе изобилие режимов способно запутать неподготовленного пользователя. А без четкого понимания различий между режимами никакая, даже самая элементарная, работа в vi попросту невозможна: широкое хождение имеют легенды о пользователях, полагающих единственным способом выхода из этого редактора перезапуск машины. И разнобой переводной русскоязычной терминологии только усугубляет дело. Однако если рассмотреть эти режимы последовательно, все оказывается не так страшно.

Начнем с того режима, который включается по умолчанию при загрузке vi (далее я буду именовать его нормальным — именно он обеспечивает быстроту перемещения по тексту и его обработки). Из него осуществляется переход во все остальные режимы и возврат из них. В нормальном режиме нажатия клавиш не приводят к вводу символов, а интерпретируются как внутренние команды навигации и редактирования, привязанные к различным алфавитно-цифровым или символьным клавишам и их комбинациям. Например, нажатие клавиши <h> вызывает перемещение курсора на один символ влево, клавиши <l> — на один символ вправо, <k> — на строку вверх, <j> — на строку вниз).

Предопределенные клавиши нормального режима чувствительны к регистру (и это имеет глубокий смысл) и последовательности нажатий. Так, повторное нажатие клавиши <d>+<d> отнюдь не эквивалентно по действию двум ее одиночным нажатиям. Кроме того, они чувствительны и к раскладке клавиатуры. В частности, при включении кириллической раскладки никакие клавиши нормального режима при настройках по умолчанию просто не оказывают никакого действия. Впрочем, методы борьбы с этим существуют, о чем я расскажу в заключение раздела.

Естественно, создание текста в командном режиме невозможно. Для этого следует перейти во второй режим (будем называть его режимом ввода), для чего служат разнообразные команды, например, `a` (от append) и `i` (от insert). Здесь нажатия клавиш приводят к вводу обычных буквенно-цифровых символов (после текущей позиции курсора в случае первой команды и перед ней — в случае второй), позволяя создавать новый текст или редактировать имеющийся. Хотя последнее более эффективно в командном режиме, возврат в который осуществляется клавишей <Esc>.

Для операций с документами (то есть файлами) предназначен третий режим, за которым резонно закрепить название командного. Он вызывается из нормального режима командой `:` (символ двоеточия). Вводимые после этого последовательности символов (или отдельные символы) воспринимаются уже как команды, не привязанные к фиксированным клавишам, но опознаваемые по своим именам (или их аббревиатурам).

Здесь возможны следующие действия:

- открытие существующего файла (`:edit имя_файла`); если какой-либо файл перед этим уже загружен, он будет закрыт и замещен новым; если же изменения в нем не были сохранены — последует сообщение об ошибке;
- вставка существующего файла в позицию курсора (`:read имя_файла`);
- запись файла (`:write`), в том числе под другим именем (`:write имя_файла`);
- выход из редактора (`:quit`), происходящий, если текущий файл не изменялся или был предварительно сохранен;
- выход из редактора с предварительным сохранением измененного файла (`:xit` — от exit).

Одна из возможных причин путаницы нормального и командного режима — в командном допустимы любые аббревиатуры указанных команд, вплоть до односимвольных:

- `:e` — для открытия файла;
- `:r` — для его вставки;
- `:w` — для записи;
- `:q` и `:x` — для выхода;
- `:wq` — выход с предварительным сохранением измененного файла (что аналогично команде `:x`).

Команды отправляются на исполнение нажатием клавиши <Enter>, после чего происходит возврат в нормальный режим. Однако попытка, например, закрыть редактор без сохранения изменений в редактируемом документе (командой `:q`) или загрузить новый файл (командой `:e`), не сохранив предыдущий, вызовет сообщение об ошибке. Для принудительного выполнения таких действий команды `:q` и `:e` должны сопровождаться символом `!` без пробела. Например, команда

```
:q!
```

закроет редактор vi, не сохранив изменений в текущем файле.

Действия командного режима частично дублируются в режиме нормальном. Так, в последнем для закрытия же файла (с предварительным сохранением изменений) используется комбинация <Z>+<Z> (двойное нажатие клавиши <Z> — регистр важен!), что является эквивалентом сочетания `:wq` командного режима.

Переход в режим выбора (как обычно, из нормального режима) осуществляется нажатием клавиш <v> или <V>. В обоих случаях нажатия клавиш управления курсором приводят не к его перемещению, а к выделению бло-

ка, в первом — начиная с текущего положения курсора в пределах строки, во втором — начиная с начала маркированной курсором строки. Повторное нажатие тех же клавиш вызывает возврат в нормальный режим. Однако если нажать какую-либо клавишу, обеспечивающую действие в нормальном режиме, оно будет распространено на весь выделенный блок. Ну, и после выполнения этого действия, естественно, происходит возврат в нормальный режим.

В современных версиях Vim предусмотрено нечто вроде индикации режимов, при включении режима ввода в нижней части экрана появляется надпись:

```
-- INSERT --
```

и при переходе в режим выбора — надпись:

```
-- VISUAL --
```

Командный режим включается только по наборе символа : (предполагается, что пользователь этого не забудет сразу же), а нормальный режим, по замыслу разработчиков, потому и нормальный, что в индикации не нуждается. Впрочем, находясь в нормальном режиме, легко убедиться, что это он и есть. Для этого достаточно нажать клавишу <Esc> и услышать звуковой сигнал.

Такая система работы может показаться запутанной начинающему пользователю. Однако она имеет глубокое внутреннее обоснование. Редактор vi создавался изначально как кроссплатформенный, который обязан работать на любых типах реальных и виртуальных терминалов. И потому все действия в нем можно осуществить, не покидая основной, алфавитно-цифровой, части клавиатуры, без обращения к дополнительным клавишам.

Это, с одной стороны, обеспечивает быстроту и эффективность работы (правда, только при наличии доведенных до автоматизма навыков). С другой стороны, внутренние команды навигации и редактирования всегда будут интерпретироваться идентично, независимо от типа терминала и его настроек.

Поскольку нормальный режим vi является не только основным, но и наиболее характерным для этого редактора (и непривычным для пользователя!), имеет смысл ознакомиться с ним подробнее. Все изобилие команд vi (а их насчитывается много десятков) можно разделить на три группы:

- команды навигации;
- команды редактирования;
- команды перехода (в режим ввода).

Команды навигации служат для перемещения курсора по тексту. В консольном режиме для этого могут быть использованы и обычные клавиши управления курсором. Однако уже в различных программах эмуляции терминала

в Иксах их поведение неоднозначно (и зависит от настроек). К тому же внутренние команды дают больше возможностей для навигации по тексту, чем клавиши стандартных клавиатур.

Вообще, для многих команд vi характерно наличие парных эквивалентов — в нижнем и верхнем регистрах одной клавиши (<w> и <W>, <e> и <E>); действие второй команды из пары как бы расширяет действие первой.

Нажатием одной клавиши или, в крайнем случае, двухклавишной комбинации (<Ctrl>+<f> — на следующую экранную страницы, <Ctrl>+<b> — на предыдущую) можно переместиться в абсолютно любое заранее определенное место текущего документа.

В дополнение к этому, команды навигации vi могут использоваться с численными аргументами. Например, команда

```
5h
```

переместит курсор на 5 символов влево (считая символ в позиции курсора), а команда

```
3k
```

на три строки вверх.

Далее, для навигации по тексту могут использоваться символы: минус для перемещения на одну строку вверх и плюс для перемещения на одну строку вниз. Особенно эффективны они в сочетании с численными аргументами: командой

```
7+
```

возможно перемещение на седьмую строку вперед, а командой

```
13-
```

на тринадцатую строку назад (в обоих случаях включая строку в позиции курсора).

Команды редактирования предназначены для модификации существующего текста без перехода в режим ввода. Конечно, и в последнем возможно удаление и замена символов стандартными клавишами, но, как и в случае с навигацией, возможности командного режима в этом отношении много шире.

Так, наряду с удалением единичного символа в позиции курсора `x` или перед ней `X`, возможно удаление слова `dw`, строки `dd` или ее части перед (`dD`) или за (`dG`) курсором, предложения `d)` или абзаца `d}`.

Как и навигационные команды, команды редактирования могут использоваться с численными аргументами. Так, команда `5dd` удалит текущую строку и еще четыре строки вниз. А с помощью команды `3dw` можно удалить три слова подряд (включая то, на котором находится курсор).

Действие ошибочно введенных команд редактирования может быть отменено командой `u` (от undo). Вторичный ввод этой команды приведет к отмене предыдущего действия и т. д. Для возврата ошибочно отмененной операции используется команда <Ctrl>+<R>.

Описание всех команд редактирования заняло бы слишком много места. Подчеркну лишь, что, как и в случае с навигацией, нажатием одной-двух клавиш можно удалить, скопировать, вставить и переместить текстовый блок практически любого размера — от единичного символа до произвольного их количества (строки, предложения, абзаца, экранной страницы).

Команды перехода могут рассматриваться как подмножество команд редактирования нормального режима, после которых возможен ввод новых символов и их последовательностей. Кроме уже упомянутых `a` и `i` (ввод после курсора и в его позиции соответственно), к ним относятся:

- `A` — ввод текста в конец строки;
- `I` — ввод в начало строки;
- `o` — создание новой строки под текущей с возможностью ввода в нее текста;
- `O` — создание новой строки над текущей, в которую также можно ввести текст.

Как и большинство прочих команд редактора vi, команды перехода могут использоваться с числовыми аргументами. Так, если дать команду `3a` и после этого ввести некоторую последовательность символов, по выходе в командный режим (клавишей <Esc>) она будет повторена трижды.

Редактор vi располагает средствами поиска и замены текстовых фрагментов, в том числе и с использованием регулярных выражений. Для этого предназначена команда командного режима `:s` (от substitute). Она дается в форме

```
:#s/text1/text2/опция
```

где `#` — количество строк, в которых операции поиска и замены должны осуществляться (без указания его действие команды распространяется только на текущую строку). Если поиск и замену необходимо выполнить по всему тексту документа, дается команда

```
:%s.
```

Следует подчеркнуть, что если не указать заменяющего текста (и опций замены), все текстовые фрагменты, указанные в качестве заменяемого текста, будут просто удалены. Во избежание этого используются опции команды `:s`. Так, опция `/c` предписывает запрос подтверждения на замену для каждого найденного фрагмента.

И поиск, и замена в vi возможна только для последовательности символов, составляющих одну строку. Заменяющая последовательность символов также должна образовывать единую строку.

Описанным не исчерпываются возможности редактора vi. В частности, он (вернее, его клон Vim) поддерживает язык макрокоманд и допускает их протоколирование. В комплекте с редактором (в каталоге /usr/local/share/vim/vim###) имеется большое количество таких макросов. Там же и детальная документация по их применению.

Функциональность редактора Vim в значительной мере зависит от его настроек. Пример конфигурационного файла (vimrc_example.vim) можно обнаружить в том же каталоге (/usr/local/share/vim/vim###). Его следует скопировать в свой домашний каталог под именем ~/.vimrc и отредактировать по потребностям.

В частности, именно в этом файле можно обеспечить работу клавиш нормального режима при кириллической раскладке клавиатуры. Для этого в него вносится строка

```
set langmap=
```

и перечисляются все символы кириллической раскладки (и в верхнем, и в нижнем регистрах) попарно с соответствующими символами раскладки латинской, разделяя пары запятой без пробела:

```
set langmap=ё,Ё~,йq, ... Б<,ю.,Ю>
```

Правда, чтобы это сработало, Vim должен быть собран с опцией `big`, задаваемой в командной строке при исполнении сценария `./configure`.

А вообще редактированием файла ~/.vimrc Vim может быть адаптирован для задач любого рода, например, работы с HTML-документами. На сайте **http://www.vim.org** имеется большое количество примеров конфигурационных файлов такого рода.

Joe: гармония простоты и функциональности

Редактор joe принадлежит к категории public domain, то есть общественного достояния: мало кто помнит, кто и когда написал его первую версию. И долгое время он пребывал в состоянии практической консервации. Пока, наконец, в этом проекте не наметилась определенная активность, ознаменовавшаяся выходом версий серии 3.X. Они привнесли несколько кардинальных новшеств, сделавших исходно хороший редактор еще лучше...

Обзор возможностей

Текстовый редактор joe — типичный представитель консольных редакторов командного стиля, ориентированных не на действие через меню, а на

управление с помощью прямых директив. Название его можно перевести примерно как "редактор дядюшки Джо". Он создан Джозефом Алленом (Joseph H. Allen) при участии Ларри Форда (Larry Foard) и Гари Грея (Gary Gray). Это открытая и бесплатная программа, доступная в исходниках на сайте проекта (**http://sourceforge.net/projects/joe-editor/**). Она реализована для всех, насколько я знаю, POSIX-совместимых систем. А некоторые конкретные версии были собраны для Windows всякого рода и даже для DOS.

Запускается joe одноименной командой, можно с именем файла, предназначенного для редактирования. В случае, если этого имени в природе не имеется, создается новый файл. Кроме этого, при запуске joe можно использовать ряд опций командной строки. Представление о них дает чтение страниц экранной документации.

Сразу после запуска joe выглядит весьма непрезентабельно: черный экран с белым текстом — и все. Что делать дальше — остается пока неясным. Единственно, строка заголовка в верхней части экрана гласит, что с помощью комбинации клавиш <Ctrl>+<K>+<H> можно вывести на экран систему помощи. И далее пролистывать ее с помощью <Esc>+<.> (точка) — вперед или <Esc>+<,> (запятая) — назад.

Так вот, внимательное знакомство с системой помощи дает представление о возможностях редактора. Они неожиданно оказываются весьма богатыми.

Думаю, понятно, что текстовый редактор позволяет вводить текст (в том числе и кириллический) и обеспечивает навигацию по нему, а также редактирование. Последняя осуществляется двояко: либо с помощью собственных клавишных комбинаций, о чем подробнее в следующем разделе, либо стандартных клавиш управления курсором. При этом клавиши эти введут себя обычным (с точки зрения пользователя DOS/Windows) образом, что отнюдь не само собой разумеющееся для консольных UNIX-редакторов.

Вообще, следует заметить, что основным средством навигации по тексту в joe являются именно собственные управляющие последовательности, а не клавиши управления курсором. Во-первых, при наличии некоторого навыка они обеспечивают большую скорость работы за счет того, что не требуется перемещения пальцев за пределы основной клавиатуры (поверьте, это действительно быстрее!). Во-вторых, как неоднократно подчеркивалось и ранее, команды joe, вызываемые клавишными комбинациями, функционируют абсолютно одинаково на терминалах любых типов.

Мышь в joe поддерживается стандартным для UNIX-консоли образом. То есть она выступает не как указательное устройство, а как средство выделения и копирования текстовых блоков. Это относится как к текстовой консоли, так и к эмуляторам терминалов графического режима.

Разумеется, в соответствие со своим званием joe позволяет и редактирование текстов: выделение блоков и отдельных знаков, их копирование, перемеще-

ние и удаление, форматирование абзацев (центрирование, выравнивание и т. д.), вставку существующих файлов в текущий документ и запись выделенных фрагментов в виде отдельных файлов.

Редактор joe имеет функцию многоуровневой отмены и возврата отмененных операций. Встроенной проверки орфографии нет, однако можно подключить внешнюю программу (такую как ispell), в том числе и для русскоязычных текстов. Имеются достаточно развитые средства поиска и замены, в том числе с использованием шаблонов и регулярных выражений. Есть возможность создания закладок (Bookmarks) и перехода к ним, что необходимо при редактировании длинных структурированных документов.

В joe имитируется многооконный режим: поле текущего документа может быть разбито пополам, и далее каждое из них также может делиться сколь угодно дробно (правда, только по горизонтали). Обеспечена также одновременная работа со многими документами, каждый из которых может быть выведен в оконном или полноэкранном виде. Количество одновременно открытых файлов не ограничено ничем, даже, по некоторым сведениям, системной памятью — joe позволяет работать с документами, размер которых превышает объем всей доступной (то есть физической и виртуальной) памяти. Правда, проверить это на современных машинах несколько затруднительно...

В joe поддерживается собственный макроязык с достаточно прозрачным синтаксисом. Кроме того, имеется режим протоколирования макросов, что позволяет быстро наращивать его возможности.

И вообще, joe — очень настраиваемый редактор. Во-первых, имеется система интерактивной настройки ряда параметров, таких как перенос слов, абзацный отступ и т. д. Правда, установки эти действуют только в текущем сеансе. Для перманентных изменений необходимо редактирование конфигурационного файла. Однако здесь, учитывая возможность встраивания макрокоманд, возможности настроек становятся поистине безграничными.

И наконец, что немаловажно в наших условиях, joe корректно работает с кириллицей. При правильно русифицированной консоли не возникает никаких проблем ни с выводом, ни с вводом символов кириллицы (в том числе, в последних версиях, и в кодировке UTF8). Более того, все клавишные комбинации работают и при латинской, и при русской раскладке клавиатуры. Правда, в последнем случае обычно требуется дополнительное нажатие управляющей клавиши.

Иными словами, joe — вполне развитый и полнофункциональный текстовый редактор общего назначения. Он в равной мере пригоден для эпизодической работы по написанию скриптов, правке конфигурационных файлов и для систематического использования при создании длинных структурированных текстов. А наличие языка макрокоманд допускает эффективно при-

менять его и в специальных целях — для разметки HTML-страниц, верстки документов в TeX, не говоря уже о собственно программировании.

Система помощи

Ознакомиться практически со всеми возможностями редактора joe можно посредством его системы помощи. Как уже говорилось, она выводится на экран нажатием комбинации клавиш <Ctrl>+<K>+<H> и насчитывает шесть секций, каждая из которых занимает собственный экран.

- Первая секция `Basic` описывает действия наиболее общего плана:
  - перемещения курсора (субсекция `CURSOR`);
  - переходы по тексту (субсекция `GO TO`);
  - операции с текстовыми блоками (субсекция `BLOCK`);
  - команды удаления символов и текстовых фрагментов (субсекция `DELETE`);
  - команды поиска;
  - проверки орфографии;
  - форматирования (субсекции `SEARCH`, `SPELL`, `MISC`);
  - операции с файлами (субсекция `FILE`).
- Вторая секция посвящена описанию манипуляций с окнами — расщеплению (`split`) экрана, скрытию и показу открытых окон, переходу между окнами, изменению их размера.
- В третьей секции собрано описание расширенных возможностей для редактирования текстов, как-то:
  - работа по протоколированию, записи и воспроизведению макросов (субсекция `MACROS`);
  - команды для прокрутки текста (субсекция `SCROLL`);
  - команды для взаимодействия с оболочкой (субсекция `SHELL`);
  - средства установки закладок, наращиваемого поиска, ввода спецсимволов (секции `BOOKMARKS`, `I-SEARCH`, `QUOTE`) и т. д.
- Четвертая секция — расширенные возможности для программистов (команды перехода к регулярным выражениям, компилирования и отладки).
- В пятой секции дано описание сложных регулярных выражений.
- И, наконец, шестая секция — операции с командной строкой встроенной в редактор командной оболочки.

Таковы возможности справочной системы joe по умолчанию. Как будет показано позже, пользователь может не только изменять имеющиеся секции, но и создавать собственные, в любом количестве и для любых целей.

Если мне удалось убедить читателя, что joe — вещь стоящая, имеет смысл подробнее рассмотреть его.

Характерные особенности

Для эффективного использования joe следует четко уяснить, что это командный редактор в чистом виде. То есть все действия по редактированию текста осуществляются соответствующими встроенными командами, к которым привязаны комбинации клавиш. В сущности, это макросы на собственном языке joe. Из чего следует, что, с одной стороны, система команд может быть сколь угодно наращена, с другой — клавишные комбинации для них могут быть переопределены произвольным образом.

Впрочем, необходимости в последнем я не вижу. Структура предопределенных по умолчанию клавишных комбинаций проста и очень логична. За самыми простыми и частыми действиями для навигации и редактирования закреплены двухклавишные комбинации. Это, как правило, сочетание одновременно нажатых клавиши <Ctrl> и буквенной. Последняя имеет, обычно, мнемонический смысл, хотя и не всегда прозрачный.

Полный список встроенных команд и привязанных к ним клавишных комбинаций дан в заключительном разделе этой статьи. Здесь же я приведу только основные примеры. Так, комбинация <Ctrl>+<B> (от backward) перемещает курсор на один знак влево, <Ctrl>+<F> (от forward) — на один знак вправо, <Ctrl>+<Z> — переход к предыдущему слову, <Ctrl>+<X> — к последующему слову и т. д.

В некоторых случаях в качестве управляющей клавиши используется клавиша <Meta>: если в комбинации с ней набрать литеру W, курсор переместится на строку вверх, литеру Z — на строку вниз. Кроме того, <Meta> служит для вызова проверки правописания для слова (<Meta>+<N>) и всего файла (<Meta>+<L>). Нажатие клавиши <Meta> два раза подряд приводит к установке закладки (bookmark), которая маркируется произвольной цифрой, а комбинация <Meta>+ эта самая цифра вызывает переход к установленной закладке. Правда, очевидно, что закладок не может быть больше 10; и к тому же, по завершении сеанса они не сохраняются.

Все клавишные комбинации в joe не чувствительны к регистру, причем не только для буквенных, но и символьных клавиш. Кроме того, все двухклавишные комбинации не чувствительны и к раскладке клавиатуры. Для более сложных или редких действий используются трехклавишные комбинации. Это почти исключительно одновременно нажатые <Ctrl>+<K>, после чего

нажимается литерная клавиша. Трехклавишные комбинации также не чувствительны к регистру. И работают при кириллической раскладке клавиатуры. В этом случае только необходимо нажимать вторую литерную клавишу вместе с той же клавишей <Ctrl>.

В joe нет отдельной функции для переименования файла. Но при любой записи текущего документа следует запрос на подтверждение имени файла, изменить которое при этом никто не запрещает. Следует только помнить, что дальнейшая работа происходит с исходным, а не переименованным файлом.

Кроме того, в joe доступны еще некоторые действия с файлами. Так, комбинация <Ctrl>+<K>+<R> вставляет текст из существующего файла в позицию курсора, <Ctrl>+<K>+<W> записывает выделенный блок в виде нового файла (разумеется, запросив предварительно его имя). С помощью комбинации <Ctrl>+<K>+<E> можно открыть для редактирования другой существующий файл. При этом следует предложение ввести путь и имя, причем и для того, и для другого работает режим дополнения клавишей табуляции, как в командных средах bash или tcsh.

Между одновременно открытыми файлами возможен обмен данными: или с помощью команд joe (то есть выделением блока в первом файле и его копированием или перемещением во второй), или с помощью мыши — стандартным выделением и вставкой с помощью средней ее клавиши. Второй способ, естественно, может применяться и для обмена между разными копиями joe, запущенными на отдельных виртуальных консолях.

Одновременно открытые файлы могут быть представлены как в полноэкранном виде, так и каждый в своем окне. Для переключения между однооконным и многооконным режимами служит комбинация <Ctrl>+<K>+<I>. Размер каждого из выведенных окон может быть увеличен (<Ctrl>+<K>+ +<G>) или уменьшен (<Ctrl>+<K>+<T>), правда, только с шагом в одну экранную строку. Переключение между открытыми документами, вне зависимости от режима, осуществляется комбинациями <Ctrl>+<K>+<N> (вперед или вниз) и <Ctrl>+<K>+<P> (назад или вверх).

К слову сказать, в joe возможен и независимый просмотр разных частей документа в отдельных окнах, для чего предназначена функция расщепления окна <Ctrl>+<K>+<O>. Ну и, конечно же, фрагменты из одной части файла могут быть легко перенесены в другую.

Универсальной комбинацией для окончания любой операции в joe является <Ctrl>+<C>. С ее помощью закрывается окно с текущим документом; если он был единственным в данном сеансе, одновременно происходит и выход из редактора. В обоих случаях следует запрос на сохранение изменений, буде таковые имелись. Отказаться от выхода или закрытия файла можно повторным нажатием той же комбинации <Ctrl>+<C>. Она же используется

для прекращения любой длящейся во времени (спеллинг, поиск) или требующей подтверждения операции.

Кроме этого, непосредственно из joe, без выхода, можно обращаться к командам оболочки (shell), причем различными способами. Так, комбинация <Ctrl>+<K>+<Z> обеспечивает временный выход в оболочку, где можно вводить любые ее команды. А по завершении операций вернуться в среду joe можно командой fg. То есть в данном случае мы имеем дело с обычной приостановкой текущей задачи.

Кроме этого, есть и более интересная возможность: открытие внутри joe, посредством комбинации <Ctrl>+<K>+<'> (апостроф), самостоятельного окна с собственной командной оболочкой. Здесь можно выполнять любые команды с выводом их результатов на экран. После чего стандартной командой exit осуществляется выход из среды, а все результаты сохраняются обычным для joe образом в виде текстового файла: возможность неоценимая при создании и редактировании всякого рода скриптов.

Следует подчеркнуть, что в данном случае запускается именно встроенная в joe оболочка, не имеющая никакого отношения к пользовательской (login shell). Она не порождает нового процесса (в чем легко убедиться командой `ps`) и не наследует никаких переменных окружения от пользовательской оболочки. Интерактивные ее возможности достаточно слабы (не поддерживаются ни автодополнение, ни история команд). Однако она дает возможность интеграции возможностей joe по редактированию и команд оболочки для обработки текста.

Единичная команда оболочки может быть выполнена после нажатия комбинации клавиш <Meta>+<!>. В этом случае внизу экрана появляется приглашение командной строки в форме:

```
Program to run:
```

как разновидность встроенной оболочки joe.

Наконец, в joe обеспечивается ввод специальных символов. Если же вместо кода нажать <Esc>, можно ввести любую esc-последовательность. То же самое можно сделать и после нажатия комбинации <Ctrl>+<\>.

Специальные символы могут вводиться не только непосредственно в редактируемом тексте, но и в строке поиска. Это позволяет легко находить и глобально заменять лишние разрывы строк и тому подобные нарушения структуры.

Таковы основные возможности joe для редактирования текстов общего характера. Кроме этого, имеется ряд команд специального назначения, используемых при программировании (поиск ошибок, компилирование и пр.). Однако и сказанного достаточно для представления возможностей редактора joe.

Макрокоманды

Если же штатных команд редактора joe оказывается недостаточно, можно прибегнуть к их самостоятельному конструированию. Для этого предназначен внутренний язык макрокоманд. К сожалению, он не описан в экранной документации. Однако получить представление о его синтаксисе и возможностях можно, изучив внимательно конфигурационный файл joerc (в следующем разделе).

Кроме того, есть еще один простой и эффективный способ изучения макроязыка, совмещающий приятное с полезным — режим протоколирования макрокоманд. Включается он комбинацией клавиш <Ctrl>+<K>+<[> (открывающая квадратная скобка), вслед за чем следует ввести номер макроса (от 0 до 9), выступающий как в качестве его имени, так и в роли запускающей клавиши. Далее просто выполняются необходимые действия (например, вводится требуемый тег HTML), после чего запись макроса останавливается комбинацией <Ctrl>+<K>+<]> (закрывающая квадратная скобка). Для воспроизведения запротоколированного макроса используется комбинация <Ctrl>+<K>+<#> (где # — указанный при записи номер макрокоманды).

С помощью протоколирования макросов можно автоматизировать ввод наиболее нужных для конкретной задачи символов и их наборов, не предусмотренных штатным образом. Например, основных тегов HTML для разметки Web-страниц, таких как параграф, разрыв строки, заголовки нескольких уровней, таблицы и списки. А заодно и изучить синтаксис языка. Поскольку в joe предусмотрена возможность помещения запротоколированных в данном сеансе макросов в тело существующего или нового документа (комбинацией клавиш <Meta>+<D>).

Например, записанные мной макросы для ввода HTML-тегов выглядят следующим образом:

```
General Structure
insf,"~/blank.html",rtn  ^K .k1    HTML Page
"",ltarw,...,ltarw       .k1     H1
rtn,"",ltarw,...,ltarw       .k2     H2
rtn,"\09","",ltarw,...,ltarw        .k3     H3
rtn,"\09","\09","",ltarw,...,ltarw .k4     H4

Body Text
rtn,"<p>","</p>",ltarw,...,ltarw    .k5     Paragraph
rtn,"<br>"      .k6  Break
rtn,"<pre>",rtn,"</pre>",uparw  .k7     Preformat
"<strong></strong>",ltarw,...,,ltarw .k8     Strong
```

```
"<em></em>",ltarw,...,ltarw       .k9     Emphasis
rtn,"<div>",rtn,"</div>",uparw  ^K .k8  Division

Lists and Tables
rtn,rtn,"<ul>",rtn,"</ul>",uparw    ^K .k2  Unordered List
rtn,"<ol>",rtn,"</ol>",uparw    ^K .k3  Ordered List
"<li>,</li>",ltarw,...,ltarw       ^K .k4  List Item
rtn,"",rtn,"<table></table>",uparw      ^K .k5  Table
```

Из этого вполне можно составить представление о командах языка: в первой колонке следуют разделяемые запятыми (без пробелов!) зарезервированные команды и вводимые символьные значения в парных кавычках. Далее идут разделенные табулятором клавишные комбинации, закрепленные за макросами, и имена макросов. Последние в оригинале представлены в виде `Macro 0`, `Macro 1` и т. д. Но никто не запрещает при редактировании придать им осмысленные имена. Да и сами макросы могут быть отредактированы должным образом в текстовом редакторе (joe, например).

Легко сообразить, что за один прием можно запротоколировать не более 10 макросов (маркированных цифрами от 0 до 9). Более того, они будут действовать только в течение данного сеанса: по выходе из редактора записанные макросы сами собой не сохраняются. Что, казалось бы, обесценивает данную возможность.

К счастью, это не так. И раз запротоколированные макрокоманды можно сохранить для дальнейшего использования. Да и количество их не обязано ограничиваться десятью. Как этого добиться?

Выясняется, что за воспроизведение макросов отвечает конфигурационный файл ~/.joerc, о котором подробнее будет сказано в следующем разделе.

Назначенные по умолчанию клавишные комбинации не являются обязательными. И их можно вручную заменить на любые другие, из числа не использованных ранее. После чего можно начать протоколирование команд сначала, нумеруя их от 0 до 9. Затем повторить процедуру встраивания их ~/.joerc. А поскольку количество незадействованных в joe клавишных комбинаций очень велико, то реально число созданных пользователем макрокоманд ограничивается только его фантазией и потребностями.

Таким образом можно легко автоматизировать процесс ввода тегов HTML или XML, конструкций JavaScript, скриптов командной среды, разметки документов TeX, а также все, что потребуется впредь. Превратив joe в специализированный инструмент для решения почти любых задач.

Настройка joe

Как я уже говорил, некоторые настройки joe можно выполнить интерактивно. Однако они весьма ограничены и к тому же будут иметь силу только в текущем сеансе. Более интересные и богатые возможности открываются при редактировании конфигурационного файла joerc.

Пример такого файла обнаруживается в каталоге /usr/local/etc/joerc. Его надлежит скопировать с свой домашний каталог и переименовать в ~/.joerc. А затем он открывается в любом текстовом редакторе, лучше всего в самом же joe и, после соответствующего изучения, правится вручную.

В экранной документации синтаксис конфигурационного файла не описан, что мотивируется его простотой и понятностью. Это действительно почти так, смысл практически всех настроек можно понять из контекста (и комментариев).

Файл .joerc разбит на четыре секции. Первая — это глобальные опции редактора, большая часть которых может быть задана также параметрами командной строки. Опция является установленной (или, напротив, отмененной), если ею начинается строка (первая колонка, в терминологии программы — это относится и ко всем остальным секциям .joerc). Если строка начинается с пробела или табуляции, дальнейшее ее содержание рассматривается как комментарий и не учитывается. Комментарием является и все, отделенное пробелом от имени опции.

Описание всех опций заняло бы слишком много места, поэтому обращу внимание только на некоторые ключевые моменты. Обязательно следует включить (то есть удалить пробел в начале строки, если он имеет место быть) опцию `-asis`.

Это необходимо для правильного отображения символов кириллицы, иначе они будут показаны латинской транслитерацией.

Полезным представляется также установка опций:

- `-lightoff`, обеспечивающей выключение подсветки выделенного блока после его перемещения или копирования, иначе выделение это будет постоянно маячить перед глазами;
- `-marking`, дающей подсветку текста между началом выделяемого блока и текущей позицией курсора.

Можно отменить также создание страховых копий или, напротив, определить место для их помещения, отличное от исходных файлов. Это достигается включением опций `-nobackups` или `-backpath path`, соответственно.

Интересная возможность — задание количества строк и колонок (знаков в строке) на экране, отличное от определенных для текущего терминала (виртуальной консоли) в целом, что задается опциями

```
-lines #
-columns #
```

где # — количество строк и знаков соответственно.

В этой же секции настраивается вид статусной строки (вывод которой, впрочем, можно и отключить опцией `-nosta`). Он определяется опциями `-lmsg` и `-rmsg`. Первая определяет компоненты, выводимые в левой части строки, вторая — в правой. В любой из них можно вывести индикацию режимов (забивки или вставки, переноса слов, автоотступов), имя файла и показатель его изменения, текущее положение курсора (в строках, колонках или знаках). Первый знак левой части статусной строки определяет ее общий вид: инвертирование цветов, выделение подчеркнутым или полужирным начертанием, мерцание.

Например, строка вида

```
-lmsg \i%k%T%W%I%X %n %m%R %M
```

указывает, что в левой части статусной строки в инвертированном виде (черным по белому, `\i`) должны быть выведены:

- префиксные ключи (`%k`), маркирующие включение режимов вставки/замены (`%T`), переноса слов (`%W`), автоматического отступа (`%I`), прямоугольного выделения (`%X`);
- имя редактируемого файла (`%n`);
- указание на модификацию файла (`%m`) и на режим "только для чтения" (`%R`);
- индикатор включения протоколирования макросов (`%M`).

Строка же вида

```
-rmsg  Row %r Col %c %o %O %u
```

выводит в правой части статусной линии номер текущей строки (`Row %r`) и колонки (`Col %c`) файла, смещение от начала в байтах (в десятичной, `%o`, и шестнадцатеричной, `%O`, формах, и системное время в 24-часовом формате `%u`).

Возможен вывод и иной информации, как то:

- системного времени в 12-часовом формате (`%t`);
- индикации измененного файла символом `*`;
- ASCII-кода символа под курсором в десятичной (`%a`) или шестнадцатеричной (`%A`) форме;
- процента просмотра файла до позиции курсора (`%p`);

- общего количества строк в файле (`%l`);
- индикации запуска встроенной оболочки (`%S`).

Кроме того, внешний вид статусной строки можно изменить, придав ему атрибуты подчеркивания (`\u`), полужирного начертания (`\b`), мерцания (`\f`).

Вторая секция конфигурационного файла — это локальные опции, которые можно определить отдельно для файлов различных типов. Для этого в ее составе создаются субсекции вида:

```
*       все файлы
*.html  документы HTML
*.c     программы на C
*rc     конфигурационные файлы
```

и т. д. Напомню здесь, что знак маски (`*`) должен обязательно начинаться с начала строки, то есть занимать первую колонку.

Для каждой субсекции можно независимо задать такие параметры, как перенос слов (или его отключение), автоматический отступ, величину табуляции и т. д., а также, что интересно, назначить собственную раскладку клавиатуры, отличную от определенной для консоли в целом. Кроме того, с файлами любого типа можно связать макросы, выполняемые при их загрузке или записи.

Третья секция описывает вид экранов помощи, выводимых клавишной комбинацией <Ctrl>+<K>+<H>. Экраны эти могут быть изменены как с позиций внешнего вида (инверсия цветов, выделение или подчеркивание и т. д.), так и по существу. В частности, здесь можно задать специальные экраны помощи для собственных макрокоманд. Более того, редактированием этой секции можно выводить и подсказки на русском (или каком-либо еще) языке. В частности, для макросов ввода тегов HTML, описанных в одном из предыдущих разделов, можно создать экран помощи вида:

```
{HTML
\i   Help Screen    turn off with ^KH    more help with ESC . (^[.)
\i \i\uGeneral\u     \uBody Text\u                \uList&Table\u
\u
\i \i^KF1 Blank  F5 P  F7 Pre ^KF8 Div  ^KF2 UL ^KF3 OL ^KF4 LI  F10 Href
^KF9
\i \iF1-F4 H1-H4 F6 Br F8 Strong F9 Em  ^KF5 Tab ^KF6 TR ^KF7 TD ^KF10
Name
}
```

Каждый такой экран заключается в фигурные скобки, предваряется собственным идентификатором, ему придаются атрибуты внешнего вида, задается

строка подсказки, после чего следует просто перечисление клавишных комбинаций и их описание.

Четвертая секция — разного рода ключевые последовательности, или связки (key bindings), в том числе и макрокоманды. Они могут быть определены отдельно для всех окон (субсекция `:windows`, начиная с первой колонки), окна редактируемого текста (субсекция `:main`) и т. д.

Внимательное рассмотрение секции показывает, что все описанные клавишные комбинации joe представляют собой макрокоманды, именно здесь и определенные. Из чего следует два вывода.

Первый — возможность переопределения клавиатурных комбинаций, назначенных для штатных команд joe по умолчанию. Например, если вам не нравится, что пролистывание экрана осуществляется комбинациями <Ctrl>+ +<U> (назад) и <Ctrl>+<V> (вперед), можно присвоить им иные значения (из числа свободных, разумеется).

Второй вывод: штатные команды joe могут быть (почти) неограниченно наращены собственными макрокомандами. Для чего достаточно их запротоколировать и поместить (напомню, посредством <Esc>+<D>) в соответствующее место четвертой секции (скорее всего, в раздел `:windows` или `:main`) файла ~/joerc. Более того, здесь же им можно присвоить и членораздельные имена (вместо данных по умолчанию `Macro 1`, `Macro 2`) и изменить закрепленные клавишные комбинации. То есть полностью индивидуализировать редактор, не прибегая ни к каким сильнодействующим средствам.

Пользовательские макросы следует помещать в одну из субсекций, `:window` или `:main`. В последней, кроме этого, целесообразно отредактировать имеющиеся там по умолчанию макросы проверки орфографии для работы с русским словарем. По умолчанию эти макросы описываются строками вида:

```
:def spellfile
:def spellword
```

для спеллинга всего текста и отдельных слов соответственно. Они просто описывают параметры вызова внешней программы проверки орфографии (по умолчанию — `ispell`). И подключение русского словаря при этом, разумеется, не предусмотрено. Чтобы осуществить его, следует просто добавить к обеим строкам после вызова исполнимой команды `ispell` опцию `-d russian`, в результате чего строки эти примут вид:

```
:def spellfile filt,"cat >ispell.tmp;ispell -d russian ispell.tmp \
</dev/tty >/dev/tty;cat ispell.tmp;/bin/rm ispell.tmp",rtn,retype
:def spellword psh,nextword,markk,prevword,markb,filt,"cat
\
>ispell.tmp;ispell -d russian ispell.tmp </dev/tty >/dev/tty;tr
-d
\<ispell.tmp '\\012';/bin/rm ispell.tmp",rtn,retype,nextword
```

Символы обратного слэша в конце строк означают только то, что каждая строка на самом деле не должна прерываться; вносить их в тело макроса не следует (нужно только позаботиться, чтобы опция переноса слов в редакторе была отключена).

Дополнительные настройки

В начале своего повествования я упоминал, что ныне (начиная с версии 3.0) в joe появилась возможность подсветки синтаксиса для ряда языков программирования и разметки. Чтобы ей воспользоваться, нужно, во-первых, включить соответствующую опцию в конфигурационном файле ~/.joerc, она расположена в секции `Default local options` и имеет вид `-highlight`.

Далее, требуются файлы описания цветов для синтаксических элементов различных языков. Примеры таких файлов расположены в /usr/local/etc/joe/synatx и охватывают языки Си (c.jsf), Assembler (asm.jsf), Fortran (fortran.jsf) и многие другие. Есть здесь и файлы описания языков командных оболочек (sh.jsf и csh.jsf), diff-файлов (diff.jsf), конфигов (conf.jsf), а также языков разметки (html.jsf и xml.jsf). Единственное, что остается с ними сделать — это отредактировать их в соответствии с предпочтительной цветовой гаммой (и сохранить в собственном домашнем каталоге под именами типа ~/.synatx/html.jsf).

Наконец, последнее — это проверить соответствие файлов описаний в главном конфигурационном файле ~/.joerc. По умолчанию в каждой языковой субсекции они указываются там в виде:

```
-syntax html
```

в предположении их стандартных имен и размещения (в указанных каталогах /usr/local/etc/joe/synatx или ~/.synatx/html.jsf).

Заключительные соображения

И так, чем же может привлечь пользователя редактор joe? На мой взгляд, многим. Разумеется, если он, то есть пользователь, вообще испытывает потребность в консольном редакторе.

Во-первых, в его пользу относительная (по сравнению, скажем, с vim или, тем более, emacs) простота освоения и использования. Немаловажно, что элементарные действия по вводу и редактированию текста в нем могут осуществляться (в том числе и) привычным для выходца из мира DOS/Windows способом.

Второй плюс — схожесть модели выполнения команд, особенно отчетливо проступающая в сравнении с emacs. Модель эта логична и легко запомина-

ется, в том числе и благодаря мнемоническому характеру литерных клавиш, сочетающихся с управляющими.

Если же сравнивать joe с меню-ориентированными редакторами, такими как mcedit или le, то его отличают очень высокая степень настраиваемости (практически не уступающая классическим редакторам командного стиля) и быстрота выполнения основных операций по вводу и редактированию. Впрочем, конечно, быстрота эта (как и во всех командных редакторах) достижима только при наработке определенного минимума практических навыков, желательно, доведенных до рефлекторного уровня.

И, кроме того, имеются альтернативные им, традиционные (для DOS/Windows) приемы навигации по тексту и пр. Весьма удобными представляются средства одновременной работы с большим количеством документов и обмена данными между ними. Возможность независимого просмотра различных частей одного файла в отдельных окнах также следует отнести к числу достоинств (коими обладают не все консольные редакторы).

Очень эффективно применение joe для составления и редактирования пользовательских сценариев командой оболочки, благодаря возможностям временного выхода в командную строку, запуску единичной команды внутри редактора и открытию почти полноценного сеанса командной оболочки изнутри него же (с возможностью записи в виде файла). Судя по документации и конфигурационным файлам, есть средства и для более сложных упражнений, но об их эффективности судить я не компетентен.

Наконец, главное достоинство joe — простота адаптации к специальным задачам. Она вполне достижима элементарным протоколированием макрокоманд и несложным их редактированием. Вероятно, такие возможности покажутся смешными записному программисту, но пользователю они в большинстве случаев более, чем достаточны.

К принципиальным упущениям joe можно отнести, пожалуй, только отсутствие режима переноса символов без образования новой строки: режим word wrapping приводит к разрыву непрерывности абзаца (и в дальнейшем может создать сложности при автоматизированной обработке текста), а его отключение — к неудобству набора.

Иными словами, joe — достаточно мощный и удобный инструмент для работы с текстами. Он может быть использован и при периодических работах (вроде правки конфигурационных файлов или составления небольших документов). Но наиболее эффективно применять joe при повседневной работе с большими и структурированными текстами. Среди всех консольных редакторов joe отличается близким к оптимальному соотношением простоты, мощности и настраиваемости. А посему беру на себя смелость рекомендовать его всем любителям работы в текстовом режиме, буде до сего времени они не приобрели иных пристрастий.

NEdit — текстовый редактор для Иксов

Текстовый редактор NEdit (Nirvana Editor) — единственный, кто по своей функциональности и настраиваемости может состязаться с самыми мощными редакторами консольного режима, такими как vim и emacs. И если титул лучшего редактора для консоли — предмет длительных, чуть ни религиозных дискуссий, то в графическом режиме пальма первенства, по моему мнению, несомненно за NEdit. Хотя последнее время и kate движется в этом направлении семимильными шагами, реализуя некоторые уникальные аспекты интеграции редактора, терминала и файлового визуализатора.

Общая характеристика

Редактор NEdit реализован для всех UNIX-подобных систем, как коммерческих, так и свободных. В числе разработчиков его большой коллектив, включающий Марка Эдела (Mark Edel), Джо Кирьякопулоса (Joy Kyriakopulos), Кристофера Конрада (Christopher Conrad) и многих других. Официальный сайт проекта — **http://www.nedit.org**. На нем всегда можно найти последнюю версию этого редактора, как в виде исходных текстов, так и прекомпилированную для основных операционных систем (rpm, deb и tgz — пакеты для Linux, бинарники для FreeBSD и т. д.).

NEdit базируется на библиотеке Motif, имеющей статус коммерческой. Однако он может быть собран и с использованием свободного аналога этой библиотеки OpenMotif. Впрочем, Motif требуется только для сборки NEdit, и потому есть резон устанавливать его из бинарников, вне зависимости от операционной системы.

По назначению NEdit — текстовый редактор для программистов. Однако, кроме этого, он может быть использован для множества целей, от элементарного набора текстов до разработки Web-страниц и верстки оригинал-макетов в TeX.

В чем величие и мощь этого редактора? Во-первых, он позволяет осуществлять все действия по вводу и редактированию текстов, в том числе весьма сложные операции по поиску и замене фрагментов введенного текста.

Во-вторых, NEdit — редактор, в котором настройке поддается практически все, от гарнитуры экранного шрифта, его цвета и цвета фона до контекстного меню, вызываемого, как обычно, правой кнопкой мыши.

В-третьих, для NEdit характерна практически неограниченная наращиваемость, осуществляемая путем подключения сценариев командной оболочки и пользовательских команд, создаваемых на встроенном языке.

Как и большинство приложений, основанных на библиотеке Motif, NEdit управляется исключительно через меню: инструментальная панель отсутст-

вует, что, однако, компенсируется достаточным количеством комбинаций "горячих клавиш" и, главное, возможностью создавать собственные макрокоманды.

Итак, основными элементами интерфейса редактора NEdit являются: строка заголовка, главное меню, строка состояния **Statistic Line**, строка поиска **Incremental Search Line** (по умолчанию отключены) и рабочее поле.

В строке заголовка, помимо управляющих элементов окна (определяющихся используемым оконным менеджером), указывается имя текущего файла, а также фиксируется, подвергался ли он изменениям со времени последней записи; в этом случае после имени появляется метка (modified).

Основные возможности редактора NEdit становятся ясными, если рассмотреть его главное меню. Оно насчитывает, за исключением непременного **Help**, семь пунктов:

- **File**;
- **Edit**;
- **Search**;
- **Preferences**;
- **Shell**;
- **Macro**;
- **Windows**.

Большинство действий, соответствующих подпунктам главного меню, могут быть выполнены с помощью клавишных комбинаций; что приятно, комбинации эти работают не только при латинской, но и при кириллической раскладке клавиатуры.

Рассмотрим содержание главного меню подробнее.

Пункт *File*

Пункт включает в себя в основном обычные файловые операции, как то:

- создание файла **New**, сопровождаемое открытием нового окна, относящегося, тем не менее, к текущему сеансу NEdit;
- открытие файла **Open**, в том числе из списка последних **Open Previous**, количество которых настраиваемо; чрезвычайно полезная опция — открытие файла путем выделения его имени (при необходимости — и полного пути) в редактируемом тексте (**Open Selected**); как и при создании файла, каждый документ открывается в своем собственном окне;
- закрытие файла **Close**; в случае его модификации по умолчанию следует запрос на запись изменений, а также установок по умолчанию, если они изменялись;
- сохранение файла **Save**, в том числе и под другим именем **Save as**; последняя опция позволяет сохранить текстовый файл не только в формате UNIX, но и формате DOS (Windows); возможно также дополнение строк

до полного абзаца, завершаемого нажатием клавиши <Enter> (**Add line breaks where wrapped**);

- уникальная опция отмены изменений в уже записанном файле **Revert to Saved**;
- вставка некоего существующего файла в текущий документ **Include File**, осуществляемая в позиции курсора;
- открытие файла макросов **Load Macro File** или тегов **Load Tags File**;
- печать текущего файла **Print** или выделенного фрагмента **Print Selection**;
- выход из программы **Exit**.

Пункт *Edit*

Содержание пункта **Edit** также достаточно тривиально:

- отмена нескольких последних операций и возврат отмененных действий **Undo** и **Redo**;
- вставка **Insert**;
- вырезание **Cut**, копирование **Copy** и вставка **Paste** выделенного фрагмента;
- вставка выделенного фрагмента в виде колонки **Paste Column**; вставляемый фрагмент в этом случае как бы вклинивается в существующий текст;
- удаление выделенного фрагмента **Delete**;
- выделение всего документа **Select All**;
- выделение текста слева или справа от позиции курсора;
- конвертация букв выделенного фрагмента из верхнего регистра в нижний, и наоборот (**Lower Case** и **Upper Case**);
- подпункт **Fill Paragraph** конденсирует абзацы в соответствии с принятыми в **Preferences** правилами переноса слов (*подробнее в разделе о настройках редактора*);
- наконец, пункты **Insert Form Feed** и **Insert <Ctrl> Code** позволяют вставлять всякого рода управляющие символы; в частности, во втором случае вызывается панель с предложением ввести ASCII-код желаемого символа в десятичном исчислении.

Следует заметить, что вставка выделенного фрагмента может осуществляться не только через меню, но и стандартным для Linux способом — щелчком средней кнопки мыши. При этом важно не фиксировать курсор в позиции вставки, это автоматически приводит к снятию выделения и очищению буфера. В этом проявляется отличие NEdit от, скажем, редакторов для KDE, где выделенный фрагмент остается в буфере и после снятия выделения,

вплоть до выделения нового фрагмента, и может быть вставлен повторно в новой позиции. Впрочем, и NEdit может быть настроен так, чтобы сохранять выделенный фрагмент в буфере после фиксации курсора.

Традиционно в NEdit с помощью мыши можно многократно вставлять выделенный фрагмент, так как после первой вставки щелчком средней кнопки выделение не снимается, и процедуру можно повторять (или удалить выделенный фрагмент, нажав клавишу <Del>, не меняя положения курсора) до фиксации курсора в новой позиции; таковая происходит после нажатия любой из клавиш управления курсором, но не при перемещении по телу документа с помощью линейки скроллинга.

Вообще говоря, в терминологии NEdit различаются два типа выделения: первичное (primary, highlighted text) и вторичное (secondary, underlined text), действия над которыми различны.

Первичное выделение осуществляется протаскиванием курсора мыши при нажатой ее левой кнопке или стрелками указателя курсора при нажатой клавише <Shift>, как и в Windows. Выделенный таким образом фрагмент может быть скопирован, удален и вставлен через меню или соответствующие клавишные комбинации, а также вставлен щелчком средней кнопки мыши.

Первичное выделение возможно не только для строк, но и для прямоугольных фрагментов. Оно осуществляется мышью обычным способом, но при нажатой клавише <Ctrl>. С выделенным прямоугольным фрагментом возможны те же действия, что и со строчным. Единственное, прямоугольный фрагмент может быть выделен только при использовании какой-либо моноширинной гарнитуры в качестве экранного шрифта (а NEdit, как будет показано в разделе о настройках, в отличие от большинства текстовых редакторов, допускает использование и пропорциональных гарнитур).

Вторичное выделение осуществляется только мышью. Оно служит для быстрой вставки фрагмента в текущую позицию курсора. Чтобы произвести вторичное выделение, нужно зафиксировать курсор в позиции для вставки, затем навести его (не фиксируя!) на начало выделяемого фрагмента, нажать среднюю кнопку мыши и вести курсор до требуемого места. После этого средняя кнопка мыши отпускается — и выделенный фрагмент (выделение при этом пропадает) волшебным образом копируется в новую позицию.

Если описанную операцию выполнить при нажатой клавише <Shift>, произойдет перемещение выделенного фрагмента. Точно так же, как и при перетаскивании мышью в Windows (drag and drop). Прием, насколько мне известно, не имеющий аналогов в других приложениях для Иксов, далеко не все из которых не поддерживают и обычный drag and drop как таковой.

Search

Опции поиска в NEdit вполне заслуживают отдельного положения в главном меню, поскольку возможности поиска и замены в NEdit чрезвычайно широки. Здесь и поиск введенной последовательности символов, и повторный поиск, и поиск фрагмента, идентичного выделенному. Возможен поиск как буквенных последовательностей (**Literal**), в том числе и с чувствительностью к регистру (**Case Sensitive Literal**), так и регулярных выражений (**Regular Expression**), поиск вперед и назад (**Search Forward** и **Search Backward** соответственно). При этом поддерживается история поиска клавишами управления курсором.

Следует отметить, что строка для ввода искомой последовательности может быть выведена не только в диалоговой панели по вызову соответствующего пункта меню, но и включена в верхней части окна программы, между строкой меню и строкой состояния (с помощью команды **Find Incremental**). В обоих случаях кириллица в строке поиска не отображается; тем не менее, самому по себе поиску это отнюдь не мешает — кириллические символы в теле документа находятся исправно.

Разумеется, возможна и замена найденных фрагментов (**Replace**) и повторная замена (**Replace Again**). Опции замены те же, что и для поиска: с чувствительностью к регистру, с использованием регулярных выражений, вперед и назад и т. д.

В пункте же **Search** есть переход к строке по ее номеру **Goto Line Number** и к выделенному фрагменту **Goto Selected**, установка закладок **Mark**, в качестве которых могут использоваться буквы латинского алфавита, и переход к ранее установленным закладкам **Goto Mark**.

Пункт *Preferences*

Содержание данного пункта меню вполне понятно, что это настройки, среди которых:

- включение/выключение строки состояния (**Statistic Line**) и строки поиска (**Incremental Search Line**);
- выведение/скрытие нумерации строк (**Show Line Number**);
- выбор языкового режима (**Launguage Mode**, в смысле — языка программирования, а не человечьего);
- включение/выключение автоматических отступов в тексте (**Auto Indent**);
- настройка режима переноса строк (**Wrap**);
- установка величины табуляции (**Tabs**);
- выбор экранного шрифта (**Text Font**);

- включение/выключение подсветки синтаксиса (**Highlight Suntax**);
- разрешение (или запрещение) создания резервных копий редактируемого документа при записи (**Make Backup Copy**);
- включение/выключение специальных возможностей — **Incremental Backup и Show Matching (...)**;
- включение/выключение забоя при наборе (**Overtype**);
- установка запрета изменение (**Read Only**) текущего документа.

Уже из этого краткого перечня можно представить себе возможности настройки редактора NEdit, о чем я подробнее буду говорить в следующем разделе. Пока же отмечу только, что все эти настройки имеют силу только в текущем сеансе. Чтобы сделать их перманентными, требуется внести соответствующие изменения в пунктах **Default Setting** и сохранить их (**Save Default**).

Пункт *Shell*

Shell — достаточно необычный пункт меню текстового редактора. Через него можно, во-первых, запустить на исполнение любую команду или программу. Это делается с помощью подпункта **Execute Command**, который вызывает нечто вроде мини-терминала. В нем может быть введена соответствующая команда оболочки (или любая другая) с любыми требующимися параметрами. Поддерживается посредством клавиш управления курсором, история команд, правда, только данных в текущем сеансе.

Во-вторых, через подпункт **Execute Command Line** можно определить в качестве команды на запуск строку из текущего документа. Для этого достаточно установить курсор в любое ее место и выбрать соответствующий пункт меню. Разумеется, если строка эта имеет смысл и записана синтаксически правильно. В противном случае в теле документа автоматически появится сообщение об ошибке. Например:

```
netscape
An error occurred running

/usr/lib/netscape/netscape-communicator
```

Или даже полная справка по использованию команды, как в случае

```
ispell

Usage: ispell [-dfile | -pfile | -wchars | -Wn | -t | -n | -x |

-b | -S | -B | -C | -P | -m | -Lcontext | -M | -N | -Ttype | -V] file

.....
```

и т. д. Разумеется, эта опция наиболее полезна для программистов. Однако и простым смертным она будет весьма нелишней при отладке сценариев

командной оболочки. Необходимость в чем может возникнуть при желании нарастить функциональные возможности NEdit.

Дело в том, что сценарии эти могут быть встроены в меню **Shell**. Некий их набор включен в последнее по умолчанию. Это проверка правописания (spell), статистика (wc, от word count — подсчет строк, слов и знаков), сортировка строк и их нумерация, а также сугубо программистские сценарии, например, make для сборки программы из набранного исходного текста. Однако ничто не мешает пополнять меню Shell своими сценариями любого назначения, освободив его от ненужных лично вам элементов.

Пункт *Macro*

Этот пункт разделяется на две секции. Первая, стабильная, предназначена для создания и запуска пользовательских макрокоманд. В ней можно видеть:

- включение режима протоколирования макросов (**Learn Keystrokes**);
- нормальное завершение (**Finish Learn**) и прерывание (**Cansel Learn**) протоколирования;
- воспроизведение (**Replay Keystrokes**) и повторение (**Repeat**) запротоколированных макрокоманд.

Вторая секция настраиваемая, в нее можно включить макрокоманды, как идущие в комплекте, так и созданные собственноручно. Например, с помощью протоколирования, иначе они не сохранятся по завершении текущего сеанса.

В комплекте с NEdit идет несколько макросов — дополнение частично напечатанных слов путем сопоставления их со словарем (**Complete Word**) или ответы на письма — с цитированием (**Quote Mail Reply**) и без оного (**Unquote Mail Reply**).

Кроме того, имеется серия языково-зависимых макросов, предназначенных для программирования. Они появляются в меню только при включении соответствующего языкового режима. Однако, как и в случае с Shell, не видно причин, по которым нельзя было бы исключить из меню заведомо не нужные макросы, заменив их собственными. Именно это и будет темой заключительных разделов главы.

Пункт *Windows*

Понимается под этим, как не трудно догадаться, управление окнами, правда, довольно ограниченное. Во-первых, можно разделить окно с текущим документом на две независимо скроллируемые части (**Split Window**). Только по горизонтали, но зато сколько угодно раз: исходное окно делится ровно

пополам, затем каждая половина — еще пополам, и т. д., насколько хватит разрешения экрана.

Для снятия разделения предназначен пункт **Close Pane**. При этом закрывается та часть окна, в которой находится курсор.

Наконец, в пункте **Windows** содержится список открытых в текущем сеансе документов. Каждый из них открывается или создается (через пункты меню **File | Open** или **File | New**, соответственно) в своем окне с полным набором его атрибутов (управляющими элементами, меню, строкой состояния и т. д.). Однако все они принадлежат текущему сеансу, и, скажем, изменение настроек сказывается на них всех. А переключение между окнами, кроме принятого в данном оконном менеджере, можно осуществить и через их список в пункте меню Windows.

Пункт *Help*

Встроенная помощь редактора NEdit весьма богата, что можно видеть из простого перечисления содержания этого пункта меню.

- **Getting Started** — краткое описание основных особенностей программы;
- **Basic operations**, где описаны:
  - **Selecting Text** — приемы выделения текста;
  - **Finding and Replacing Text** — поиск и замена фрагментов;
  - **Cut** и **Paste** — вырезания и вставки;
  - **Using the Mouse** — использование мыши;
  - **Keyboard Shortcuts** — комбинации "горячих клавиш";
  - **File Format** — формат файлов NEdit;
- **Features for Programming** — для программистов; не будучи таковым, задерживаться здесь не буду;
- **Regular Expressions** — здесь весьма подробно описаны синтаксис регулярных выражений и прочие их особенности, а также приведены примеры;
- **Macro/Shell Extensions** — заслуживает всяческого внимания; именно здесь подробно описаны: использование скриптов оболочки командной строки, процесс протоколирования макрокоманд, синтаксис встроенного макроязыка NEdit и пр.;
- **Customizing** — не менее интересная позиция; под этой рубрикой приведены сведения о всех видах настройки NEdit; что составит предмет следующего раздела;
- **NEdit Command Line** — описание параметров для запуска NEdit; большая часть достигаемых таким путем опций может быть реализована также и через настройки;

- **Server mode and nc** — описание использования NEdit в режиме "клиент-сервер";
- **Crash recovery** — описание действий в экстремальных ситуациях.

Кроме перечисленного, в пункте **Help** содержатся подробные сведения о текущей версии редактора, политике его распространения, приведены списки рассылки и небольшой перечень проблем и ошибок.

Пора, однако, заканчивать с рассмотрением главного меню. Поскольку в NEdit существует еще и меню контекстное (в терминологии NEdit — **Window Background Menu**), вызываемое, как положено, щелчком правой кнопки мыши на выделенном фрагменте текста. Правда, оно достаточно простое, содержит лишь пункты **Undo** и **Redo**, **Cut**, **Copy** и **Paste**, что в комментариях не нуждается. Однако, как будет показано, и оно может быть наращено.

Таким образом, из рассмотрения меню функциональность NEdit становится ясна. Он обладает очень развитыми средствами ввода и редактирования текстов как общего содержания, так и исходных текстов на языках программирования и разметки. Меню его организовано логичным и привычным для пользователя Windows способом, комбинации "горячих клавиш" многочисленны и обычно имеют простой мнемонический смысл. Все это, вместе с мощной системой помощи, делает NEdit простым в освоении.

К недостаткам NEdit (или, вернее, особенностям, которые могут не нравиться), можно отнести отсутствие инструментальной панели с кнопками. Еще в NEdit нет штатных средств управления проектом. Хотя бы в элементарном исполнении, то есть в виде открытия и сохранения группы связанных файлов (глав книги, например, или серии Web-страниц). Тем не менее, все это не перечеркивает высоких достоинств редактора.

Однако мало было бы проку от богатства возможностей редактирования, если бы они не сопровождались адекватным богатством настроек. Что же, посмотрим, каковы они, возможности настройки.

Настройка NEdit

Я уже говорил, большая часть настроек редактора NEdit сконцентрирована в пункте **Preference** главного меню. И даже кратко перечислил возможности настройки текущей сессии. Однако нас больше интересует не сиюминутный результат настройки, но ее долгосрочная перспектива. Для чего следует рассмотреть подпункт **Preference | Default Settings**. Содержание его близко подпунктам меню **Preference** и включает установки:

- языкового режима **Language Modes**;
- включения/выключения автоматических отступов **Auto Indent**;
- режима переноса слов **Wrap**;

- величины табуляции **Tabs**;
- экранного шрифта **Text Fonts**;
- настройки меню **Customize Menus**;
- режима поиска **Searching**;
- подсветки синтаксиса **Syntax Highlighting**;
- переключателей вывода строки состояния **Statistics Line**, строки поиска **Incremental Search Line** и нумерации строк **Show Line Numbers**;
- резервного копирования **Make Backup Copy**;
- включения сортировки ранее открытых файлов (**Sort Open Prev. Menu**);
- выдачи предупреждающих сообщений **Warning**;
- начального размера окна при запуске NEdit **Initial Window Size**.

Большинство из этих пунктов заслуживают того, чтобы задержать на них внимание. Чем мы и займемся, хотя и в иной несколько последовательности. Но начнем по порядку, с языкового режима.

Режимы *Language Modes*

Как уже говорилось, под этим подразумевается язык не человеческий, а программистский. И список поддерживаемых режимов включает, не считая чисто текстового **Plain**, почти все известные мне (и многие неизвестные) языки программирования и разметки. Здесь и C и C++, Java и JavaScript, Ada, Fortran, Pascal, Python, Tcl, awk, языки командных оболочек линии sh и линии csh, языки разметки SGML/HTML и LaTeX, PostScript, SQL и прочая, и прочая, и прочая. Разумеется, есть и режим собственного языка NEdit Macro.

Кроме языков, поддерживаемых в штатном комплекте, можно ввести поддержку любого иного языка программирования, в том числе и изобретенного. Для этого достаточно выбрать в списке языков пункт **New**, определить для нового языка его имя, закрепленное расщирение имени файла и прочие параметры.

Первое видимое следствие переключения языкового режима — появление подсветки синтаксиса соответствующего языка (если, конечно, эта опция включена). Так, по умолчанию в NEdit задействован режим **Plain text**. В этом случае, скажем, теги в HTML-документе показываются теми же черными (для примера) буквами, что и его содержание. Если же переключиться в режим **SGML/HTML**, теги, их аргументы и значения последних заиграют различными цветами. Впрочем, при открытии файла определение его типа обычно происходит автоматически, по расширению (и правильно), и соответствующий языковый режим включается сам собой.

Однако различие между языковыми режимами не исчерпывается цветовой гаммой. Многие макросы имеют языковую привязку. И появляются в меню **Macro** только тогда, когда включается соответствующий их языку режим.

Настройки *Auto Indent*, *Wrap* и *Tabs*

Разумеется, подсветка синтаксиса необходима, в первую очередь, программистам. Хотя и всем прочим, хоть иногда просматривающим исходники Web-страниц, тоже не вредит. А вот следующая группа настроек предназначена и для простых смертных, так как определяет правила набора текста. Так, **Auto Indent** — это просто включение или выключение (**Off**/**On**) возможности создания автоматических абзацных отступов. Поясню на примере: если вам нужно набрать серию абзацев с общим отступом 7 пунктов от левого края, вы задаете (например, клавишей табулятора) отступ только первого абзаца — во всех остальных при включении (**On**) опции **Auto Indent** будет появляться автоматически, до его уничтожения вручную.

В этом же пункте, кроме переключения, есть еще одна опция — **Smart**, но она не активизирована. Подозреваю, что ее можно включить только при режимах каких-либо языков, предусматривающих некие фиксированные отступы в строках исходного текста.

А вот величина отступа, задаваемого клавишей табуляции, и определяется в пункте **Tabs**. Здесь нужно просто задать (в знаках) величину отступа при табуляции. Правда с оговоркой **for hardware space**, поскольку тут же присутствует и опция **emulate tabs**, то есть предполагается и некая программная табуляция — для терминалов, клавиши <Tab> не имеющих.

Опция **Wrap** важна, так как она отражается на ряде других параметров набора текста. Для нее возможны варианты:

- **None** — каждый абзац набирается в одну строку вплоть до нажатия клавиши <Enter>; это вполне приемлемо для исходных текстов программ, но вряд ли очень удобно для набора текстов просто, так как строка "убегает за горизонт";
- **Auto Newline** — выбор этой позиции приводит к тому, что каждая строка обрывается на границе окна, в результате чего сплошной текст (вроде этого, например) приобретает читабельность; однако при экспорте его в какой-либо текстовый процессор обнаруживается, что каждая строка заканчивается символом возврата каретки (то есть образует самостоятельный абзац), искоренение которых, например, в MS Word — занятие не из самых веселых;
- **Continuous** — оптимальный выбор для набора длинных связных текстов; в этом случае при достижении границы окна происходит разрыв экранной (не истинной!) строки без появления лишнего символа возврата каретки

и, соответственно, без образования нового абзаца; каковой появляется только по нажатии клавиши <Enter>; это аналогично условиям переноса строк по умолчанию в emacs; в последнем, в отличие от NEdit, разрыв экранной строки маркируется обратным слэшем (\), что, конечно, наглядно показывает отличие строк экранных и истинных, но, с другой стороны, загромождает текст;

- **Wrap Margin** — позволяет явным образом задать длину строки в знаках, после чего происходит ее разрыв символом возврата каретки.

Я столь подробно остановился на параметрах переноса слов, потому что они представляются мне наиболее существенными при наборе текста. Особенно если в дальнейшем предполагается экспорт его в текстовый процессор или какую-либо программу верстки. К тому же именно установками пункта **Wrap** определяется реакция на команду меню **Edit | Fill Paragraph**: при установке в позиции **Continuous** текст в NEdit, независимо от того, в каком режиме переноса слов был набран, будет сконденсирован в единые абзацы, разделами между которыми будут только явные нажатия клавиши <Enter>.

Не уверен, что выразился вполне внятно, поэтому поясню на примере: некий текст был набран в режиме **Auto Newline**, после чего выяснилось, что его следует экспортировать в MS Word; чтобы избавить себя в дальнейшем от ручного искоренения символов возврата каретки или головной боли по измышлению автоматизации этого процесса, достаточно просто пройтись по всем абзацам текста, нажимая <Ctrl>+<J>, — и при включенной опции **Continuous** мы получим его аккуратно поделенным именно тем образом, что и требуется.

Выбор *Text Fonts*

Этому вопросу я всегда придавал первостепенное значение. Во-первых, из соображений эстетических (грешным делом, люблю красивые шрифты), во-вторых, по причине плохого зрения. Так вот, с точки зрения выбора экранных шрифтов NEdit заслуживает аплодисментов.

При вызове пункта **Text Fonts** возникает панель с предложением определить основную гарнитуру (по умолчанию нормального, то есть не курсивного и не полужирного начертания) и отдельно гарнитуры для каждого из начертаний (опять же, курсивного, полужирного и полужирного курсивного). Впрочем, можно нажать экранную кнопку **Fill Highlight Fonts from Primary** — и тогда все прочие, кроме нормального, начертания унаследуют его гарнитуру.

В строке **Primary Font** определить гарнитуру (и прочие параметры) экранного шрифта можно вводом вручную, с указанием всех требуемых атрибутов. Однако проще нажать экранную кнопку **Browse** и выбрать из списка

всех доступных системе шрифтов. Выбор осуществляется в панели из трех колонок; в них необходимо явным образом отметить все три доступные для выбора атрибута:

- **Font** — гарнитура;
- **Style** — начертание;
- **Size** — кегль.

Что характерно (вернее, не характерно для большинства текстовых редакторов, оперирующих обычно только моноширинными гарнитурами), хотя по умолчанию в списке присутствуют моноширинные шрифты, можно включить также показ шрифтов пропорциональных и использовать их в свое удовольствие.

Подчеркну, таким образом можно определить шрифт только для экранного представления набираемого текста. На шрифты интерфейсных элементов, а также системы помощи это не окажет никакого воздействия. Разумеется, и они могут быть подобраны по вкусу, но об этом — несколько позже.

Настройка *Customize Menus*

Этот подпункт дает возможность настроить три элемента — пункты **Shell** и **Macro** главного меню, а также контекстное, или, в терминологии NEdit, фоновое, меню **Window Background Menu**, вызываемое правой кнопкой мыши.

При настройке пункта **Shell** вызывается панель со списком имеющихся команд, которые с помощью соответствующих экранных кнопок могут быть удалены (кнопка **Delete**), скопированы (кнопка **Copy**) или перемещены вверх или вниз по списку (кнопки **Move^** и **Move** соответственно).

В расположенном ниже поле командной строки **Shell Command to Execute** содержится собственно код скрипта командной оболочки. Здесь можно непосредственно руками отредактировать соответствующую команду.

Например, в комплекте NEdit предусмотрен скрипт для проверки орфографии (пункт **spell**). Он вызывает стандартную программу ispell в окне терминала

```
cat>spellTmp; xterm -e ispell -x spellTmp; cat spellTmp; rm spellTmp
```

и, естественно, понимает только английский язык. Однако обучить его русскому языку (или любому другому при наличии соответствующего словаря) трудов не составит. Для этого достаточно указать в командной строке параметр `-d russian` (предписывающий обращаться при спеллинге именно к русскому, а не какому другому словарю) — и все в порядке.

Заодно, при желании, можно указать и параметры окна терминала (цвет текста и фона, геометрию и т. д.). Или просто заменить вызов xterm на ту

терминальную программу, которая вам больше нравится. В результате чего скрипт проверки орфографии примет примерно следующий вид:

```
cat>spellTmp; rxvt -e ispell -d russian -x spellTmp; cat spellTmp; rm
spellTmp
```

Кроме того, в строке **Menu Entry** можно изменить название скрипта (именно содержимое этой строки и выступает в качестве пункта меню) или организовать иерархическое меню, а также привязать скрипт к какому-либо языку; в последнем случае скрипт будет фигурировать в меню только в том случае, если выбрать соответствующий языковый режим.

Кроме того, в пункте **Accelerator** за скриптом можно закрепить некую комбинацию "горячих клавиш". А поле **Mnemonic** предназначено для ввода литеры, которая будет в меню выделена и послужит для быстрого доступа к команде. Можно изменить и ряд условий исполнения команды, о чем будет разговор в соответствующем разделе.

И так можно поступить (при необходимости) с любым из имеющихся в наборе скриптов. Не следует только забывать, завершив редактирование одного, нажимать кнопку **Apply** перед переходом к следующему — иначе изменения будут утеряны.

Разумеется, в меню **Shell** можно включить и скрипты собственного изготовления, для чего в списке имеется пункт **New**. Однако это будет нашим занятием в одном из следующих разделов.

Сходным образом настраивается и меню **Macro**. На предназначенной для этого панели также имеется список макрокоманд с кнопками для их удаления, копирования и перемещения вверх и вниз, поле имени **Menu Entry** с возможностью создания многоуровневых иерархических меню, привязкой к языковому режиму и закрепленной клавишной комбинации, а также обширное поле, предназначенное для кода макроса, доступное для непосредственного редактирования.

Выбрав из списка макросов пункт **New**, можно нарастить меню собственными командами на языке NEdit Macro:

- написать в поле **Macro Command Execute** соответствующий код вручную;
- посредством клавиши **Paste Learn/Replay Macro** поместить туда запротоколированную (с помощью пунктов меню **Macro** | **Learn Keystrokes**) последовательность действий.

Абсолютно аналогичным образом можно настроить и контекстное (или фоновое) меню, вызываемое щелчком правой кнопки мыши на выделенном фрагменте. Как уже говорилось, по умолчанию оно включает привычные для Windows-мигранта пункты **Undo**, **Redo**, **Cut**, **Copy** и **Paste**, которые с помощью экранных кнопок **Delete**, **Copy**, **Move^** и **Move** можно удалить, скопировать и переместить.

А избрав в списке пункт **New**, можно пополнить контекстное меню собственными командами, написанными на языке NEdit Macro вручную или посредством протоколирования действий.

Параметры *Searching*

Покончив с настройками меню, можно заняться определениями параметров поиска. Я уже говорил, что возможности для этого в редакторе NEdit весьма обширны. А настраиваемость придает им еще и гибкость. В пункте **Searching** присутствует два переключателя:

- **Verbose** — включение поиска многословных фрагментов;
- **Keep Dialogs Up** — непрерывный поиск последующих вхождений (по умолчанию для поиска следующего вхождения требуется выбрать в главном меню **Search | Find Again**).

Третий пункт настройки поиска — режим по умолчанию:

- **Literal** — искомое значение рассматривается как буквенная последовательность;
- **Case Sensitive** — с чувствительностью к регистру;
- **Regular Expression** — поиск регулярных выражений, представление о синтаксисе которых можно получить из соответствующего пункта меню **Help**.

Переключатели

Эта группа настроек — просто переключатели, с помощью которых можно показать или скрыть:

- подсветку синтаксиса (**Syntax Highlighting**) языка, соответствующего данному режиму;
- строку состояния (**Statistics Line**);
- строку поиска (**Incremental Search Line**);
- нумерацию строк (**Show Line Numbers**).

Правда, для пункта **Syntax Highlighting** настройке поддается также **Recognition Patterns** и **Text Drawing Styles**. Что такое первое — я пока не разобрался, а второе — это приписывание каждому элементу языковых конструкций цвета и шрифтоначертания. Для языка HTML, например, можно определить различные цвета тегов, их атрибутов и значений последних, комментариев, метатегов и т. д.

Переключатель **Make Backup Copy** разрешает или запрещает создание резервных копий редактируемого файла. Включение его приводит к тому, что

при перезаписи измененного документа предыдущая его версия сохраняется в виде файла с расширением bck.

С помощью **Sort Open Prev. Menu** можно включить алфавитную сортировку ранее открывавшихся файлов, входящих в список **File | Open Previous**. При выключении этой опции файлы в нем следуют в хронологическом порядке.

Пункт *Warning*

Этот пункт представляет собой конфигурирование сообщений при закрытии файла или выходе из редактора. Здесь имеется всего два переключателя. Первый отвечает за выдачу предупреждения о не сохранении изменений в закрываемом файле, второй же разрешает или запрещает сообщение об изменении установок по умолчанию при закрытии программы. На мой взгляд, оба далеко не лишние, и не худо бы включить их по умолчанию.

Размеры *Initial Window Size*

Как не трудно понять, здесь задается размер окна по умолчанию, возникающего при запуске NEdit. Предусмотрено пять вариантов такового:

- 24 на 80 знаков;
- 40 на 80 знаков;
- 60 на 80 знаков;
- 80 на 80 знаков;
- настройка геометрии (**Custom...**).

Первые четыре в комментариях не нуждаются, пятый же вариант позволяет задать произвольное количество строк и столбцов. Речь идет о размере окна именно в знаках, а не в пикселах, поэтому реальный размер окна зависит от используемых по умолчанию гарнитуры и кегля шрифта.

Конфигурирование

Надеюсь, из приведенного описания становится ясно, что через меню **Preferences | Defaults Settings** в NEdit можно настроить почти все. Единственно, следует не забыть сохранить сделанные изменения, дабы сделать их перманентными (через **Preferences | Save Defaults**).

Однако способов изменения некоторых параметров через меню обнаружить не удается. Так, остались цвета фона и курсора, шрифт надписей меню и цвет его линейки, положение полосы скроллирования и еще кое-что.

Отчаиваться не следует — на сей предмет существует метод ручной правки конфигурационного файла /$HOME/.Xdefaults, описывающего особенности X-приложений.

Секции для NEdit в нем нет. Однако создать ее (хотя бы собственными средствами этого редактора) труда не составит. Для этого достаточно вставить такие строки:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! NEdit
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Строки с восклицательными знаками, естественно, не обязательны, служа всего-навсего ограничителями секции, а строка `! NEdit` — комментарий, дабы не забыть, о чем, собственно, идет речь.

Теперь дело за малым — нужно создать наполнение секции. Можно обратиться к системе помощи, тому самому ее пункту, где говорится о конфигурировании NEdit и, в частности, о ресурсах X Window (**Help | Customizing | X Resources**).

Здесь можно найти список почти всех параметров, которые не подлежат изменению через меню **Preferences**, но которые в принципе можно изменить:

- `Nedit.shell: /bin/csh`

  указание на оболочку командной строки, скрипты которой выполняются через меню **Shell**;

- `Nedit.wordDelimiters: .,/\\`'!@#%^&*()-=+{}[]":;<>?`

  описывает знаки, маркирующие (в дополнение к пробелам и табуляции) границы между словами при их выделении мышью (двойным щелчком на слове);

- `Nedit.remapDeleteKey: False`

  указание значения `True` переопределяет клавишу <Delete> как <Backspace>;

- `Nedit.stdOpenDialog: False`

  возвращение значения `True` устанавливает стандартный для приложений, основанных на Motif, стиль диалоговых панелей;

- `Nedit.bgMenuButton: [<Shift>][<Ctrl>][<Meta>][<Alt>][Btn3Down]`

  способ доступа к контекстному (или фоновому) меню. По умолчанию для этой цели служит, как и в Windows, правая кнопка мыши. Однако можно использовать и любой из четырех других вариантов;

- `Nedit.maxPrevOpenFiles: 30`

  максимальное количество файлов в списке, вызываемом через меню **File | Open Previous**. Список открывавшихся ранее документов хранится в конфигурационном файле /$HOME/.Neditdb;

- `Nedit.printCommand: (system specific)`

  этот параметр и серия следующих за ним (начинающихся с `NEdit.print`) определяют условия печати документов из редактора NEdit. По умолчанию все они используют системные установки. Насколько я понимаю, лучше их таковыми и оставить, если нет веских причин для иного решения;

- `Nedit.multiClickTime: (system specific)`

  время ожидания второго щелчка кнопкой мыши, в течение которого он считается за двойной. Здесь логично сохранить системную установку — не зря же вы ее делали для всех приложений;

- `Nedit*scrollBarPlacement: BOTTOM_LEFT`

  эта строка описывает положение и вид полосы прокрутки. При значении в примере (как ни странно, это не значение по умолчанию, уже при первом запуске NEdit полоса прокрутки располагается справа) она должна располагаться вдоль левой границы окна и прокручиваться вперед смещением ползунка сверху вниз. Другие возможные значения:

  - `BOTTOM_RIGHT`;
  - `TOP_LEFT`;
  - `TOP_RIGHT`;

  в двух последних случаях прямой скроллинг осуществляется движением ползунка снизу вверх;

- `Nedit*text.autoWrapPastedText: False`

  по умолчанию запрещает автоматический перенос слов при наборе текста. Лучше так и оставить, различные опции переноса слов для каждого конкретного случая проще настроить через меню **Preferences | Wrap** (или **Preferences | Default Settings | Wrap**), как было описано;

- `Nedit*text.foreground: black`

  цвет шрифта набираемого текста;

- `Nedit*text.background: white`

  в этой строке определяется цвет фона редактируемого поля;

- `Nedit*text.selectForeground: black`

  определяется цвет текста выделенного первичным способом (левой кнопкой мыши или стрелками указателя курсора при нажатой клавише <Shift>) фрагмента;

- `Nedit*text.selectBackground: gray80`

  определяет цвет фона выделенного первичным способом фрагмента. Следует заметить только, что число после наименования цвета обозначает яркость (saturation) его в процентах. То есть `gray80` — это светло-серый,

а, скажем, `gray20` — темно-серый. Во всех относящихся к цветовой гамме пунктах возможны также значения имени цвета с прилагательными — `light` или `dark`;

- `Nedit*text.highlightBackground: red`

  определяет цвет подсветки парных элементов (например, открывающей скобки при фиксации курсора на закрывающей);

- `Nedit*text.cursorForeground: black`

  цвет курсора;

- `Nedit*text.lineNumForeground: gray47`

  цвет, которым передается нумерация строк (если эта опция включена);

- `Nedit*text.blinkRate: 600`

  частота мерцания курсора в миллисекундах. Если установить значение 0, курсор, к счастью, мерцать не будет вообще;

- `Nedit*foreground: black`

  задает цвет текста меню и диалоговых панелей;

- `Nedit*background: gray70`

  здесь определяем цвет фона меню, диалоговых панелей и полосы прокрутки;

- `Nedit*fontList: helvetica-bold-14`

  можно, наконец, определить гарнитуру, начертание и кегль шрифта интерфейсных элементов.

Далее следует предложение использовать всякого рода расширения имен элементов, такие как `.background` и т. д., для изменения цветов, шрифтов и прочих характеристик следующих далее элементов.

`Nedit.statsForm`

предлагает определить указанные характеристики для статусной строки и строки поиска. При желании одна характеристика может распространяться на оба элемента, для этого вместо (`.`) следует использовать (`*`). Например:

- `Nedit*statsForm*background: gray`

  задает для них единый серый цвет фона;

- `Nedit*menuBar`

  можно по той же модели определить характеристики линии главного меню;

- `Nedit*textHorScrollBar`

  полосы горизонтальной прокрутки;

- `Nedit*textVertScrollBar`

  полосы вертикальной прокрутки;

- `Nedit*helpText`

  позволяет определить характеристики окна системы помощи. Однако у меня она на шрифт сообщений помощи никакого действия не оказала. Тем не менее, эту проблему удалось победить, поместив собственноручно сконструированную строку

  ```
  Nedit*font: -cronyx-fixed-medium-r-*-*-20-*-*-*-*-*-koi8-r
  ```

  Не возьмусь судить, насколько она синтаксически грамотна, но, по крайней мере, после этого шрифт сообщений в окне помощи действительно изменился.

Таким образом, в секцию `!Nedit` файла /$HOME/.Xdefaults можно внести изрядное количество строк. Но можно — не значит нужно. Достаточно ограничиться только теми строками, определяющими характеристики, не устраивающие вас по умолчанию.

Таким образом, редактированием файла /$HOME/.Xdefaults удается настроить то немногое, что не получается настроить через главное меню. Однако в домашнем каталоге пользователя можно обнаружить еще два конфигурационных файла, имеющих отношение к NEdit, — .Nedit и .Neditdb.

Не факт, что возникнет необходимость их ручной правки, но полноты картины для вкратце опишу их структуру.

Содержание файла .Nedit составляет описание тех самых параметров, которые настраиваются через главное меню **Preferences | Default Settings**. К коему и следует, как гласит комментарий, прибегать для его изменения. Тем не менее, он доступен и для ручного редактирования, что, как и всякую свободу выбора, следует расценить только положительно.

Начальная (после комментариев) строка файла — указание на номер версии:

```
Nedit.fileVersion: 5.X
```

Затем следуют секции описания настраиваемых меню **Shell**, **Macro** и контекстного:

```
Nedit.shellCommands: \
spell:<Alt>+B:s:EX:\n\


Nedit.macroCommands: \
Headers>header1:F1::: {\n\
...
Nedit.bgMenuCommands: \
Undo:::: {\n\
...
```

соответственно. Описывать их содержимое я не буду. Скажу только, что первая строка после заглавия секции содержит название скрипта или макроса (в том самом виде, как оно отображается в соответствующем меню. Затем, разделяясь двоеточиями, следуют всякого рода дополнительные сведения, такие как закрепленная клавишная комбинация или мнемоническое обозначение.

Следующие секции

```
Nedit.highlightPatterns:
C:Default\n\
C++:Default\n\
...
Nedit.languageModes:     C:.c
.h::::::".,/\\`'!|@#%^&*()-=+{}[]"":;<>?~"\n\
C++:.cc .hh .C .H
.i .cxx
...
Nedit.styles:     Plain:black:Plain\n\

Comment:gray20:Italic\n\
...
```

определяют разнообразные параметры (подсветка, отступы и т. д.) для различных языковых режимов.

За этим следуют опции переноса слов, автоматических отступов, подсветки синтаксиса и прочие, подробно рассмотренные ранее, в разделе о настройке через меню, смысл их достаточно прозрачен. Завершающие разделы — описание геометрии открываемого окна, шрифтов редактируемого текста — также не должны вызвать вопросов. И посему, скажем, при необходимости можно изменить, например, кегль шрифта, не обращаясь к меню. Другое дело, что вряд ли такая необходимость возникнет...

Ну, а файл .Neditdb — это просто список (задумчиво называемый базой данных) ранее открывавшихся файлов с указанием полных абсолютных путей. Что характерно, в списке пункта меню **File | Open Previous** они даются без дублирования — все же буквы db в названии присутствуют не зря.

Макросы и Shell

Надеюсь, мне удалось продемонстрировать если и не все богатство штатных возможностей редактора NEdit, то изрядную его долю. Однако величие программы в том, что штатными средствами его функциональность не ис-

черпывается. Вкратце о возможности создания собственных макросов (путем написания их кода руками) я уже упоминал. Однако есть и другой способ, который для начала может оказаться более простым — протоколирование действий. Для этого требуется всего-навсего:

1) включить режим протоколирования (через меню **Macro | Learn Keystrokes**);
2) произвести посредством клавиатуры (но не с помощью мыши!) все необходимые действия в той последовательности, в какой мы хотим их сохранить на будущее;
3) завершить режим протоколирования (через меню **Macro | Finish Learn**);
4) в случае ошибки при наборе команд — прервать протоколирование (через меню **Macro | Cancel Learn**) и, включив протоколирование по новой, повторить требуемые действия уже без ошибок.

Запротоколированная последовательность действий будет функционировать в течение данного сеанса работы с NEdit. И может быть вызвана первый раз через меню **Macro | Reply Keystrokes** и **Macro | Repeat**. Однако при следующем запуске NEdit будет утрачена.

Чтобы этого не случилось, следует сохранить ее в меню для потомства. Для этого, успешно завершив протоколирование данного макроса, отправляемся в меню **Preferences**, выбираем там пункт **Default Settings**, а в нем — опцию **Customize Menu**. Каковая, в свою очередь, обнаруживает, как уже говорилось, пункт **Macro Menu**, вызывающий панель **Macro Commands**.

В списке доступных команд (в левой части панели) фиксируемся на пункте **New**. В поле **Menu Entry** задаем название команды и, при необходимости, название более высокого уровня для группы сходных команд, например, таким образом:

```
Group commands>Command1
```

без пробелов перед и после знака >. Здесь же можно приписать команду к какому-либо из доступных языковых режимов. Например,

```
Group commands>Command1@SGML/HTML
```

будет указывать, что данная команда активизируется только при выборе языкового режима SGML/HTML.

Затем заполняем поле **Accelerator**. Здесь за нашей командой можно закрепить какую-либо клавишу (из числа, например, функциональных, или Windows-клавиш) или их комбинацию: например, сочетание клавиш <Alt> или <Ctrl>, возможно, в сочетании с <Shift>, с какой-либо буквой, желательно, имеющей мнемонический смысл.

Для этого достаточно просто зафиксировать курсор в поле **Accelerator** и нажать требуемую клавишу (например, <F12>) или их комбинацию. Следует только внимательно следить, чтобы эта клавишная комбинация не исполь-

зовалась для вызова штатных функций NEdit: в этом случае приоритет в любом случае будет за последними, а, возможно, клавишная комбинация не будет вызывать ни старого, ни нового закрепленного действия.

Вслед за этим можно заполнить поле **Mnemonic**, введя в него какую-либо букву из имени нашей команды (как оно указано в поле **Menu Entry**). При вызове меню буква эта будет подчеркнута и, теоретически, может использоваться для быстрого вызова команды (сочетанием ее с клавишей <Alt>). Однако на деле это работает далеко не всегда. Вернее, почти всегда не работает, поскольку большинство букв латинского алфавита уже задействованы для штатных команд NEdit.

Наконец, последнее из предварительных действие — это включение, по потребности, переключателя **Requires Selection**. Он предназначен для команд, которые осуществимы только с предварительно выделенными фрагментами (типа копирования, вырезания и т. д.).

А вот теперь нажимаем экранную кнопку **Paste Learn/Replay Macro** (правой части панели). И текст макроса волшебным образом появляется в поле **Macro Command to Execute**. Где может быть любым образом отредактирован, дополнен, сокращен и т. д. После чего жмем кнопку **OK**, выходя их режима редактирования **Macro Menu**, и с удовольствием и пользой применяем новую функцию на практике. Если все работает нормально — сохраняем текущую ситуацию через **Preferences | Save Defaults** для увековечивания внесенных нами изменений.

Таким образом можно насытить меню редактора NEdit командами для выполнения часто требуемых, но отсутствующих в штатном комплекте действий. Например, как будет показано ниже, для ввода тегов HTML. Однако и этого может показаться недостаточно для полного счастья. Например, возникнет потребность в систематическом исполнении каких-либо внешних программ. Например, команд оболочки.

К тому же подчас необходимо, чтобы команда оболочки выполнялась в отношении редактируемого в данный момент в NEdit документа. Примером чего служит приведенная в прошлом разделе команда `ispell` для проверки орфографии. И тут нам на помощь приходит возможность редактирования меню **Shell** (через **Preferences | Default Settings | Customize Menu | Shell Menu**).

Выбрав соответствующие пункты приведенной последовательности, вызываем панель **Shell Commands** и выбираем из списка доступных команд пункт **New**. Подобно тому, как это проделывалось для макросов, заполняем поля **Menu Entry**, то есть имени команды (также, возможно, иерархически построенного и привязанного к языковому режиму), **Acceleretor** (для закрепления комбинаций клавиш) и **Mnemonic** (для выделения литеры быстрого вызова).

Затем определяем входные элементы команды в строке переключателей **Command Input**. Здесь можно предписать команде исполняться для выделенного фрагмента (**selection**), текущего окна (**window**), каждого элемента документа (**either**) или ни для чего (**none**). Объяснить смысл каждого переключателя в общем виде я несколько затрудняюсь, но на практике смысл их в отношении каждой конкретной команды обычно достаточно прозрачен.

Аналогично и с выходными элементами команды. Каковые могут быть показаны в текущем или новом окне, а также в виде диалоговой панели.

Следующая далее группа переключателей при установке в положение **On** предписывает:

- замещать элемент ввода элементом вывода (**Output replace input**);
- сохранять текущий файл перед выполнением команды (**Save file before executing command**);
- перечитывать его же после выполнения таковой (**Re-load file after execute command**).

Определившись и с этим, можно непосредственно задавать команду вместе со всеми требуемыми параметрами и их значениями. Это делается в поле ввода команды (**Shell Command to Execute**), точно так же, как в командной строке консоли или терминального окна.

Однако можно и набрать команду непосредственно в поле редактирования NEdit, после чего просто скопировать (с помощью мыши или клавишных комбинаций) ее в поле ввода команды. Преимущество последнего способа в том, что предварительно (с помощью меню **Shell Execute Command Line**) можно проверить работоспособность команды и, руководствуясь сообщениями об ошибках, ее отредактировать. Можно, разумеется, и просто скопировать успешно запускаемую команду и окна терминала...

Завершив создание новой команды, принимаем введенные изменения (через экранную кнопку **Apply**) или выходим из режима редактирования **Shell** через **OK**. А, выполнив проверку правильности созданной команды, сохраняем ситуацию (через **Save Defaults**) для использования в дальнейшем.

Пример применения: HTML-редактор своими руками

Не могу поручиться, что в предыдущем разделе внятно изложил возможности NEdit по созданию и редактированию макрокоманд и команд оболочки. И потому попробую проиллюстрировать эти возможности на конкретном примере — создании серии связных директив для составления не очень сложных (и преимущественно текстовых) HTML-документов.

При этом я прекрасно понимаю, что сказанное далее способно вызвать у профессионального программиста лишь ироническую улыбку. Однако написал я это к вящей славе редактора NEdit. Каковой позволяет настроить себя под конкретную задачу даже людям с моим уровнем подготовки.

Итак, перед нами стоит задача составления достаточно длинных и относительно структурированных документов, предназначенных для размещения в Интернете. При этом не исключается возможность, что они в дальнейшем будут воплощены и на бумаге в виде журнальных или книжных публикаций (для чего может потребоваться импорт документа в текстовый процессор или программу верстки). Что нужно для комфортного выполнения этой работы?

Путей несколько. Можно:

- набирать текст в текстовом процессоре (типа OpenWriter из комплекта OpenOffice) с дальнейшим экспортом в формат HTML; тем, кто видел HTML-код, генерируемый текстовыми процессорами, ясно, почему этот вариант я обсуждать далее не желаю;
- набирать текст в визуальном Web-редакторе типа Mozilla Composer; однако замечание из пункта первого имеет место быть и здесь (не говоря уже о личной антипатии к Mozilla как средству редактирования Web-страниц);
- набирать текст в любом симпатичном и подходящем по возможностям HTML-редакторе с одновременной расстановкой требуемых тегов штатными его средствами; однако далеко не всегда такие редакторы хорошо приспособлены для столь "примитивной" работы; к тому же, подобно тому, как изобилие форматирующих возможностей текстового процессора нарушает течение оригинальной творческой мысли, всякого рода панели с кнопками тегов не способствуют концентрации мысли на содержательной стороне документа; к тому же, оба известных мне полнофункциональных HTML-редактора под Linux обладают одним общим недостатком: они позволяют набирать текст либо в одну строку (что неудобно визуально), либо с переносом слов через фиксированное количество знаков с образованием символа возврата каретки (что отнюдь не полезно при импорте в текстовый процессор);
- набирать текст в обычном текстовом редакторе с последующим помещением в редактор кода HTML, где и осуществляется его (текста) разметка; что, помимо сложностей с двухэтапным созданием документа, приводит к появлению двух идентичных (по идее) файлов, какой из которых является более актуальным — удается вспомнить не всегда.

И потому оптимальным представляется составление документа в текстовом редакторе с одновременной разметкой HTML-тегами. Но проставлять их вручную, даже если помнишь все наизусть, — занятие достаточно нудное.

И посему возникает естественное желание его автоматизировать. Благо, NEdit с его развитым макроязыком и простыми средствами его применения такую возможность обеспечивает.

Для начала определимся, какие теги требуются для каждодневного использования:

- заголовки нескольких (скорее всего первого-третьего) уровней;
- теги образования абзаца (`<p>`) и разрыва строки (`<br>`);
- структурные теги `<strong>`, `<emphasis>` и `<preformatted text>`;
- всякого рода списки (маркированные, нумерованные, списки с определениями) и их элементы;
- гиперссылки и именованные якоря (`<a href>` и `<a name>` соответственно).

Вот в такой последовательности и будем создавать соответствующие макросы. Включаем режим протоколирования (через меню **Macro | Learn Keystrokes**) и просто-напросто набираем с клавиатуры последовательность символов

```
<h1></h1>
```

Для обеспечения непрерывности набора логично вернуть курсор в позицию между открывающим и закрывающим тегами. После чего завершаем протоколирование (через меню **Macro | Finish Learn**), через меню **Macro Replay Keystrokes** проверяем правильность исполнения и переходим в меню **Preferences | Default Settings | Customize Menu | Macro Menu**.

В появившейся панели **Macro Commands** указываем в поле **Menu Entry**

```
Headers>header1
```

в поле **Accelerator** приписываем команде клавишу вызова (например, <F1>, благо, она не закреплена за вызовом помощи), при желании определяем мнемоническую литеру (хотя большого смысла в этом я не вижу) и смело жмем на экранную кнопку **Paste Learn/Reply Macro**. В результате в поле **Macro Command to Execute** появляется код вроде следующего:

```
insert_string("<")
insert_string("h")
insert_string("1")
insert_string(">")
insert_string("<")
insert_string("/")
insert_string("h")
insert_string("1")
insert_string(">")
backward_character()
```

```
backward_character()
backward_character()
backward_character()
backward_character()
```

Разумеется, никто не заставляет нас оставлять в неприкосновенности столь неуклюжую конструкцию. Нетрудно сообразить, что `insert_string` означает просто вставку некоей символьной последовательности. И написать проще и короче:

```
insert_string("<h1></h1>")
```

Что и будет означать вставку тега заголовка первого уровня.

А ознакомившись с примерами макросов на сайте проекта, можно по их образу и подобию создавать свои собственные, достаточно сложные конструкции.

Поскольку проверку правильности макрокоманды мы уже произвели, нажимаем **Apply** для ее включения в список и переходим к следующему пункту нашей программы. Конечно, можно повторить протоколирование набора для тегов заголовков всех уровней последовательно. Но проще с помощью экранной кнопки **Copy** скопировать только что созданный макрос, изменить в поле **Menu Entry** ее название с **header1** на **header2**, закрепить за ней клавишу <F2> и в поле **Paste Learn/Reply Macro** отредактировать код вручную, заменив строки

```
insert_string("<h1></h1>")
```

на

```
insert_string("<h2></h2>")
```

И подобную процедуру повторить для заголовков всех потребных уровней (вряд ли число их превысит четыре).

Затем переходим к созданию макросов для наиболее часто требующихся при наборе тегов параграфа и разрыва строки. Порядок действий аналогичный: протоколирование ввода тега, вызов панели **Macro Commands**, описание имени команды (с учетом ее положения в иерархическом меню), закрепление свободной комбинации "горячих клавиш", помещение "заученного" кода макроса в поле команд для исполнения, включение новой макрокоманды в список доступных. И так — для всех тегов, которые представляются нам требуемыми при повседневном использовании...

Завершив создание макросов, следует не забыть увековечить их в меню с помощью **Preferences | Save Defaults**. После чего они будут доступны в последующих сеансах NEdit, поскольку размещаются в секции

```
Nedit.macroCommands:
```

файла .Nedit.

Описанный процесс создания макросов, которых хватает в 90 случаях из 100, занимает считанные минуты. Что, опять-таки, подтверждает величие и силу редактора NEdit: можно представить себе, как способен настроить его для своих задач человек с достаточной программистской квалификацией...

Нетрудно настроить по собственному желанию и контекстное меню. Чего в нем недостает? Да тех команд, которые все равно требуют обязательного выделения мышью. Например, вставки гиперссылки. Добавляем их по тому же сценарию (через меню **Preferences | Default Settongs | Customize Menu | Window Background Menu)**.

Не забыв при этом про крайне полезную опцию — возможность открывать файл, выделив его имя в редактируемом тексте. Включив ее в наше контекстное меню, получаем возможность просмотра по серии связанных гиперссылками Web-страниц — согласитесь, такая возможность есть не в каждом специализированном HTML-редакторе...

Чего еще хотелось бы от несложного HTML-редактора? Ну конечно, визуализации производимых изменений в каком-либо внешнем браузере. Каковым, с большой долей вероятности, будет Mozilla текущей версии.

Для этого мы прибегнем к настройке меню **Shell** (**Preferences | Default Settings | Customize Menu | Shell Menu**). В соответствующей панели определяем поле имени команды; если есть желание просматривать плоды своих трудов в разных браузерах, можно создать иерархическое меню вроде

```
Browser>netscape
Browser>links
Browser>opera
```

Оставляя, скорее всего, пустыми поля **Accelerator** и **Mnemonic**, затем вводим в поле **Shell Command to Execute** строку

```
xterm -e mozilla %
```

или иную любимую терминальную программу в качестве первого элемента. И ни в коем случае не забыв параметр `%`, предписывающей Netscape отображать именно редактируемый в NEdit файл.

Теперь разбираемся с переключателями. В линейке **Command Input** логично отметить значение **None** (иначе есть риск получить по закрытии Netscape пустой файл), в линейке **Command Output** — значение **same window**. Кроме того, не худо включить также опцию записи текущего файла перед выполнением команды (**Save file before executing command**).

Все, принимаем изменения (кнопкой **Apply**) и проверяем действие команды. Если все нормально, выбор из меню **Shell** пункта **netscape** должен вызвать появление сначала терминального окна, а затем — окна браузера Netscape Navigator, отображающего редактируемый файл в текущем его состоянии. Если да — сохраняем ситуацию (через меню **Preferences | Save Defaults**) и с

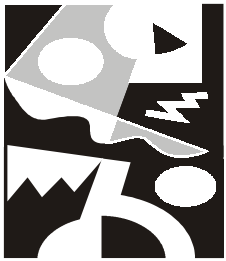
гордым и независимым видом пользуемся собственноручно созданным HTML-редактором...

Подводя итоги, можно сказать, что по своим возможностям и настраиваемости NEdit вплотную приближается к Vim или emacs. В то же время он предоставляет пользователю удобный и привычный интерфейс, лишенный архаических особенностей как emacs, так и vim.

Недостатки NEdit я уже отмечал: это отсутствие инструментальной панели и средств управления проектом. Однако первое замечание спорно, а последняя функция в элементарном исполнении, то есть в виде открытия и сохранения группы связанных файлов (глав книги, например, или серии Web-страниц), может быть реализована с помощью встраиваемых в меню сценариев оболочки и макрокоманд. Тем более что зачатки управления проектом, в виде возможности открытия файла при выделении его в редактируемом тексте, имеются.

Короче говоря, если NEdit и не отвечает в полной мере идеалу текстового редактора, то вплотную к нему приближается. Дополнительное его достоинство — отсутствие привязки к конкретной графической среде (типа KDE или GNOME).

Глава 19



Html-редактор Quanta Plus

- О Web-инструментарии вообще.
- Представление героини.
- Главные элементы интерфейса.
- Дополнительные элементы интерфейса.
- Работа с проектами.
- Настройки редактора.
- Дополнительные возможности.
- Итоги.

Составление Web-документов — частный случай работы с текстами вообще. И в прошлой главе мы приложили немало усилий для того, чтобы максимально автоматизировать этот процесс в рамках простых текстовых редакторов, таких как joe и NEdit. Однако бывают случаи, когда этого недостаточно, тогда есть смысл прибегнуть к специализированным средствам.

О Web-инструментарии вообще

Появление Web и HTML можно сравнить с изобретением полковника Кольта, уравнявшего людей сильных и слабых посредством широко известного из литературы и кино инструмента. Всемирная паутина предоставляет неограниченные возможности для творческого самовыражения. Единственное условие — умение представить свое произведение в одной из форм, для этой самой паутины приемлемых. А формой, наиболее универсальной из приемлемых, является тривиальная HTML-страница — структурированный текст, размеченный посредством соответствующих тегов.

Разметка HTML-страницы — занятие несложное, однако расставлять теги все равно нужно. И потому не замедлил появиться инструментарий для вы-

полнения этого процесса. В том числе — и ориентированный на открытые и свободные POSIX-совместимые платформы, из каковых наибольшее распространение получил Linux (в меньшей степени FreeBSD). Опять же, обращаясь к собственным воспоминаниям, замечу, что именно подготовка Web-материалов оказалась той областью, в которой мне впервые (в далеком уже 1998 г.) удалось применить мощь POSIX-систем в мирных, то есть настольно-производственных, целях.

К слову сказать, работа с HTML-материалами представляет интерес отнюдь не только для профессиональных Web-дизайнеров, Web-мастеров и прочих лиц, по долгу службы связанных с интернет-технологиями. Ибо это чуть ли не единственный формат, почти однозначно воспринимаемый на всех аппаратных платформах, во всех операционных системах, в любом национально-языковом окружении (в том числе даже и русском, с его многочисленными кодировками). И, в отличие от чистого текста, формат позволяет представлять материал в структурированном и визуально оформленном виде.

Изначально существовало два направления в технологии подготовки Web-страниц — визуальное их редактирование и редактирование HTML-кода. В первом случае порядок действий точно такой же, как и при подготовке для печати обычных текстов в программах вроде всем известного MS Word. Что легко и просто, однако далеко не всегда способно дать результат, адекватный задуманному (а при отсутствии специфического опыта — подчас прямо противоположный). И потому о визуальных редакторах речи здесь не будет.

Да, еще: все сказанное далее относится к работе с преимущественно текстовыми Web-материалами. Подготовка Web-графики — тема совершенно отдельная и здесь почти не затрагивается.

В качестве же редактора HTML-кода можно использовать обычный текстовый редактор. Их в свободном исполнении, как известно, существуют многие множества. Причем тут опять же возможны два подхода: ввод тегов в процессе набора материала или предварительная его подготовка в текстовом формате с последующей его разметкой. И то, и другое не представляет особых сложностей, но весьма занудно, так как сводится к многократному повторению рутинных действий. И потому возникает естественное желание автоматизировать этот процесс.

Напрашивающееся решение автоматизации ввода тегов в уже существующий текстовый материал — использование shell-скриптинга. Благо, среди классических UNIX-утилит нетрудно отыскать соответствующие средства, например, потоковый (неинтерактивный) текстовый редактор sed. В сочетании с утилитами поиска файлов и последовательностей символов (find и grep соответственно) он, благодаря штатным средствам POSIX-систем (механизмам конвейеризации команд и перенаправления их ввода/вывода), способен решить большинство задач HTML-разметки (и не только ее). Дополнительный плюс этого подхода — возможность пакетной разметки большого

количества Web-страниц одновременно. Тема эта вкратце была затронута в *интермедии* после *главы 12*.

Есть, однако, и минусы. Сочинение shell-скриптов вообще требует некоторых навыков, а в данном случае необходимо еще и знание особенностей собственно утилиты sed (или языка awk, также пригодного к выполнению этой задачи). Требуются и некоторые чисто программистские навыки: представление об операторах, например. По моему скромному мнению, временные затраты на сочинение сценариев HTML-разметки оправданы только в том случае, если уже имеется достаточно большой объем чисто текстовых материалов, которые необходимо претворить в Web-страницы. При сочинении же их с нуля более целесообразно автоматически вводить теги одновременно с набором текста.

И тут свободные программы (в лице обычных текстовых редакторов) окажутся небесполезными. Практически любой представитель этого семейства, функциональность которого выходит за элементарные рамки (а за точку элементарности можно принять штатный ее из FreeBSD и nano, входящий в большинство Linux-дистрибутивов), располагает собственным макроязыком программирования. Остается только сочинить соответствующие макросы и привязать их к "горячим клавишам", чтобы в дальнейшем в ответ на нажатие, скажем, клавиши <F1> получать заголовок первого уровня, клавишей <F4> оформлять параграфы (тег <p>) и т. д. Если же и макросы сочинять лень — в распоряжении пользователя всех "продвинутых" редакторов имеется возможность протоколирования собственных действий с оформлением результата в виде макрокоманд. В предыдущей главе можно найти примеры того, как легким движением руки универсальные текстовые редакторы joe и NEdit превращаются в специализированные инструменты для создания Web-страниц. А для таких гигантов текстового редактирования, как vim или emacs, дополнения для HTML-разметки придуманы давным-давно, нужно только их поискать на соответствующих сайтах (для начала, на **http://www.vim.org** и **http://www.gnu.org/software/emacs** соответственно).

Особого внимания в качестве Web-инструмента заслуживает kate — наиболее мощный из штатных текстовых редакторов KDE. Есть в kate и штатные способы полуавтоматического ввода тегов HTML (и даже XML, с проверкой валидности последних), и средства ведения проектов, и многое другое. Правда, расширение штатных возможностей этого редактора — задача нетривиальная, требующая уже умения программировать. Однако и наличных модулей расширения для kate (объединяемых в комплекс KPart) хватает для решения многих и многих задач.

Такой адаптированный текстовый редактор — прекрасный инструмент для работы с отдельными Web-страницами и небольшими их наборами типа домашних страниц. Однако плох тот "хомяк", который не мечтает стать полноценным контент-сайтом. А при работе с таковым хочется уже большего — средств ведения проектов и поддержания их целостности, как минимум.

Конечно, и эта проблема решаема штатными POSIX-средствами. Все уважающие себя текстовые редакторы POSIX-мира позволяют запускать внутри себя команды оболочки и командные конструкции, в том числе и пользовательские сценарии. И ничто не мешает сочинить набор скриптов для открытия группы объединенных в проект файлов, проверки целостности внутренних ссылок, предварительного просмотра в распространенных браузерах и прочих функций, ожидаемых от средств ведения проектов. Кроме одного: а не изобретаем ли мы тем самым велосипед? И не сделал ли это кто-нибудь до нас?

Поскольку пользователь POSIX-систем стоит на плечах гигантов, ответ, разумеется, будет положительным. Остается только такие средства отыскать. К счастью (или к сожалению?) полноценных редакторов HTML-кода для свободных платформ не так и много — всего три. Это bluefish и screem, базирующиеся на библиотеке Gtk, и Quanta Plus для работы в среде KDE. Именно о последней и пойдет речь.

Представление героини

Итак, Quanta Plus — программа, позиционируемая как универсальное средство Web-разработчика. Это относительно недавнее достижение свободной мысли: первые ее версии появились в начале 2000 г. В числе разработчиков ее норвежец Эрик Лаффон (Eric Laffoon) и два наших соотечественника, хотя и бывших — из Украины: Дмитрий Поплавский и Александр Яковлев (рис. 19.1).

Некоторое время quanta развивалась как самостоятельное приложение, затем в виде отдельного пакета была включена в штатный комплект KDE. Ныне (начиная с KDE версии 3.3.1) quanta — главная составляющая KDE-пакета kdewebdev, включающего также еще несколько утилит этого профиля (о некоторых я расскажу в конце этой главы).

Об установке quanta много говорить не стоит, ее можно инсталлировать штатными средствами конкретного дистрибутива вместе со всей средой KDE или отдельными ее компонентами. Нет препятствий и для ручной сборки пакета из исходников, предварительно нужно только озаботиться установкой kdebase и kdelibs (прочие зависимости опциональны и будут выведены по завершении исполнения конфигурационного сценария).

Запуск quanta также элементарно прост и может быть выполнен:

- из K-меню (через пункты **Разработка | Среда Web-разработки (Quanta Plus)**);
- из строки мини-терминала (пункт K-меню **Выполнить программу**);
- из терминального окна (в двух последних случаях нужно просто набрать в командной строке `quanta`).

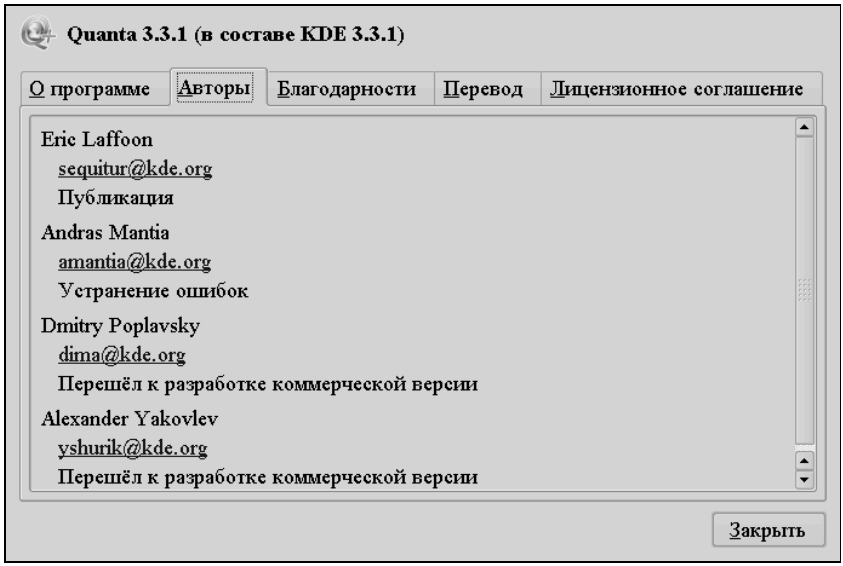


Рис. 19.1. Приятно, что в числе разработчиков quanta есть и наши соотечественники, хотя и бывшие

Иконку для запуска quanta можно вынести на рабочий стол или в панель задач KDE. Можно и просто открыть HTML-файл в редакторе, щелкнув в konqueror на нем правой кнопкой, выбрав из контекстного меню пункт **Открыть в | Quanta Plus**.

По своему запуску quanta обеспечивает обычные для программ этого класса возможности набора и редактирования HTML-кода: автоматический ввод основных тегов и их атрибутов, подсветку синтаксиса, предварительный просмотр Web-страницы и т. д. Весьма развиты просто средства обработки текстов — поиск и замена (в том числе с использованием регулярных выражений), проверки орфографии. Из Web-специфичных "продвинутых" возможностей стоит отметить, во-первых, уже упоминавшиеся средства управления проектами (удачно дополняемые интегрированным файловым менеджером, представляющим собой облегченный вариант konqueror) и особенно визуальный редактор, позволяющий выполнять HTML-разметку методами, привычными по работе с текстовыми процессорами.

Допускает quanta также и весьма изощренные приемы работы — с языком разметки XML и стилевыми таблицами, сценариями PHP и многим другим, необходимым для профессионального Web-мастера. Однако я сконцентрирую свое внимание на том, что этот редактор дает народу (то есть простому пользователю) и как его настроить оптимальным образом для набора HTML-документов.

Главные элементы интерфейса

Функции обработки HTML-материалов реализованы в quanta посредством весьма специфического интерфейса. Рисунок 19.2 показывает примерный вид редактора по умолчанию. Можно видеть, что собственно рабочее поле (а режим прямого редактирования здесь основной) обрамлено множеством инструментальных панелей и прочих интерфейсных элементов.

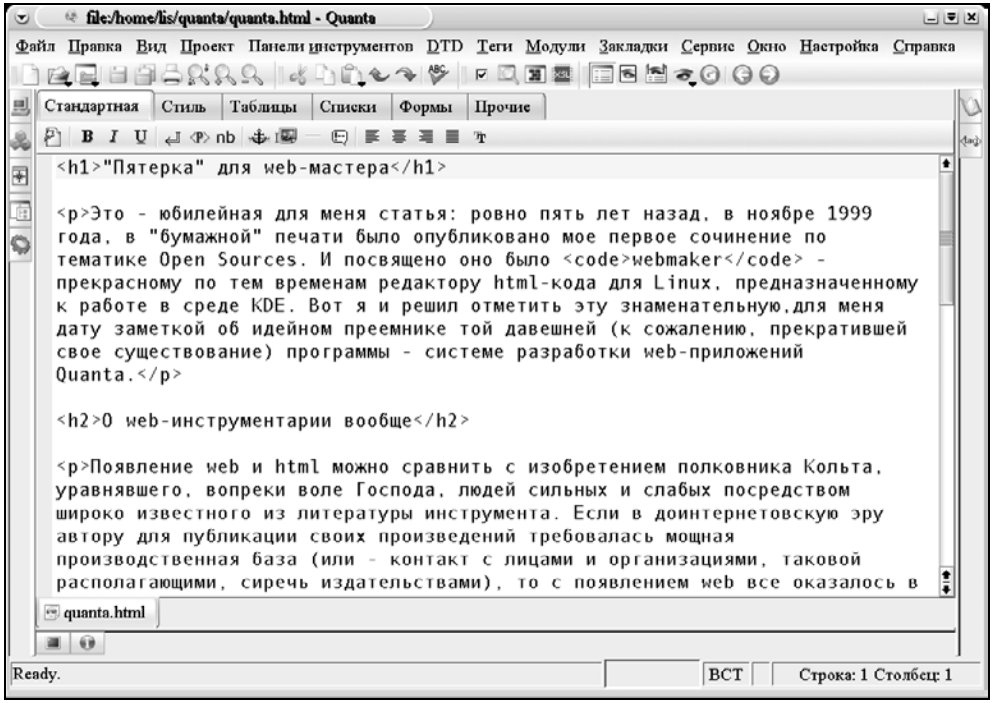


Рис. 19.2. Web-редактор quanta — интерфейсные элементы по умолчанию

Вдоль верхней границы рабочей области имеем (сверху вниз):

- строку главного меню;
- главную инструментальную панель;
- панель вкладок, в которые сгруппированы теги;
- соответствующую каждой закладке собственно панель тегов.

О главном меню можно говорить или очень много, или ничего. Поэтому отмечу только, что через него осуществляется доступ ко всем функциям программы, которые в принципе доступны. Ну, а смысл каждого пункта и

подпункта ясен либо из его названия, либо (в редких случаях) устанавливается методом ползучего эмпиризма.

Не буду распространяться и на тему главной инструментальной панели: при наведении на любую ее пиктограмму всплывает исчерпывающая (и русскоязычная) подсказка.

А вот о вкладках и соответствующих панелях тегов (в терминологии quanta они именуются пользовательскими панелями инструментов) стоит сказать подробнее: именно через них мы в дальнейшем будем выполнять тонкую индивидуальную настройку редактора.

По умолчанию вкладок шесть:

- **Стандартная**;
- **Стиль**;
- **Таблицы**;
- **Списки**;
- **Формы**;
- **Прочие**.

Смысл их более-менее понятен из названий, однако некоторые комментарии все же лишними не будут.

Панель **Стандартная** объединяет элементы, используемые постоянно в ходе разметки тела HTML-страницы: параграфы и разрывы строки, гиперссылки и вставки изображений, выделения и выравнивания. Стоит отметить, что полужирному и курсивному выделению соответствуют не визуальные теги `<b>` и `<i>`, а структурные теги `<strong>` и `<emphasis>`, а выравнивание достигается значениями атрибутов тега `<div>`. Вообще, весьма точное следование букве спецификаций W3C — одна из характерных черт редактора quanta.

В панель **Стиль** объединены теги для заголовков (с 1 по 4 уровень), преформатированного текста, верхних и нижних индексов, цветового выделения, а также для работы со стилевыми таблицами (CSS).

Пиктограммы панели **Таблицы** позволяют создать таблицу целиком (**Редактор таблиц**), с требуемым числом строк и колонок, с заголовком, шапкой, примечаниями и даже данными. Возможно и поэлементное создание таблицы в абсолютно любой последовательности.

Панель **Списки** предназначена для создания именно этих элементов HTML-разметки — нумерованных и ненумерованных списков (ordered lists и unordered lists соответственно) и их элементов (list items), а также списков определений (definition lists).

Панель **Формы** служит для создания простых интерактивных элементов Web-страницы: форм, выпадающих меню, переключателей и т. д.

Наконец, в панель **Прочие** попало несколько пиктограмм, не охваченных в предыдущих вкладках, как-то: вставка даты/времени, ссылки на адрес электронной почты, метатегов, мнемонических кодов для замены специальных символов.

Очень важна кнопка **Прочие теги**. Во-первых, внимательный просмотр пользовательских панелей и пункта **Теги** главного меню показывает, что все многообразие тегов современного HTML не охвачено ни там, ни там. И потому, если потребуется вставить тег типа `<code>`, придется обратиться к этой кнопке. Во-вторых, элементы XML-разметки в quanta также не предусмотрены, их на первых порах придется задавать посредством пиктограммы **Прочие теги**. Хотя в дальнейшем мы увидим, что и ту, и другую операцию легко автоматизировать.

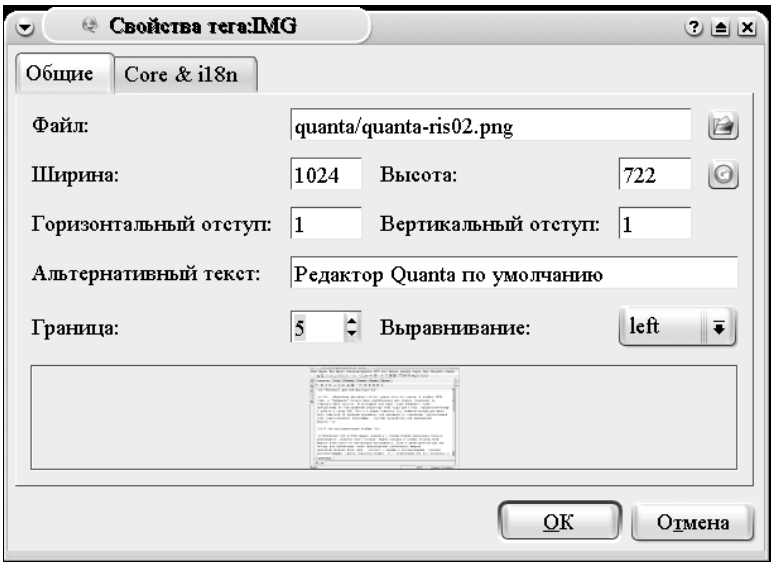


Рис. 19.3. Указание значений атрибутов для тега `<img>` при вставке изображения в HTML-страницу

Все кнопки пользовательских панелей можно разделить на две группы: обычные и диалоговые. Нажатие первых вызывает вставку простого тега (при необходимости — как отрывающего, так и закрывающего), например, параграфа или разрыва строки. Диалоговые же кнопки связаны с тегами, требующими (или допускающими) указания атрибутов и их значений. В этом случае вызывается диалоговая панель, в соответствующих полях которой можно указать требуемые параметры, как это можно видеть на примере тега `<img>` на рис. 19.3.

Редактировать значения атрибутов тегов можно и иным способом: достаточно щелкнуть правой кнопкой на теге, допускающем указание таковых, и в появившемся контекстном меню выбрать пункт **Редактировать тег** для вызова той же диалоговой панели.

Дополнительные элементы интерфейса

Основных элементов интерфейса (меню, главной панели и пользовательских панелей) вполне достаточно для разметки HTML-страницы. Дополнительные же элементы (панели, обрамляющие рабочую область с трех остальных сторон) призваны повысить комфортность работы либо же служат специальным целям.

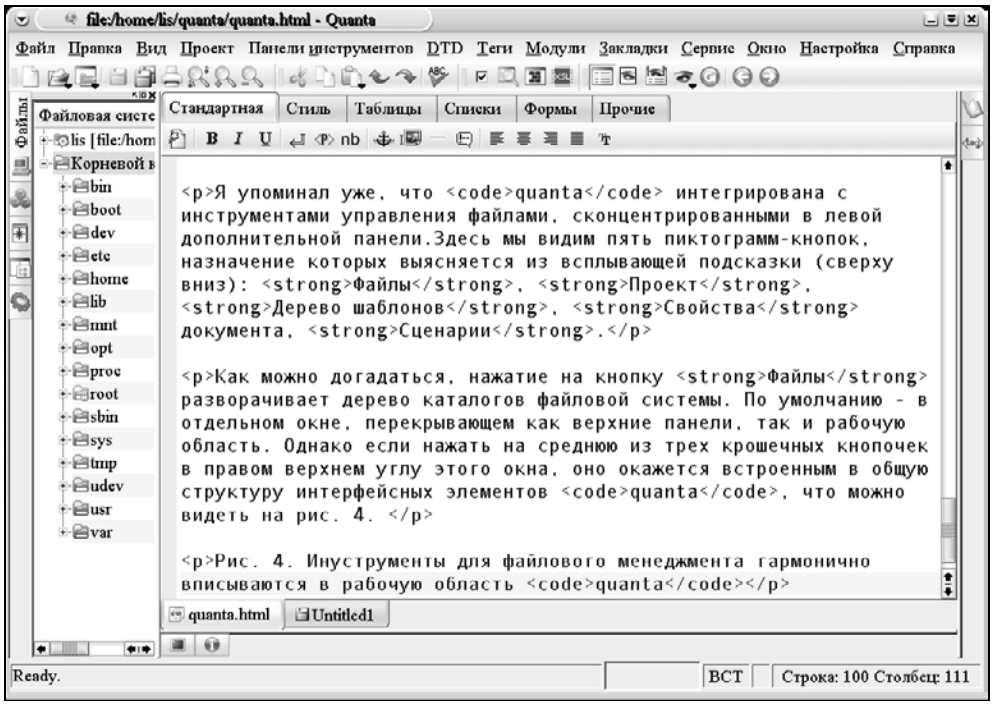


Рис. 19.4. Инструменты для файлового менеджмента гармонично вписываются в рабочую область quanta

Я упоминал уже, что quanta интегрирована с инструментами управления файлами, сконцентрированными в левой дополнительной панели. Здесь мы видим пять пиктограмм-кнопок, назначение которых выясняется из всплывающей подсказки: **Файлы**, **Проект**, **Дерево шаблонов**, **Свойства** документа, **Сценарии**.

Как можно догадаться, нажатие на кнопку **Файлы** развертывает дерево каталогов файловой системы. По умолчанию — в отдельном окне, перекрывающем как верхние панели, так и рабочую область. Однако если нажать на среднюю из трех кнопок в правом верхнем углу этого окна, оно окажется

встроенным в общую структуру интерфейсных элементов quanta, что можно видеть на рис. 19.4.

Манипуляции с файлами в одноименном окне осуществляются через контекстное меню, вызываемое щелчком правой кнопки мыши. Здесь можно: скопировать, удалить и переименовать файл, а также каталог, просмотреть их свойства (через этот пункт меню возможен также просмотр атрибутов принадлежности и доступа и, при наличии соответствующих привилегий, их изменение).

Кнопка **Проект** выводит в том же окне (автономном или встроенном) дерево файлов проекта. Свойства же прочих кнопок панели управления файлами, как и панелей, обрамляющих рабочую область снизу и справа, читателю предлагается изучить в качестве домашнего задания.

Работа с проектами

Поддержка проектов — один из необходимых атрибутов развитого HTML-редактора: как я уже говорил, все прочие его функции вполне можно смоделировать в редакторе обычном.

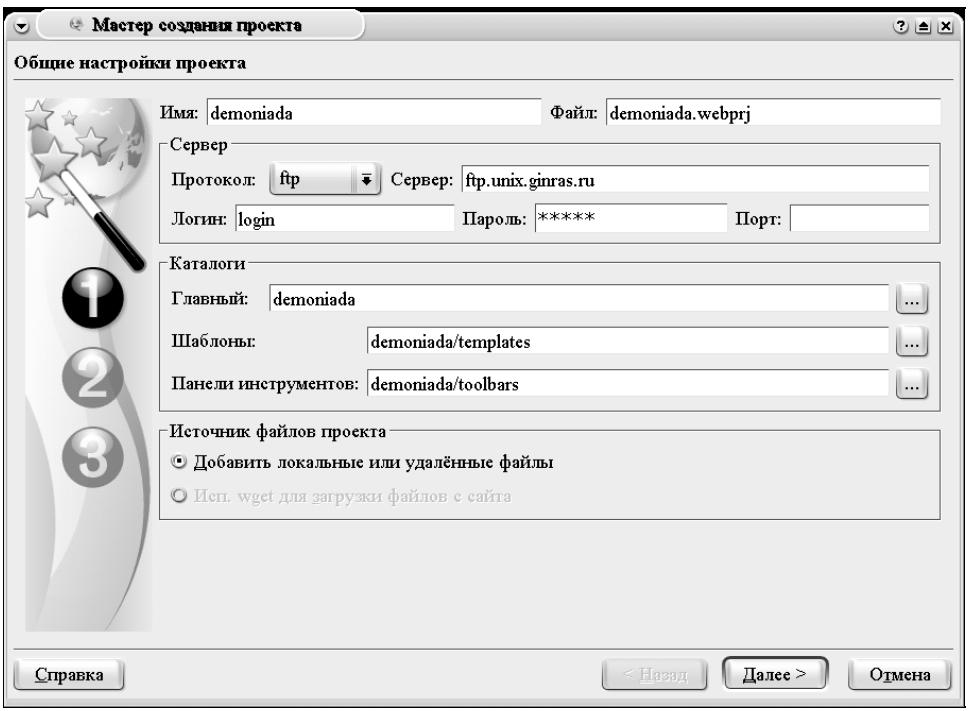


Рис. 19.5. Создание проекта начинается с указания его имени, главного каталога и каталогов шаблонов и пользовательских панелей

И quanta такую поддержку обеспечивает, хотя и не в том объеме, как этого бы хотелось.

Для создания проекта следует обратиться с одноименному пункту главного меню — к его подпункту **Новый проект**, вызывающему своего рода мастер проекта. Первой из диалоговых панелей (рис. 19.5) задается имя проекта, сервер с его протоколом (например, ftp, можно, разумеется, создать и проект на локальной машине, без подключения к сети), главный каталог и подкаталоги для шаблонов и пользовательских панелей (о которых речь пойдет дальше). В следующей панели, при необходимости, можно добавить в проект уже существующие файлы — все из данного каталога или по определенной маске (например: `*.html`).

Затем (рис. 19.6) автор может указать свое имя и адрес электронной почты, а также раз и навсегда определить DTD (Document Type Definition — объявление типа документа) всех файлов проекта и их кодировку (я, кажется, забыл отметить, что quanta прекрасно работает со всеми кодировками русского языка, включая UTF8, но исключая cp866).

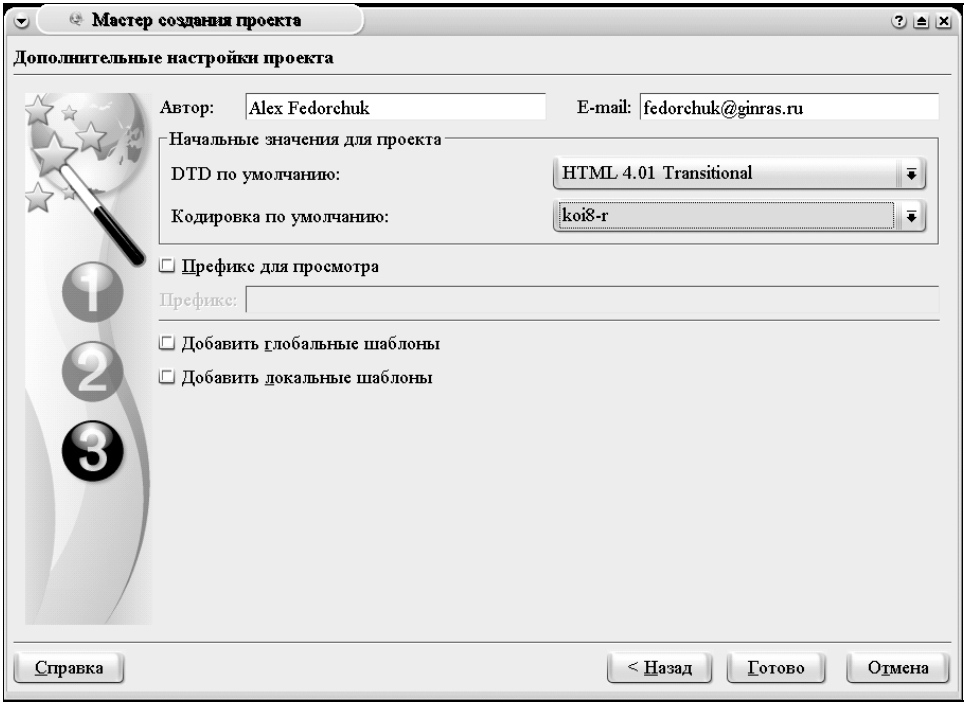


Рис. 19.6. Следующий шаг — задание общего DTD и кодировки для документов проекта

Главное, что дает создание проекта — это возможность определить для него шаблоны и пользовательские панели инструментов. Касаться шаблонов я здесь не буду — это, как уже было сказано, тема домашнего задания. А о пользовательских панелях речь пойдет в скором времени.

Настройки редактора

Ибо настало время обратиться к настройкам quanta. Черновая настройка редактора выполняется через одноименный пункт главного меню — подпункт **Настроить Quanta**, вызывающий конфигурационную панель с несколькими пунктами (рис. 19.7). Содержание их вполне очевидно, и задерживаться на нем я не буду.

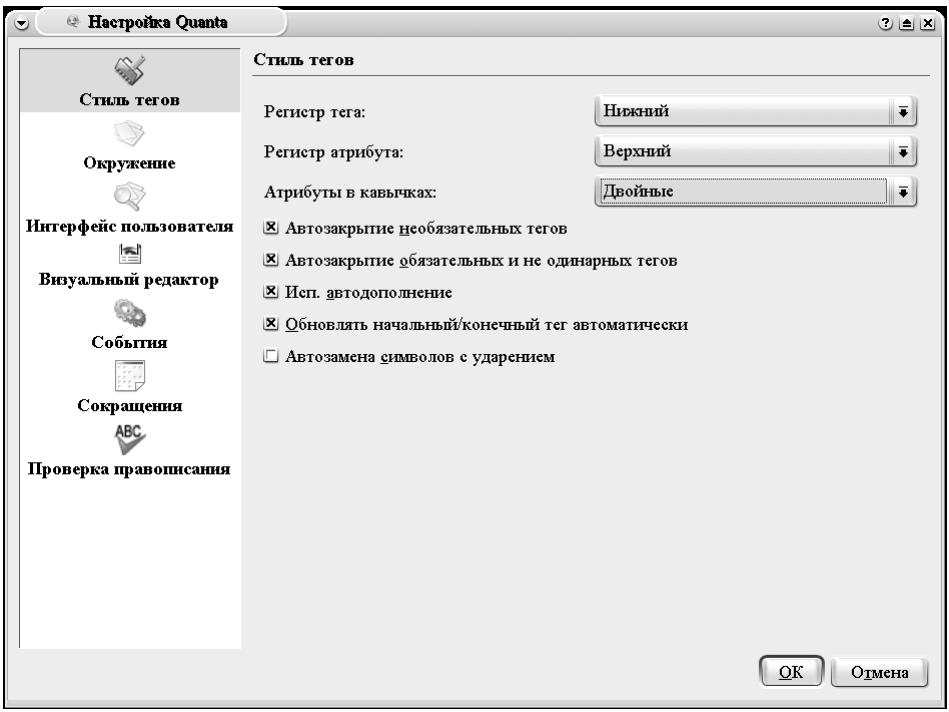


Рис. 19.7. Общие настройки редактора quanta — вещь достаточно очевидная для умеющих читать (даже только по-русски)

Отдельный момент настройки — подпункт **Настроить редактор** в том же пункте **Настройка главного меню**. Здесь также все достаточно ясно, если внимательно ознакомиться с пунктами соответствующей панели (рис. 19.8).

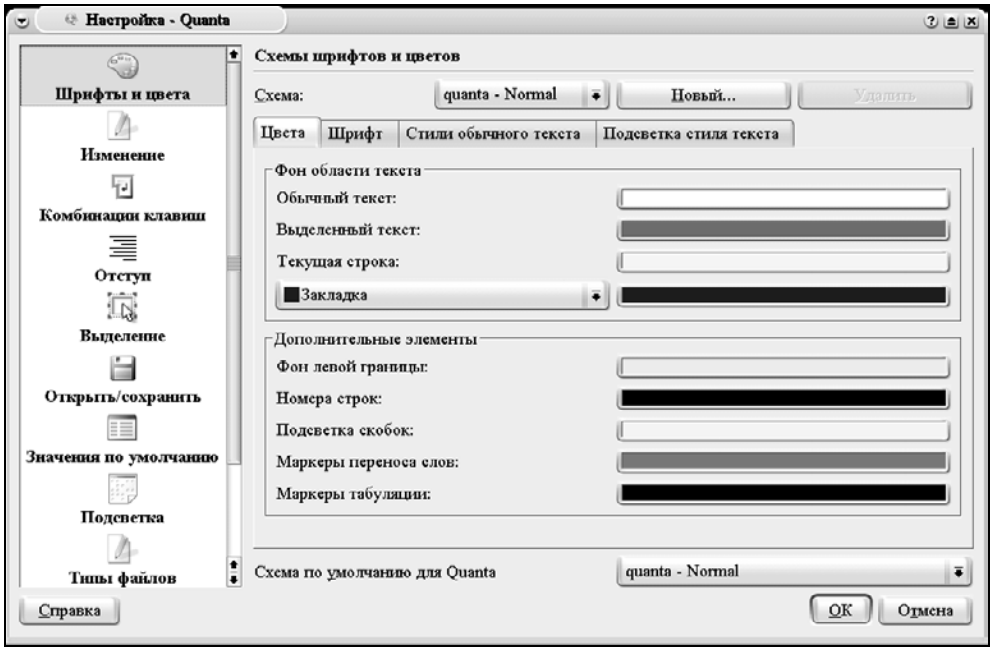


Рис. 19.8. Настройка редактора — также дело не очень сложное

Помнится, я обещал, что quanta позволяет автоматизировать ввод тегов, в том числе и тех, которые не предусмотрены штатными средствами — через меню или пользовательские панели. Однако внимательный читатель ни в одном из пунктов меню **Настройка** не обнаружит ни определений собственных тегов, ни средств автоматизации их расстановки. Не просматривается и удобного способа для задания элементов XML-разметки. Что же делать?

Оказывается, просто-напросто обратиться к пункту главного меню, именуемому **Панели инструментов** — именно таким образом настраивается все то, о чем идет речь.

В указанном пункте обнаруживаются такие варианты:

- **Загрузить** — вызов одной из уже существующих пользовательских панелей, глобальных — подкаталог share/apps/quanta/toolbars/ в корневой директории KDE (например, /opt/kde или /usr/local/kde), локальных, принадлежащих данному пользователю ($HOME/.kde/share/apps/quanta/toolbars/) или панелей проекта (помещаемых в каталог /path_to_proj/toolbars); по умолчанию загружены все доступные панели из числа глобальных, а локальных панелей и панелей проекта пока в природе не существует;

- **Сохранить** — сохранение вновь созданной панели в качестве локальной или панели проекта; очевидно, что сохранять и изменять глобальные панели обычный пользователь не может;
- **Добавить пользовательскую панель** — это именно то, чем мы вскоре займемся;
- **Удалить пользовательскую панель** — это не удаление как таковое, а лишь дезактивация одной из загруженных (по умолчанию или через подпункт **Загрузить**) панелей;
- **Переименовать пользовательскую панель инстументов** и **Отправить панель инструментов по E-Mail** — в комментариях не нуждаются;
- **Загрузить панель инстументов** — скачивание из Интернета (с узла **quanta.kdewebdev.org**) какой-либо ранее разработанной панели.

Очевидно, что наши действия должны начинаться с пункта **Добавить пользовательскую панель**. В ответ нам будет предложено задать имя панели (по умолчанию **User_0**). После этого в число вкладок добавится новая — с соответствующим именем, а соответствующая ей панель инструментов будет пуста, остается лишь наполнить ее содержимым.

Для этого нам придется отправиться в пункт **Настройки** главного меню и выбрать там подпункт **Настроить панели инструментов**. После этого взору предстает следующее окно (рис. 19.9).

Вверху окна выпадающее меню, в котором перечислены имена всех активных (то есть загруженных в данный момент) панелей. В левом окне — перечень доступных действий; повторяю, они соответствуют кнопкам загруженных пользовательских панелей (и подпунктам **Теги** главного меню). А правое содержит действия, кнопки для которых включены в текущую пользовательскую панель.

Так что для наполнения новообразованной панели нужно выбрать ее имя в выпадающем меню; очевидно, что правое "окошко" при этом окажется пустым, и в него просто перетаскиваются пиктограммы для требуемых действий.

Ясно, что таким образом можно только перекомпоновать кнопки тегов, которые и так были ранее доступны. А где же обещанная возможность создавать теги собственные? Да рядом, в подпункте **Настроить действия** пункта **Настройка** главного меню. В появившемся диалоговом окне выбирается тип действия кнопки (кроме тега, к кнопке пользовательской панели можно привязать также сценарий и просто текст), задать ее название и всплывающую подсказку, а также ввести собственно тег (при необходимости также и закрывающий), как это показано на рис. 19.10. Для тегов, допускающих ис-

пользование атрибутов и их значений, можно поставить переключатель запуска диалога, то есть создать диалоговую кнопку.

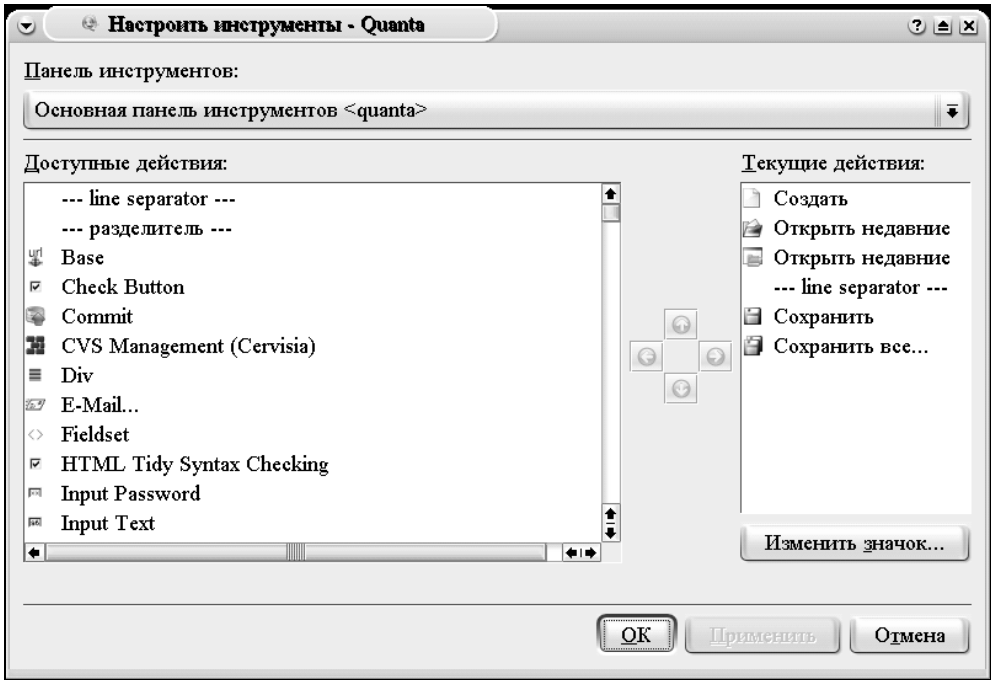


Рис. 19.9. Настройка пользовательских панелей инструментов

Вновь созданной кнопке будет сопоставлен некий значок. Который несложно изменить через пункты меню **Настройка | Настроить панели инструментов**.

Заполнив новую панель инстументов кнопками для требуемых тегов, остается только сохранить ее либо как локальную, либо как панель проекта. Первый вариант понятен, нужно только записать файл панели (а он имеет вид name.toolbar.tgz, то есть представляет собой сжатый архив) в нужный подкаталог. Для HTML-панелей это будет $HOME/.kde/share/apps/quanta/toolbars/html. А вот зачем нужны собственные панели в отдельном проекте?

Ответ не сложен. Представьте себе, что вы ведете два сайта, один в стиле pure html, другой же на базе XML-технологий. Очевидно, что в том и ином случае потребуются совершенно разные наборы тегов (а теги XML создавать в quanta так же легко, как и обычные HTML-теги), объединение которых в общих пользовательских панелях приведет только к их загромождению. Тут то и стоит вспомнить о возможности создания собственного комплекта панелей для каждого проекта.

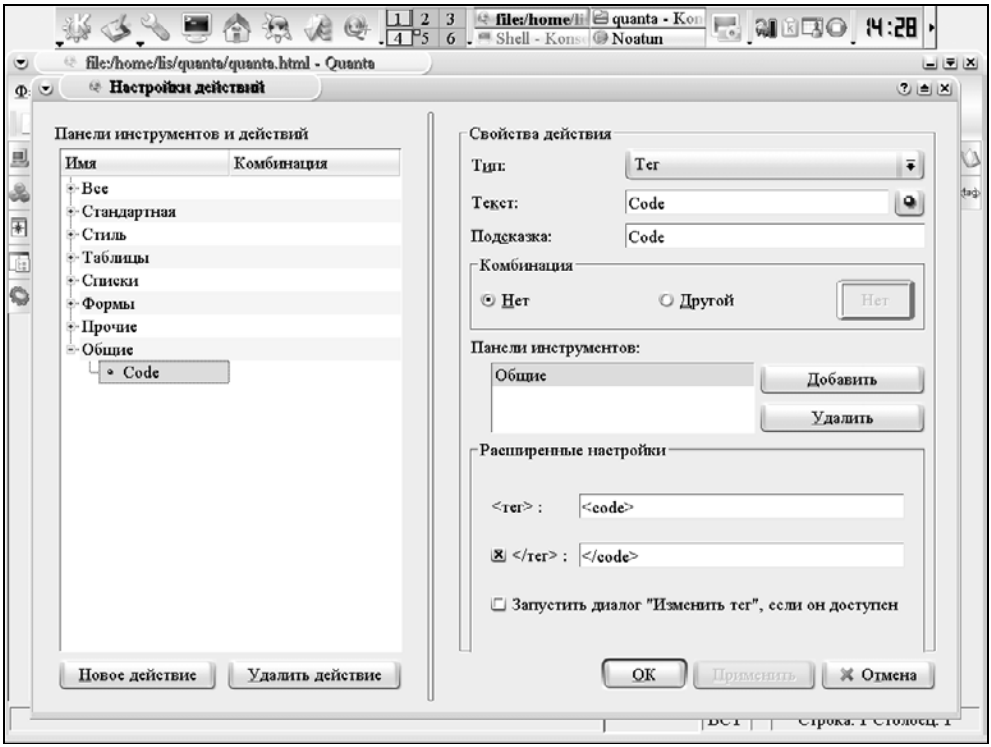


Рис. 19.10. Задаем атрибуты вновь создаваемой кнопки новой пользовательской панели

Все это, конечно, очень благородно, — скажете вы мне. Но ведь предлагалось использовать quanta не просто для разметки уже существующего текста, а для его набора — с автоматическим вводом тегов в процессе оного. А разве это верх удобства — отрывать при наборе руки от клавиш и тянуться к мыши для того, чтобы ввести тег нажатием на кнопку инструментальной панели? Отнюдь, — скажет любой пользователь с навыками ремингтониста (это мужской род от слова "машинистка", на заре машинописи работа эта считалась столь тяжелой, что ее исполняли исключительно мужчины). И будет, безусловно, прав. Так что пора исполнить обещание и рассказать, как же вводить теги исключительно с клавиатуры, без всякого участия мышей и прочих грызунов.

А сделать это можно соответствующей настройкой "горячих клавиш". Для чего опять отправляемся в меню **Настройка | Настроить действия**, обращая внимание на блок **Комбинации** соответствующего диалогового окна (рис. 19.10). В блоке этом мы видим два переключателя: **Нет**, отмеченный по умолчанию, и **Другой**. Его-то и следует отметить, что вызовет появление еще

одного окна (рис. 19.11). Тут достаточно нажать ту клавишу (или комбинацию их), к которой хотелось бы привязать ввод данного тега — и в дальнейшем она будет выступать в качестве "горячей", действие которой эквивалентно нажатию экранной кнопки на пользовательской панели.

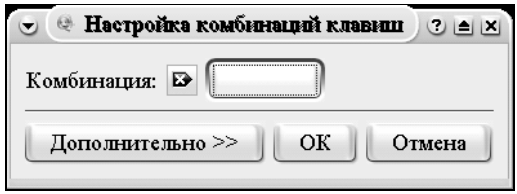


Рис. 19.11. Привязка клавишных комбинаций

Теперь остается самая малость — обеспечить загрузку созданной пользовательской панели при старте quanta, само собой это не произойдет. Тут потребуются некоторые действия руками, хотя и несложные. Во-первых, в домашнем каталоге должен присутствовать файл descrition.rc, он размещается в подкаталоге $HOME/.kde/share/apps/quanta/dtep, соответствующем нужному DTP (например, html-transitional). Проще всего скопировать его прототип из opt/kde/share/apps/quanta/dtep/HTML-transitional и чуть отредактировать. А именно — отыскать в нем (в секции `[Toolbars]`) строку вида

```
Names = standard, style, tables, lists, forms, other
```

и дописать туда имя файла новой пользовательской панели — через запятую и пробел, без суффиксов, например — `mybar`.

Я привел лишь простой рецепт создания собственной пользовательской панели и обеспечения ее загрузки, без объяснения механизма действий. Для более глубокого понимания оного можно ознакомиться с документацией по редактору quanta, вызываемой через меню **Справка | Руководство "Quanta"**. Здесь же описываются и принципы работы с DTP, знание которых необходимо для решения более сложных задач — полноценного использования XML, стилевых таблиц, PHP-сценариев и прочего высшего Web-пилотажа.

Дополнительные возможности

Внимательный читатель обратил внимание, что в редакторе quanta, при всем богатстве его возможностей, не обнаруживается такого важного для ведения крупных Web-проектов средства, как контроль целостности ссылок, ни внутренних, ни тем более внешних. Это действительно так. Однако вспомним, что quanta — лишь один (хотя и главный) из компонентов пакета

kdewebdev. Остается ознакомиться с его возможностями: не найдем ли мы там чего-либо недостающего для полного счастья?

И конечно же, найдем: программу klinkstatus, именно для проверки ссылок и предназначенную. Запускаем одноименной командой ее из командной строки или мини-терминала (штатно в K-меню она отсутствует, хотя никто не мешает ее туда встроить) и видим окно следующего вида (рис. 19.12). Отправляемся в меню **Файл | Открыть URL**, выбираем индекс-файл нашего сайта, устанавливаем требуемую глубину вложенности подкаталогов, жмем кнопку **Проверить** — и через некоторое время получаем полный список всех ссылок, как работающих (отмеченных зелеными флажками), так и оборванных (красные кресты). Последними на машине, в данный момент не подключенной к Интернету, будут все внешние ссылки. Проверку коих при необходимости можно исключить, сняв соответствующий переключатель.

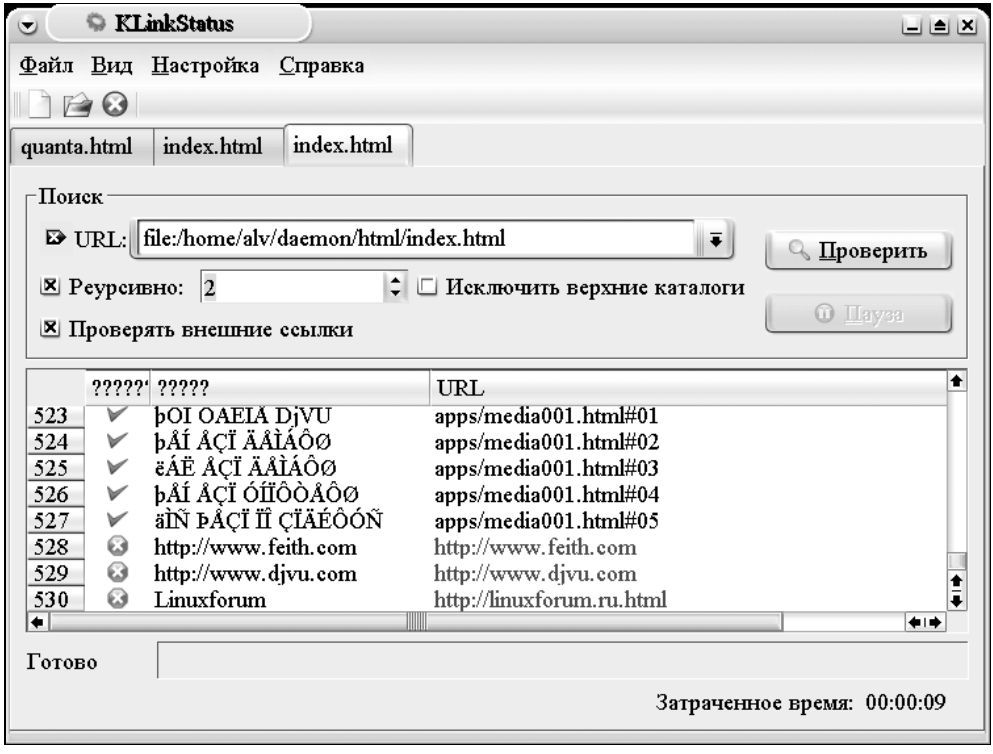


Рис. 19.12. Проверка целостности ссылок утилитой klinkstatus — обращаем внимание на глубину рекурсии и переключатель проверки внешних ссылок

Из рисунка можно видеть главный недостаток текущей версии klinkstatus — русские заголовки страниц предстают в виде абракадабры. Впрочем, на функциональности собственно проверки ссылок это не отражается.

Программа klinkstatus не блещет богатством настроек (рис. 19.13), хотя все жизненно необходимое тут присутствует. А именно, задание глубины рекурсии, включения/отключения проверки внешних ссылок, число адресов в истории и т. д.

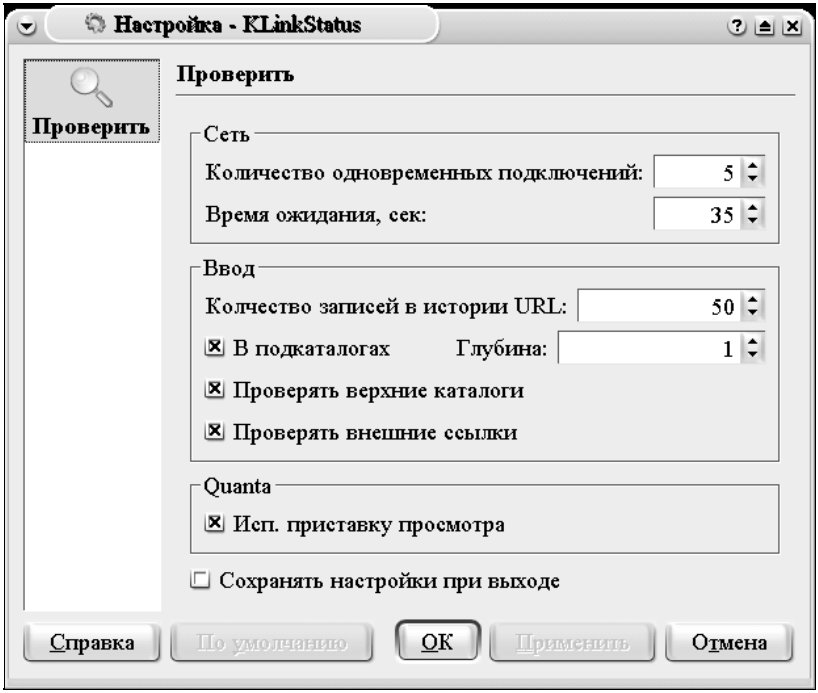


Рис. 19.13. Настройка klinkstatus

В составе пакета kdewebdev есть еще несколько полезных утилит. Например:

- kimagemapeditor — программа для создания карт изображений, то есть разбиения рисунка на отдельные области, к каждой из которых привязана гиперссылка;
- kmdr-editor — редактор диалогов;
- xsldbg — отладчик XSL, работающий в командном режиме, и графический интерфейс к нему — kxsldbg.

Однако их рассмотрение далеко выходит за рамки темы этой книги. Замечу только, что все эти средства могут быть и интегрированы в quanta в качестве

модулей. По умолчанию это сделано для klinkstatus, kimagemapeditor и xsldbg. То есть они могут вызываться из меню редактора — через пункт **Модули** и далее **Link Checker**, **KImageMapeditor** или **XSLT Debugger** соответственно.

В меню **Модули** можно обнаружить еще один пункт **KFileReplace**, пользу которого трудно переоценить. Он позволяет выполнить поиск и замену текстовых фрагментов в группе файлов (например, в каталоге, включая вложенные подкаталоги) в пакетном режиме. Каждый, кому приходилось менять адрес электронной почты Web-мастера на сотнях страниц своего сайта, проникнется величием такой возможности.

Допускается в quanta и подключение дополнительных модулей — через меню **Настройка | Настроить модули**. Правда, для этого их кто-то должен написать. Но в качестве дополнительных модулей могут использоваться составные KPart из kate, а их пишут довольно активно.

Итоги

Я коснулся лишь некоторых особенностей программы quanta — тех, которые наиболее важны для меня лично. За чертой рассмотрения остались такие вещи, как работа с XML, стилевыми таблицами, сценариями (то, что требуется профессионалу в области Web-технологий, к каковым себя не причисляю). А потому подведу предварительные итоги.

Думаю, мне удалось продемонстрировать как широту возможностей описываемого редактора (особенно в сочетании с дополнительными модулями), так и гибкость его индивидуальных настроек. Конечно, мне давно не приходилось видеть редакторов HTML-кода для Windows, однако по смутным воспоминаниям о HomeSite, как будто бы в нем не было ничего такого, что невозможно было бы реализовать в quanta, штатными ли ее средствами, или с помощью модулей. Включая даже режим визуального редактирования, о котором я практически не говорил — ввиду тривиальности приемов работы в нем.

Так что думается, что значение этой программы выходит за рамки сочинения любительских Web-страниц — это вполне полноценный и профессиональтный инструмент Web-мастера. Ну, а его возможности по составлению HTML-документации — просто выше всяких ожиданий. В частности, окончательная версия этой книги доводилась до ума именно в Quanta Plus.

Вместо заключения

Вот и подошел к концу разговор о свободных POSIX-системах. За рамками настоящей книги осталось множество вопросов, в первую очередь касающихся пользовательских приложений. Ибо описание их в подробностях потребовало бы еще одного тома — мир приложений Open Source воистину необъятен. Да и описание их — занятие весьма скучное и неблагодарное. Во-первых, они постоянно меняют свои версии, что связано и с изменением функциональности. Во-вторых, со временем одни проекты отмирают (или их программы просто теряют популярность), сменяясь другими. И, скорее всего, все написанное устареет раньше, чем эта книга выйдет из печати.

В компенсацию несказанного замечу, что материалы о пользовательских приложениях для свободных POSIX-совместимых систем регулярно появляются на сайте **http://www.posix.ru**, к которому и предлагаю обращаться заинтересованным читателям. Кроме того, много информации любознательный пользователь может найти на сайтах **http://www.linuxcenter.ru** и **http://www.citforum.ru**.

Эта книга была уже написана и готовилась к печати, когда мне удалось ознакомиться с сочинением Эрика Реймонда "Искусство программирования для UNIX" в русском переводе. В которой я обнаружил множество близких мне мыслей. И если бы я прочитал его раньше, кое-что в моей книге было бы написано по-другому. Что ж, остается ждать, что кто-нибудь продолжит этот труд, написав книгу "Искусство использования UNIX".

Предметный указатель

A

ACPI 381
Archlinux 87

B

Base POSIX 49
Bourne Shell 423
BSD Installer 264
BSD-утилиты 123

C

Cfdisk 252
Chunk 270
CLI 299
Coreutils 161

D

Directory 147
DragonFly 292, 477

E

Emacs 357
Environment variable 319
Ext2fs 97, 375

F

Fdisk 251
Fedora 97
FFS 225
Free Software Foundation 22
Front-end 255

G

Gcc 415
General Public License 25
Gentoo Linux 90
GID 168
GNU nano 587
GRUB 256

I

IceWM 523
Info 257
Inode 144, 147

K

KDE 27
Kernelland 44
Keybindings 321
Knoppix 54
Konqueror 178
Krename 584

L

Lilo 101, 368
Linking 396
Linux native 254
Logical Partitions 212
Logical Volume 212
Login shell 138, 401, 603
Log-файлы 130

M

Major 152, 166
Man 257

Mandrake 85
Man-страница 342
Midnight Commander 178
Minix 71
Minor 152, 166
Mirroring 219
Mount 280
MoviX 82

N

Named pipe 153
NEdit 586
NICE 137
Nice value 142
NLS 401

O

Open Source 17
Open Systems 17

P

Parted 252
Partition Magic 252
Partition Types 212
Patchkit 74
PCF 498
Precommand modifier 443
Preferences 522
Procfs 153

R

RAID 264
Raidtools 265
ReiserFS 223, 257, 369
Rpm 419
Run depends 394
Runlevels 385, 490

S

Sfdisk 251
Sha-bang 338
Slackware Linux 28
Source Based 54, 99, 243
SVGAlib 292
Swap 284
Symlinks 150
Sysinstall 264
System V 223

U

UFS 226
UID 168
Userland 44

V

Vinum 269

W

Winkeys 476
Word completion 433

X

XBD 50
XCU 50
XENIX 68
XRAT 50
XSH 50

А

Автокоррекция 440
Атрибут учетной записи 113

Б

Библиотека:
  Glib 533
  Gtk 528
  Qt 529
  Gtk 644
Бит суидности 170
Браузер Opera 27

В

Временные атрибуты 172

И

Идентификаторы 145
Инверсия поиска 362
Интерпретатор языка команд 46

К

Командные редакторы 586
Контент файлов 349
Корневая файловая система 240
Кривые Безье 497

Л

Локаль 492

М

Микроядро Mach 59, 69
Монтирование 251

О

Оффсет 263

П

Партиции 259, 263
Пользовательские акаунты 121
Портообразные системы 420
Потоковое редактирование 362

Р

Растровый шрифт 498
Режим протоколирования
  макрокоманд 604

С

Символы разметки TeX 359
Симлинк 180
Сокеты 153

Т

Тарбалл 393
Текстовая консоль 287
Точки монтирования 190, 248

Ф

Файлы устройств 151

Э

Эмуляция SCSI 193